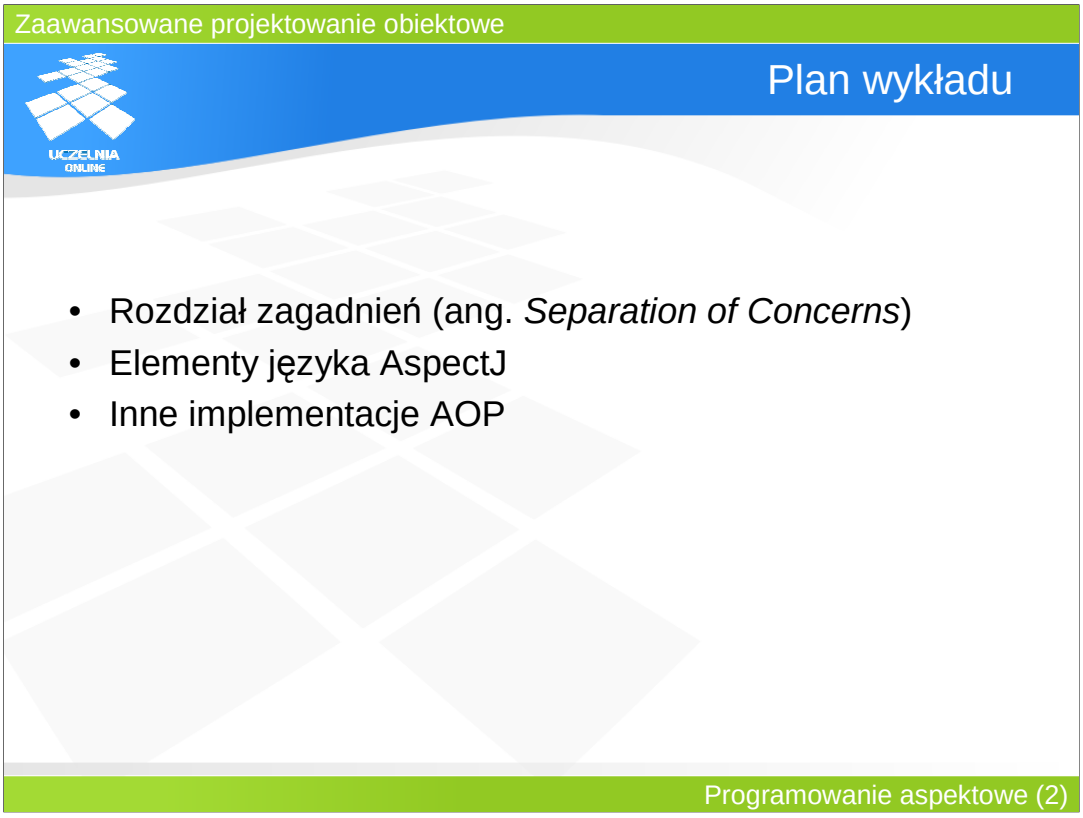

Zaawansowane projektowanie obiektowe
Programowanie aspektowe
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

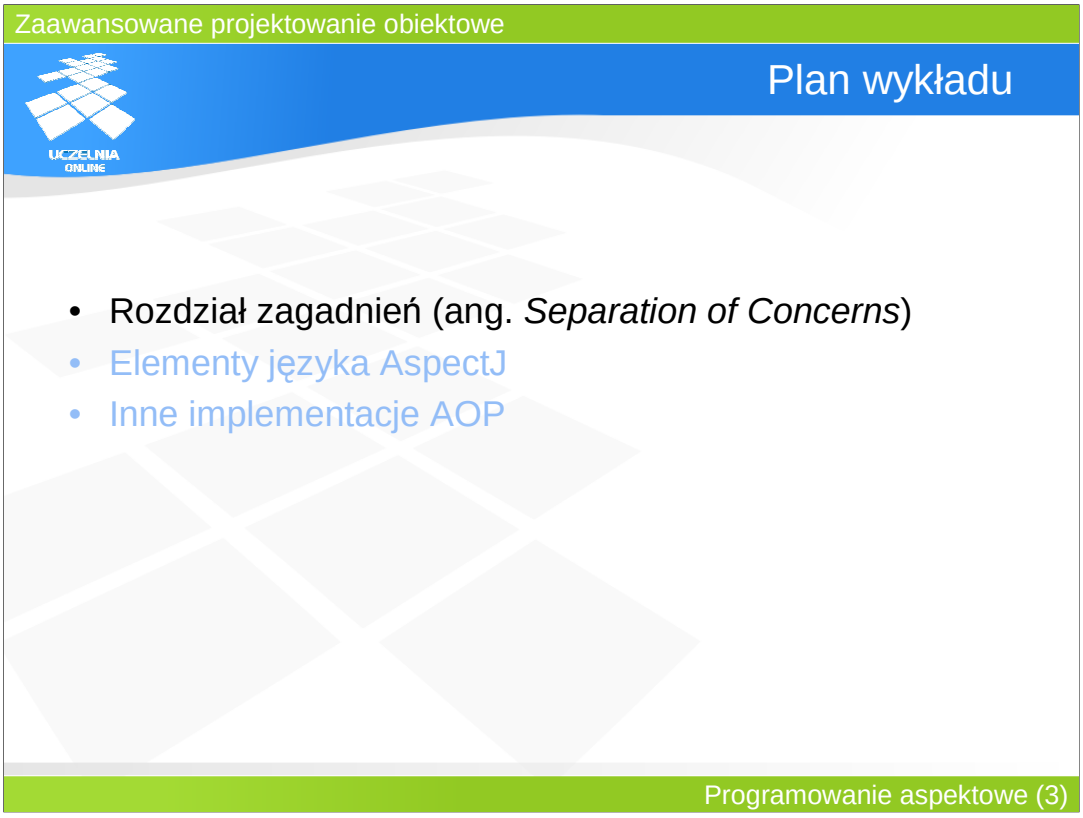


Wykład poświęcony programowaniu aspektowemu stanowi jedynie wprowadzenie do tej tematyki. Jego celem jest wskazanie potrzeby powstania tego mechanizmu oraz opisanie konstrukcji jego podstawowych elementów.

Pierwsza część wykładu obejmuje problem rozdziału zagadnień w oprogramowaniu i jego znaczenie dla pielęgnowalności oprogramowania. Problem ten, mimo zmian paradygmatów programowania, nadal pozostaje nierozwiązany do końca.

W kolejnej, najdłuższej części omówione zostaną podstawowe pojęcia aspektów: punktu złączenia, punktu cięcia oraz porady oraz ich implementacja w języku AspectJ – najpopularniejszej implementacji aspektów dla języka Java.

Ostatnim elementem wykładu będzie krótki przegląd innych rozwiązań aspektowych.



Pierwsza część wykładu jest poświęcona koncepcji programowania aspektowego i jego zaletom w porównaniu do innych paradygmatów, przede wszystkim programowania obiektowego.

Rozdział zagadnień

Zasada rozdziału zagadnień (*Separation of Concerns*)

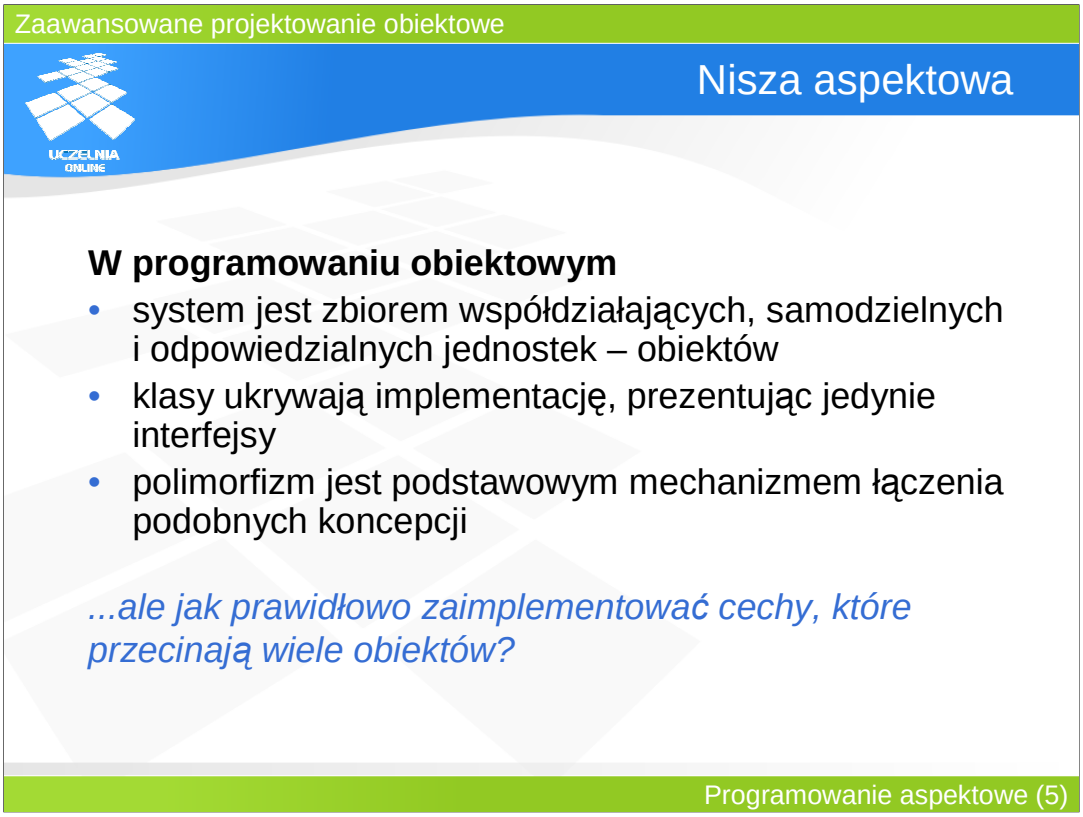
- program powinien być zdekomponowany w taki sposób, aby różne jego aspekty znajdowały się w dobrze odseparowanych od siebie częściach systemu
- każdy aspekt zajmuje się jednym zagadnieniem
- zagadnienie to szczególny cel programu, koncepcja albo funkcja

D. Parnas "On the criteria to be used in decomposing systems..." (1972)
G. Polya "How to Solve It?" (1973)
E. Dijkstra "A Discipline of Programming" (1976)

O tym, że modularyzacja jest podstawą prawidłowej pielęgnacji kodu, pisali najwięksi badacze inżynierii oprogramowania, m.in. D. Parnas i E. Dijkstra. Postulowali oni dekompozycję programu na części, z których każda będzie dotyczyła jednego zagadnienia.

Kolejne paradygmaty programowania, począwszy od programowania funkcyjnego, poprzez strukturalne, aż po obiektowe, próbowały spełnić ten postulat. W kolejnych generacjach języków i metod programowania pojawiały się nowe koncepcje, które dzieliły program według różnych kryteriów. Jednak żadna z dotychczasowych idei nie odpowiadała w pełni temu wymaganiu.

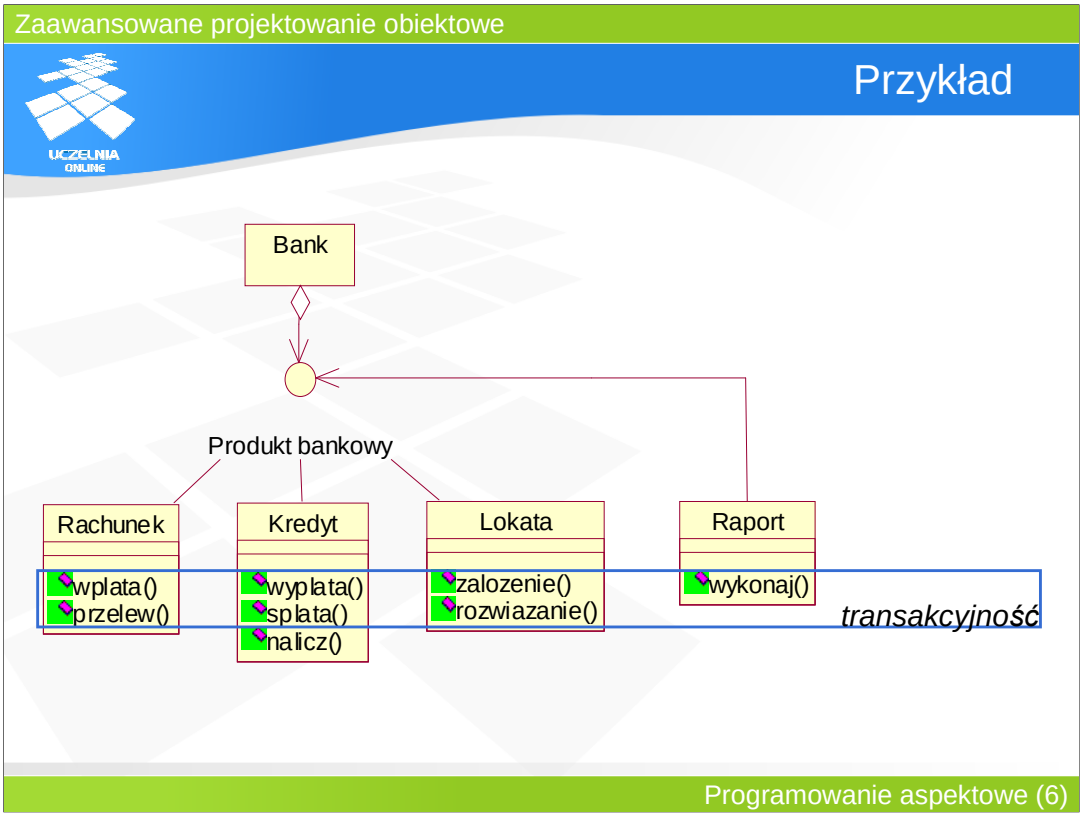
Można z tego wyciągnąć wniosek, że prawidłowa modularyzacja jest jednym ze świętych Graali inżynierii oprogramowania.



Wydaje się, że programowanie obiektowe jest skuteczną odpowiedzią na problem modularyzacji. Podkreślana w tym paradygmacie zasada odpowiedzialności nakazywała podział systemu na obiekty, natomiast mechanizmy dziedziczenia, polimorfizmu oraz składania klas pozwalały w różny sposób łączyć klasy we współpracujące grupy.

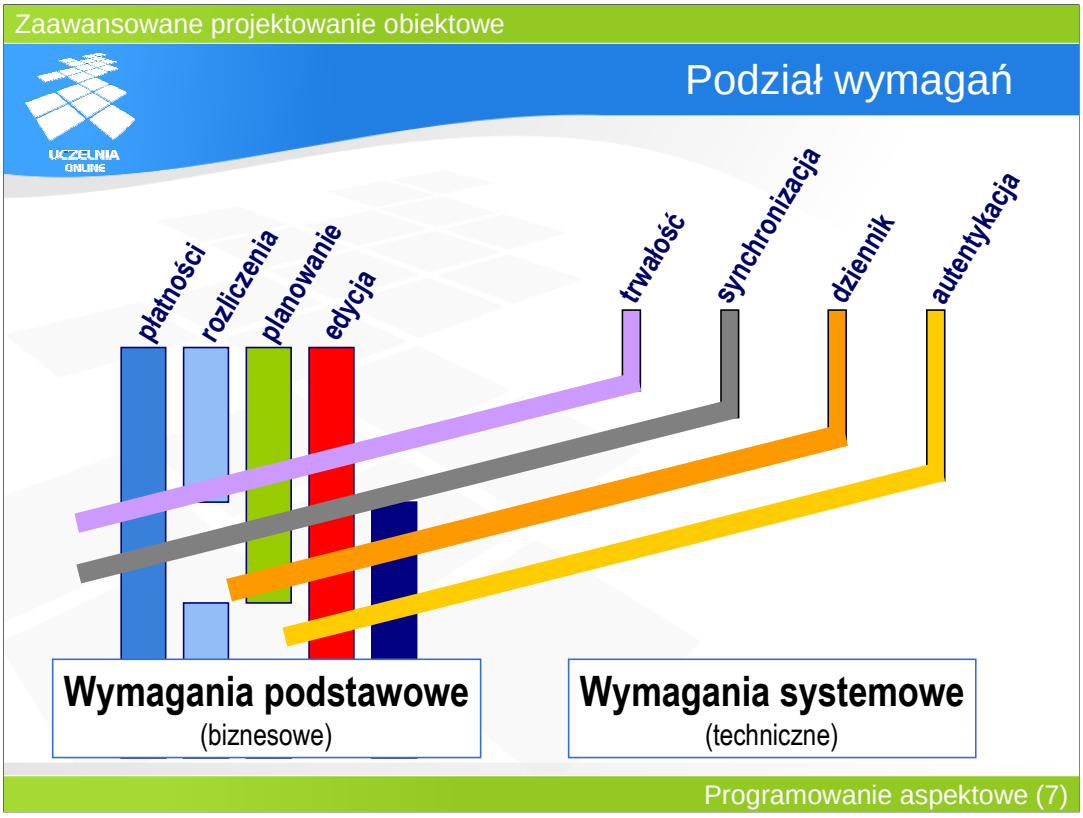
Okazuje się jednak, że istnieją takie modele rzeczywistości, które bardzo trudno podzielić na moduły. W większości przypadków dotyczy to sytuacji, w których przyjęte kryterium podziału (np. funkcjonalne) okazuje się być nie jedynym, ale jednym z wielu. Często przywoływanym przykładem jest problem zapisywania informacji o wydarzeniach w systemie (logowania). Wywołanie metody logującej pojawia się wówczas niemal w całym systemie, bez względu na przeznaczenie poszczególnych jego elementów. Próba modyfikacji kodu logującego wymaga skomplikowanych, pracochłonnych i narażonych na błędy operacji na wielu fragmentach kodu w systemie

Pojawia się zatem zapotrzebowanie na mechanizm, który umożliwi osiągnięcie pełnej modularyzacji systemu niezależnie od liczby kryteriów podziału.



Przykładem takiej sytuacji jest również fragment kodu systemu bankowego. Obiekt Bank przechowuje kolekcję Produktów bankowych: Rachunków, Kredytów i Lokat. Ponadto przechowuje obiekty klasy Raport, które operują na Produktach bankowych.

Łatwo zauważyć, że poszczególne metody tych klas wymagają zapewnienia wspólnych właściwości, które nie są związane z podziałem funkcjonalnym obiektów, np. transakcyjności wywołań metod. Próba zaimplementowania tej cechy przez odwołania w każdej klasie do kodu realizującego ją może prowadzić do bałaganu i obniżenia pielęgnowalności kodu.

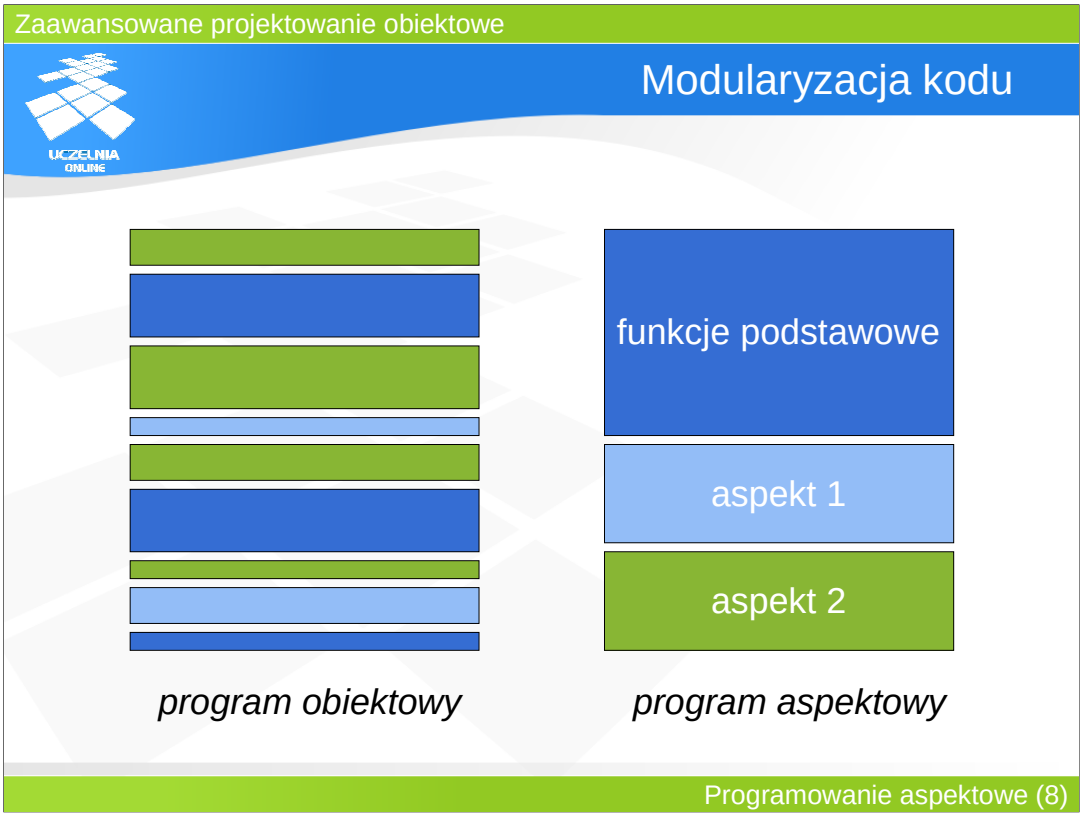


Na podstawie wniosków wyciągniętych z analizy przykładu można zatem podzielić wymagania implementowane w każdym systemie na dwie grupy:

- wymagań podstawowych, które reprezentują główne biznesowe cele, jakie użytkownik chce osiągnąć za pomocą tego systemu. W przypadku systemu bankowego może to być zarządzanie rachunkami bankowymi, realizacja operacji bankowych etc.
- wymagań systemowych, które odpowiadają np. technicznym wymogom związanym z wieloma lub nawet wszystkimi wymaganiami podstawowymi. Przykładem takiego wymagania jest np. trwałość zapisu danych, transakcyjność, logowanie wywołań).

Warto zauważyć, że wymagania systemowe są ortogonalne w stosunku do wymagań podstawowych, tzn. przecinają je i wymagają implementacji w wielu niezwiązanych ze sobą miejscach. Implementacja każdego z wymagań podstawowych w osobnym module spowoduje, że moduły te będą musiały być przecięte wymaganiami systemowymi.

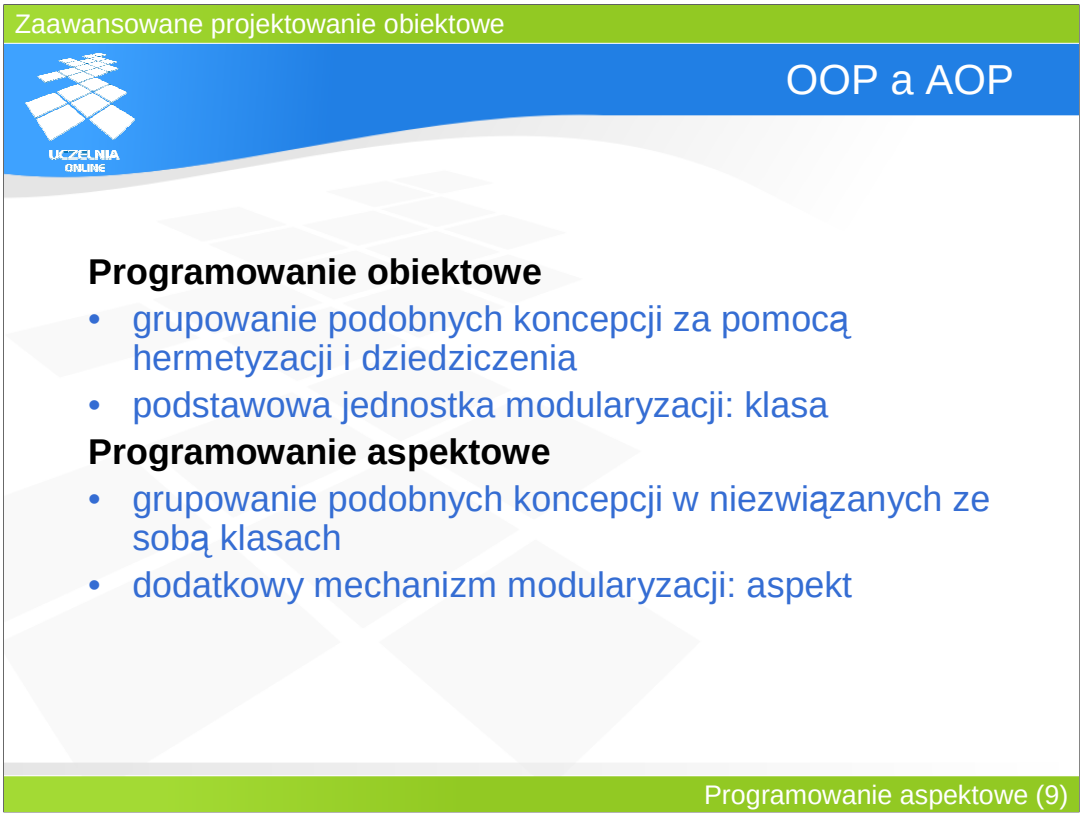
Taka sytuacja jest jednak rzeczą naturalną w projektowaniu oprogramowania. Problem dotyczy zatem nie opisu rzeczywistości, ale sposobu reprezentacji i implementacji poszczególnych zagadnień systemowych, tak aby zachowując modularność umożliwić im bezproblemowe przecinanie innych wymagań.



Program obiektowy, który rozwiązywałby taki problem, najprawdopodobniej wywoływałby w wielu miejscach metody służące do realizacji poszczególnych zagadnień systemowych. W efekcie przypominałby spaghetti-code, w którym pierwotna koncepcja modularyzacji byłaby całkowicie nieczytelna.

Programowanie aspektowe (ang. *Aspect-Oriented Programming*) pozwala rozwiązać ten problem w sposób znacznie bardziej przejrzysty i elegancki. Do typowych mechanizmów obiektowych dodaje ono koncepcję aspektu, grupującego zagadnienia przecinające inne fragmenty kodu. Dzięki temu możliwe jest logiczne wydzielenie ich w postaci osobnych jednostek modularyzacji bez naruszania pierwotnego podziału, co pozwala zachować czytelność kodu systemu, jego hermetyczność etc. Klasy reprezentujące zagadnienia biznesowe nadal zawierają tylko to, za co odpowiadają.

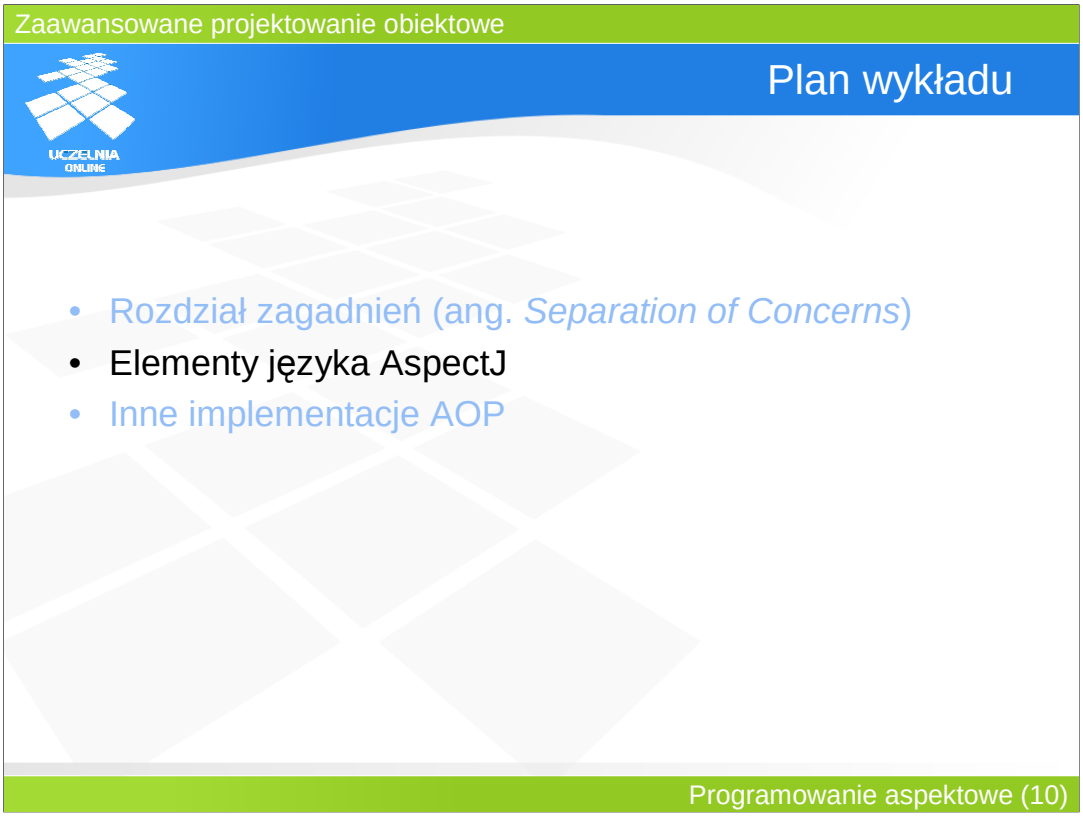
Aspekty, stosowane obok klas, umożliwiają zatem lepszą, wielokryterialną strukturalizację systemu informatycznego niż użycie samych tylko klas.



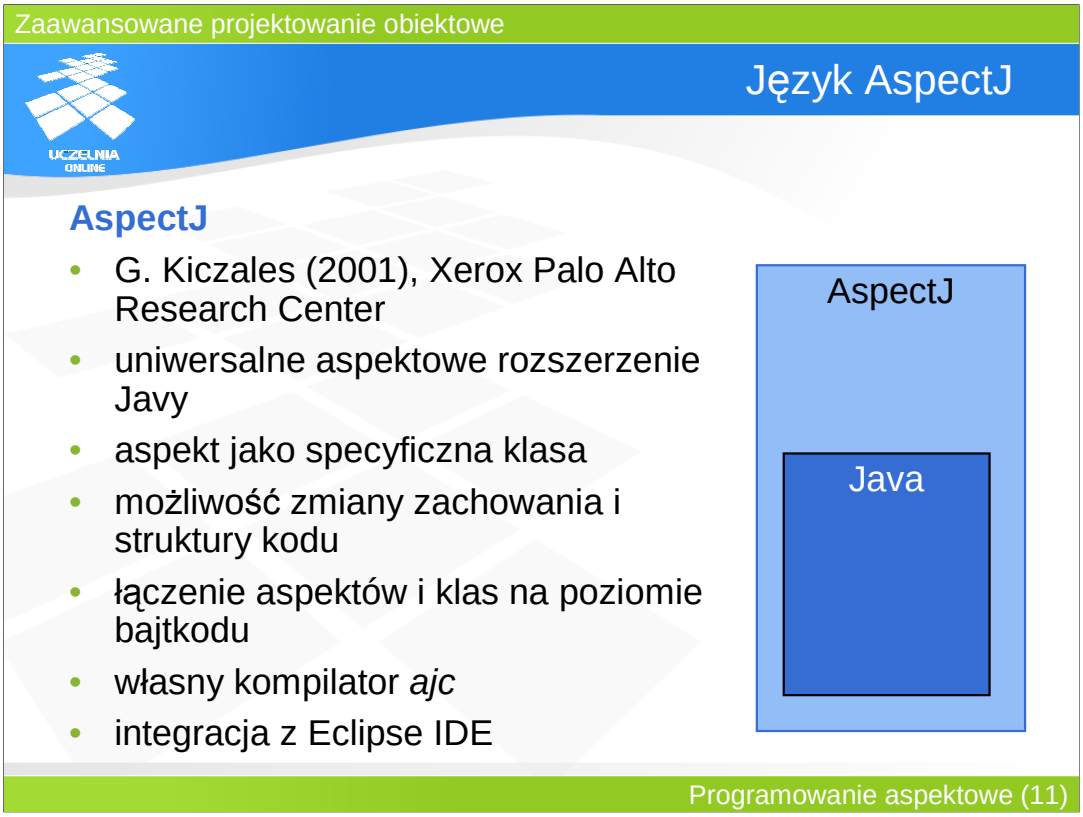
Różnica pomiędzy programowaniem obiektowym a programowaniem aspektowym nie polega na innym celu (w obu przypadkach chodzi o grupowanie podobnych koncepcji i separację różnych), ale na innym doborze narzędzi.

W przypadku programowania obiektowego podstawowymi narzędziami są pojęcie klasy, jej hermetyzacja i dziedziczenie. Zwykle pozwalają one stosować grupowanie koncepcji według jednego kryterium, co w części zastosowań jest wystarczające. Zaletą programowania obiektowego jest jego utrwalona i pewna pozycja na rynku, oraz oparcie w szerokiej gamie popularnych języków programowania.

Programowanie aspektowe zatem nie stoi w sprzeczności z założeniami ani narzędziami programowania obiektowego. Dodatkowym mechanizmem grupowania jest aspekt , reprezentujący pojedyncze zagadnienie (ang. *concern*). Pozwala on na rozszerzenie możliwości grupowania na wiele kryteriów, w tym także przecinających się. Aspekt grupuje kod i reguły jego łączenia z innymi fragmentami programu.



W drugiej części zostaną przedstawione podstawowe pojęcia programowania aspektowego oraz sposób ich zapisu w języku AspectJ. Zaprezentowane będą także możliwości AspectJ oraz przykłady ich zastosowań

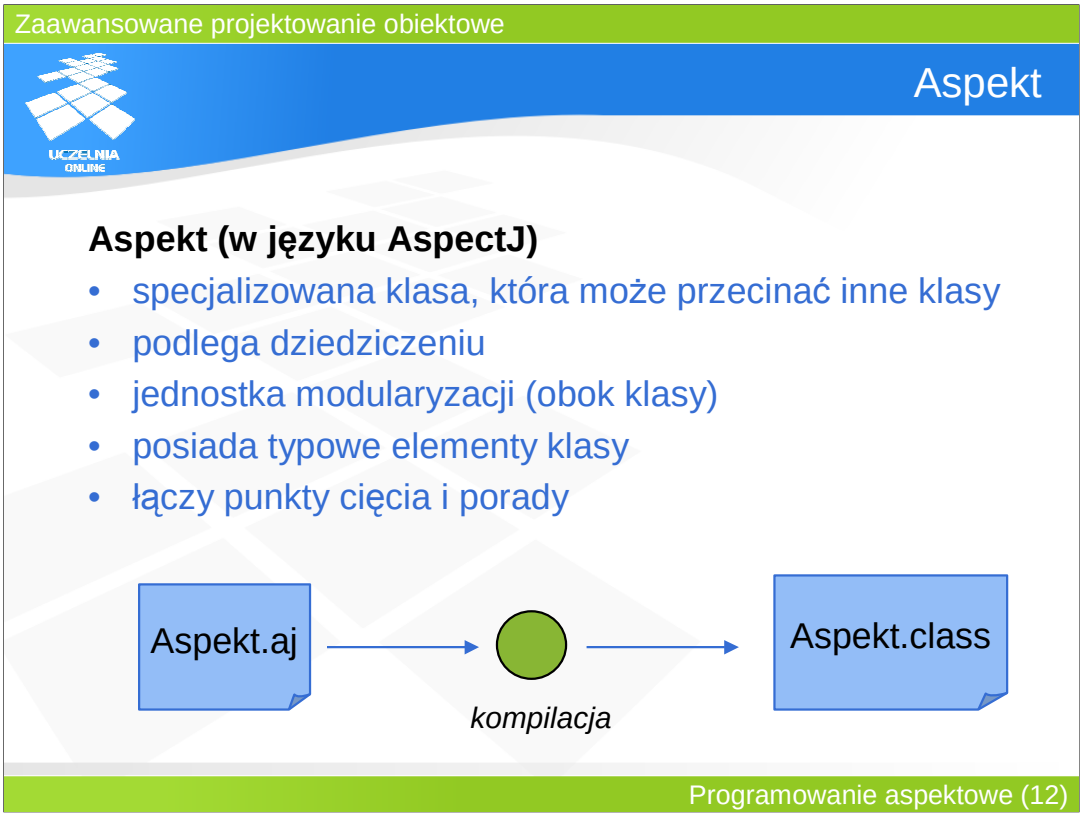


AspectJ jest językiem opisu aspektów zbudowanym w postaci nadbudówki nad językiem Java. Powstał w laboratoriach Xeroxa w zespole kierowanym przez G. Kiczalesa, prowadzącego badania nad technikami modularyzacji kodu. Pierwsza publiczna wersja języka powstała w roku 2001, i tę datę uznaje się za początek jego istrninia.

Stanowi on uniwersalne aspektowe rozszerzenie języka Java, tzn. że każdy program Javy jest jednocześnie poprawnym programem języka AspectJ. Nie modyfikuje on żadnej konstrukcji tego języka, a dodaje nowe – przede wszystkim pojęcie aspektu. Aspekt jest specyficznym rodzajem klasy, który razem z nią pozwala modularyzować program.

AspectJ posiada możliwości przecinania klas zarówno statycznie (modyfikując jego strukturę, hierarchię dziedziczenia), jak i dynamicznie (zmieniając zachowanie programu). Łączenie aspektów i klas odbywa się na poziomie bajtkodu, tzn. że kod wynikowy jest nieprawidłowo podzielony na moduły, jednak kod źródłowy podlega modularyzacji na klasy i aspekty.

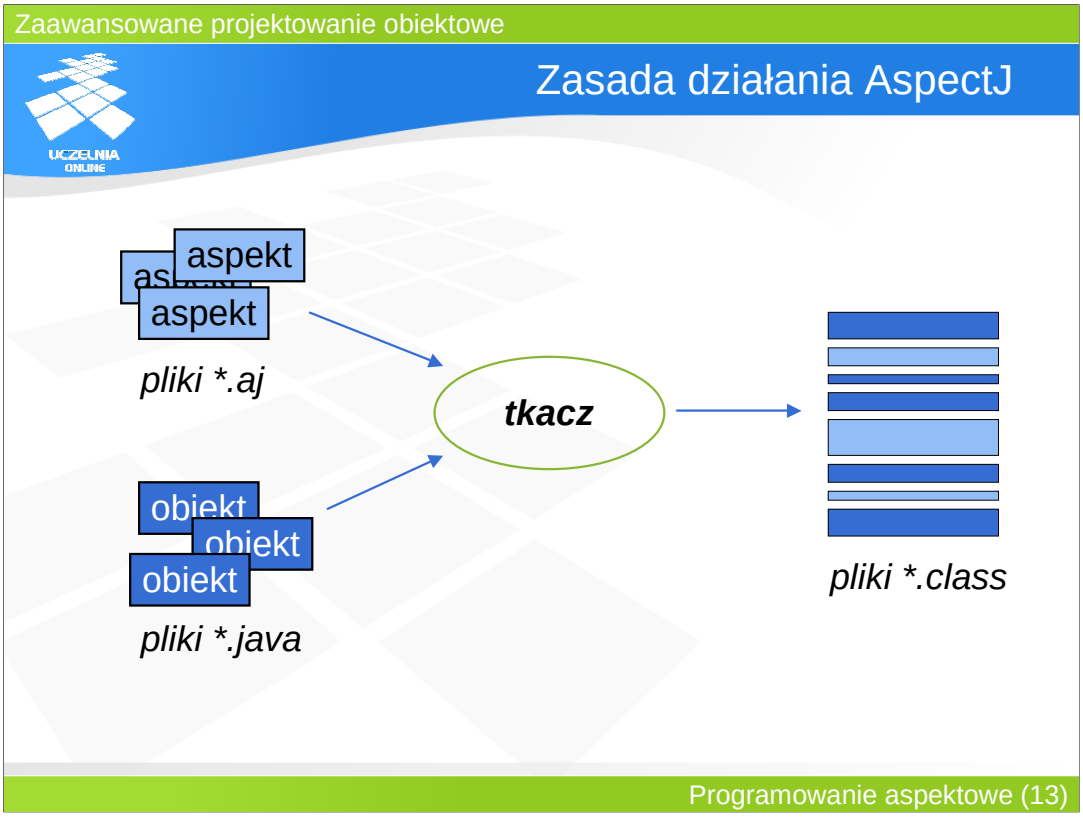
AspectJ jest pełnym językiem programowania: oprócz składni posiada także własne narzędzia – kompilator i debugger. Od czasu przeniesienia go do projektu Eclipse obserwuje się coraz silniejszą integrację języka z tą platformą. Istnieje jednak także niezależna od IDE wersja AspectJ.



Aspekt jest szczególnie wyróżnioną jednostką programu w języku AspectJ, ponieważ występuje w nim jako niezależny moduł, podobny do klasy. W rzeczywistości można traktować jako specjalizowaną klasę, która może przecinać inne klasy, modyfikując ich strukturę i zachowanie. Aspekt w języku AspectJ jest dodatkową, obok klasy jednostką modularyzacji

Deklaracja aspektu, poza typowymi elementami klasy, jak pola i metody (choć nie może on np. zawierać klas wewnętrznych), definiuje *punkty cięcia* – czyli punkty, w których oryginalny kod programu przecina się z aspektem, oraz *porady*, będące fragmentami kodu jaki jest umieszczany i wykonywany w odpowiednim punkcie cięcia.

W postaci źródłowej aspekty są zapisywane w plikach z rozszerzeniem .aj, a po skompilowaniu przyjmują postać bajtkodu i są przechowywane w plikach z rozszerzeniem .class. Przez maszynę wirtualną Javy są traktowane jako zwyczajne klasy.

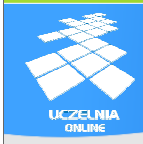


Niepożądany efekt, o którym była mowa wcześniej – czyli przeplatanie się kodu – jest nie do uniknięcia. Celem jest jednak nie całkowite wyeliminowanie go, ale przesunięcie z kodu źródłowego do kodu wynikowego lub nawet wykonania programu. Proces przeplatania nazywany jest tkaniem (ang. *weaving*), a jego wykonawcą jest tkacz (ang. *weaver*).

Niezależnie od zastosowanej techniki tkania, istota tego procesu polega na identyfikacji określonych punktów programu (punktów złączeń), w których wykonywany jest inny kod, niż wskazuje na to oryginalna wersja programu, lub modyfikowana jest struktura programu. Typowym przykładem identyfikowanego punktu złączenia jest np. wywołanie metody.

W przypadku AspectJ elementami podlegającymi tkaniu są klasy Javy oraz kod i reguły tkania zawarte w aspektach. Na podstawie tych elementów tkacz generuje bajtkod, który jest poprzeplatanym kodem wynikowym, tak jakby podstawą do jego wygenerowania był poprzeplatany kod źródłowy. Zatem rolę tkacza odgrywa w tym przypadku specjalny kompilator, który produkuje kod wynikowy będący prawidłowym programem w języku Java.

Inne metody tkania obejmują przeplatanie kodu w trakcie ładowania kodu lub jego wykonywania. Można wyobrazić sobie np. wirtualną maszynę języka Java wyposażoną w możliwość łączenia aspektów z kodem.



Przykład prostego aspektu

```
public class HelloWorld {
   public static void hej(String tekst) {
      System.out.println(tekst);
   }
   public static void hejKolego(String tekst, String imie) {
      System.out.println(imie + ", " + tekst);
   }
   public static int hejLudzie(String tekst, String[] imiona) {
      for (int i = 0; i < imiona.length; i++)
         System.out.println(imiona[i] + ", " + tekst);
      return imiona.length;
   }
}
```

na podstawie przykładu R. Laddada (2002)

Ponieważ opis języka najlepiej rozpocząć od przykładu, warto przyjrzeć się powyższej klasie. Składa się ona z trzech metod, wyświetlających teksty do wskazanych osób: *hej()*, *hejKolego()* oraz *hejLudzie()*.

Należy zmodyfikować zachowanie tego programu tak, aby wywołanie jakiejkolwiek metody o nazwie zaczynającej się od *hej* powodowałoby na początku uprzejme przywitanie się, a na końcu – podziękowanie.

Przykład prostego aspektu

```
public aspect Maniery {
    pointcut wywolanieHej () :
        call(public static void HelloWorld.hej*(..));

    before() : wywolanieHej() {
        System.out.println("Dzień dobry!");
    }

    after() : wywolanieHej() {
        System.out.println("Dziękuję!");
    }
}
```

na podstawie przykładu R. Laddada (2002)

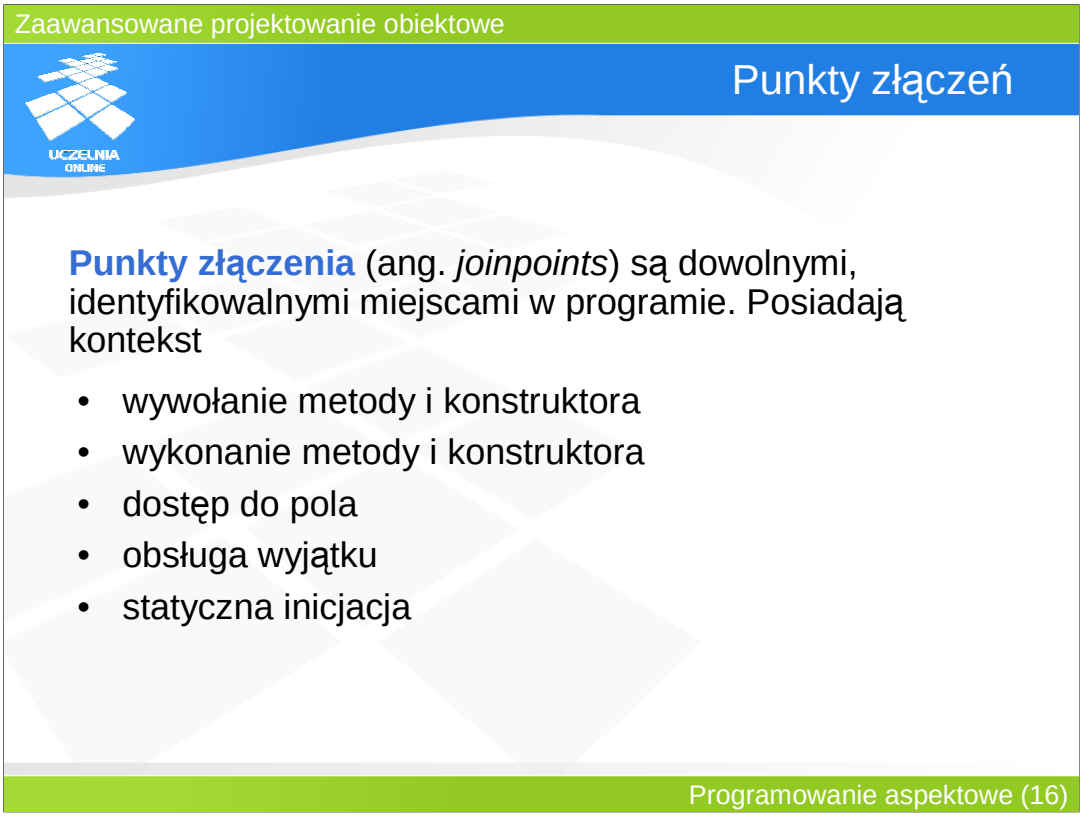
W celu realizacji tego zadania zaimplementowano aspekt Maniery. Składa się on z fragmentu kodu zwanego punktem cięcia oraz dwóch porad.

Punkt cięcia określa, że momentem interesującym dla tego aspektu jest wywołanie dowolnej metody zaczynającej się od słowa *hej* oraz umieszczonej w klasie HelloWorld.

Punkt ten jest przydatny przy realizacji dwóch porad: *before()* i *after(),* wykonywanych odpowiednio przed i po osiągnięciu punktu złączenia.

Porada pierwsza mówi, że bezpośrednio przed osiągnięciem punktu złączenia określonego jako *wywołanieHej()*, należy wyświetlić napis "Dzień dobry!". Druga porada dotyczy postępowania bezpośrednio po osiągnięciu punktu złączenia: wyświetlony zostanie napis "Dziękuję".

W efekcie wywołanie dowolnej metody o nazwie zaczynającej się od *hej* będzie poprzedzone przywitaniem, a zakończone podziękowaniem.



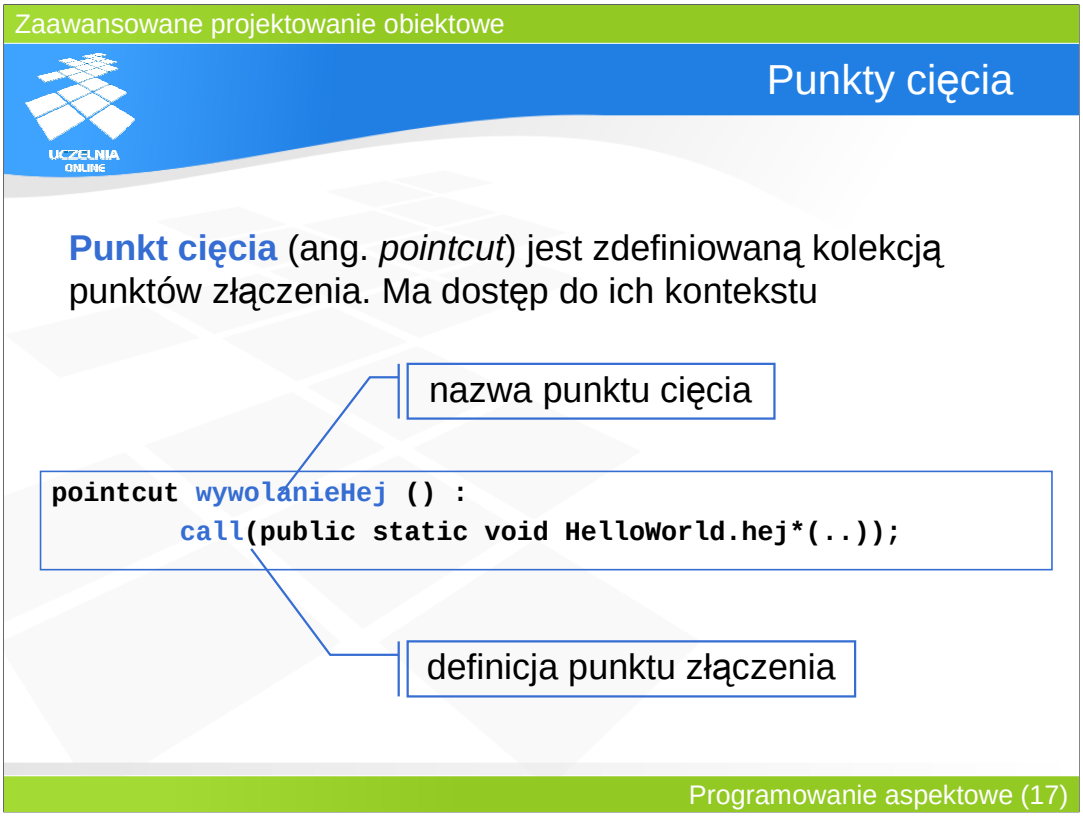
Pierwsze, bardzo ważne pojęcie każdego języka i systemu programowania aspektowego, to punkt złączenia. Może nim być dowolny identyfikowalny punkt programu. W praktyce typowymi punktami złączenia (niezależnie od możliwości języka) są:

•wywołanie i wykonanie dowolnej metody i dowolnego konstruktora

•odwołanie do pola w klasie (do odczytu i do zapisu)

•statyczna inicjacja klasy

•obsługa wyjątku



Punkt cięcia jest konsekwencją istnienia punktów złączenia: stanowi on ich zdefiniowaną kolekcję, określoną za pomocą pewnej deklaracji. Punkty cięcia mogą mieć dostęp do kontekstu, w jakim nastąpiło złączenie, i wykorzystać go w swojej definicji.

Podany przykład nazwanego punktu cięcia *wywolanieHej()* składa się ze słowa kluczowego *pointcut*, definiującego punkt złączenia, nazwy i parametrów, jakie są wówczas przekazywane, oraz definicji punktu złączenia. W tym przypadku punktem złączenia jest wywołanie metody o nazwie zaczynającej się od słowa *hej* znajdującej się w klasie HelloWorld. Każdy element, który w punkcie cięcia został opisany szczegółowo, można opisać także ogólnie, stosując znaki * (dopasowanie nazwy do prostego wyrażenia regularnego) oraz + (oznaczający domknięcie przechodnie relacji dziedziczenia).

Rodzaje punktów cięcia

Punkty cięcia dzielą się według złożoności na
- proste, definiujące jeden punkt złączenia
- złożone, zbudowane z kolekcji punktów złączenia powiązanych operatorami logicznymi (!, || i &&)

Punkty cięcia dzielą się według tożsamości na
- nazwane, posiadające nazwę, za pomocą której odwołuje się do nich porada
- anonimowe, zapisane wprost w warunku porady

Punkty cięcia można podzielić według dwóch podstawowych kryteriów: złożoności oraz tożsamości.

W przypadku pierwszego kryterium punkty złączeń dzielą się one na proste punkty cięcia, zbudowane z jednej klauzuli dopasowującej, oraz punkty złożone, opisane negacją, alternatywą lub koniunkcją prostych punktów cięć.

Ponadto punkty cięcia – podobnie jak klasy w Javie – mogą posiadać nazwy, ale nie muszą. Nazwane punkty cięcia mogą być wielokrotnie używane w różnych miejscach aplikacji, natomiast punkty anonimowe są tworzone w miejscu ich użycia i nie można odwołać się do nich z innej części systemu.

Wywołanie metody i konstruktora

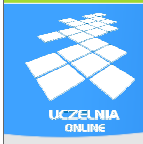
Punkt cięcia *call* reprezentuje wywołania metod i konstruktorów (po ewaluacji parametrów, przed wykonaniem)

```
call (sygnatura metody lub konstruktora)
```

call(* Klasa.metoda*(int,..))	wywołanie metody o nazwie zaczynającej się od *metoda*, której pierwszy argument jest typu String
call(* *.metoda(..))	wywołanie metody o nazwie *metoda* w dowolnej klasie
call(Klasa+.new(..))	wywołanie dowolnego konstruktora w klasie Klasa i jej podklasach

Najpopularniejszym rodzajem punktu cięcia jest wywołanie metody. Poprzez wywołanie należy rozumieć moment po wydaniu polecenia wykonania tej metody i ewaluacji argumentów, ale przed fizycznym rozpoczęciem wykonywania kodu.

Punkt jest opisywany klauzulą *call*, której parametrem jest sygnatura metody lub konstruktora. W przypadku konstruktora nazwą metody jest słowo *new*. Podobnie jak w innych przypadkach, do określenia metody można użyć znaków specjalnych * i +. Lista typów parametrów metody także może być niepełna: dowolną liczbę nieznanych typów można zastąpić dwiema kropkami (..).



Wykonywanie metody i konstruktora

Punkt cięcia *execution* reprezentuje wykonanie ciała metody lub konstruktora

```
execution (sygnatura metody lub konstruktora)
```

execution(* * Klasa.metoda(..))	wykonanie metody *metoda* o dowolnych parametrach w klasie *Klasa*
execution(public * pl.elearning..*.*(..))	wykonanie dowolnej publicznej metodzie w dowolnej klasie w dowolnym pakiecie poniżej *pl.elearning*
execution(* * pl.elearning..*.new(..))	wykonanie dowolnego konstruktora w dowolnej klasie w pakiecie *pl.elearning*

Nieco innym typem punktem cięcia jest wykonanie metody lub konstruktora. Punkt ten jest osiągany w momencie fizycznego rozpoczęcia wykonania kodu metody lub konstruktora, zatem następuje później niż punkt cięcia *call*.

Parametrem, podobnie jak w poprzednim przypadku, jest sygnatura metody lub konstruktora. Zasady jej opisu są identyczne jak w przypadku wywołania metody.

Dostęp do pola

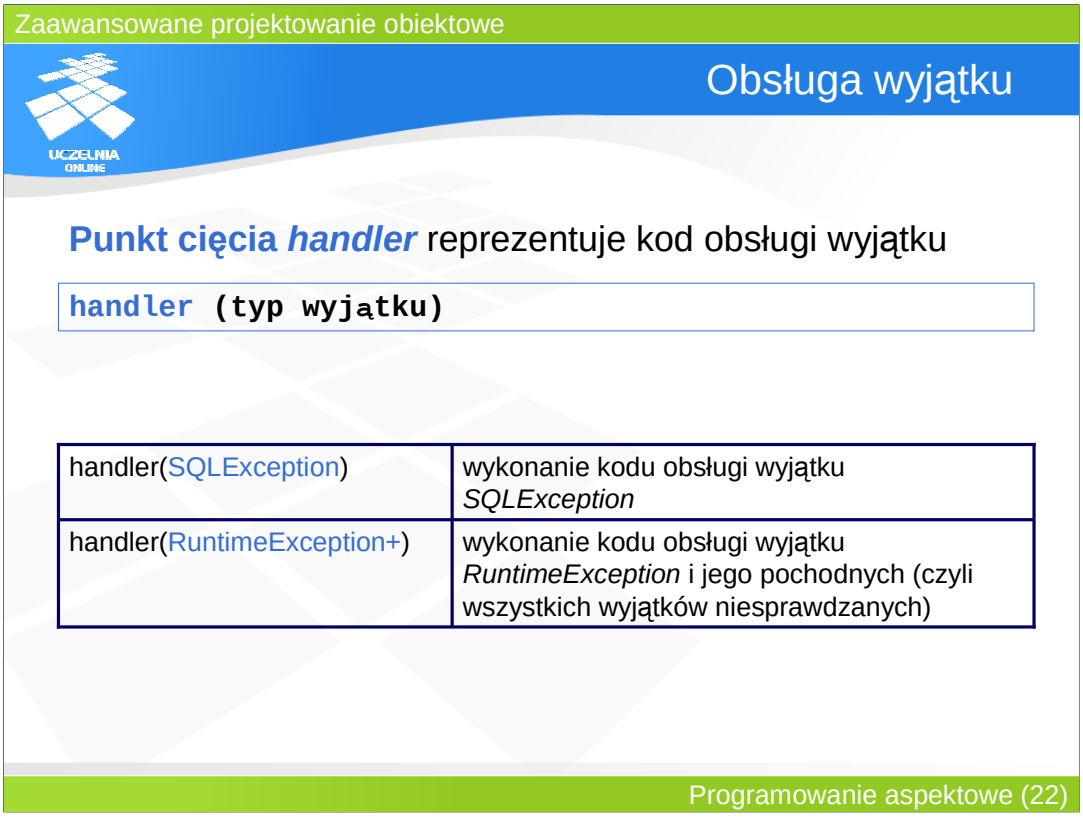
Punkty cięcia *set* i *get* reprezentują odwołania do pól klasy i obiektu

```
set (deklaracja pola)
get (deklaracja pola)
```

set(InnaKlasa Klasa.pole)	nadanie wartości polu *pole* typu *InnaKlasa* w klasie *Klasa*
get(InnaKlasa *.pole)	odczyt wartości pola *pole* typu *InnaKlasa* w dowolnej klasie

Punkty cięcia *get* i *set* reprezentują odwołania do pól obiektu. Punkt *set* jest osiągany w momencie przypisania wartości do pola, natomiast punkt *get* – jej odczytu.

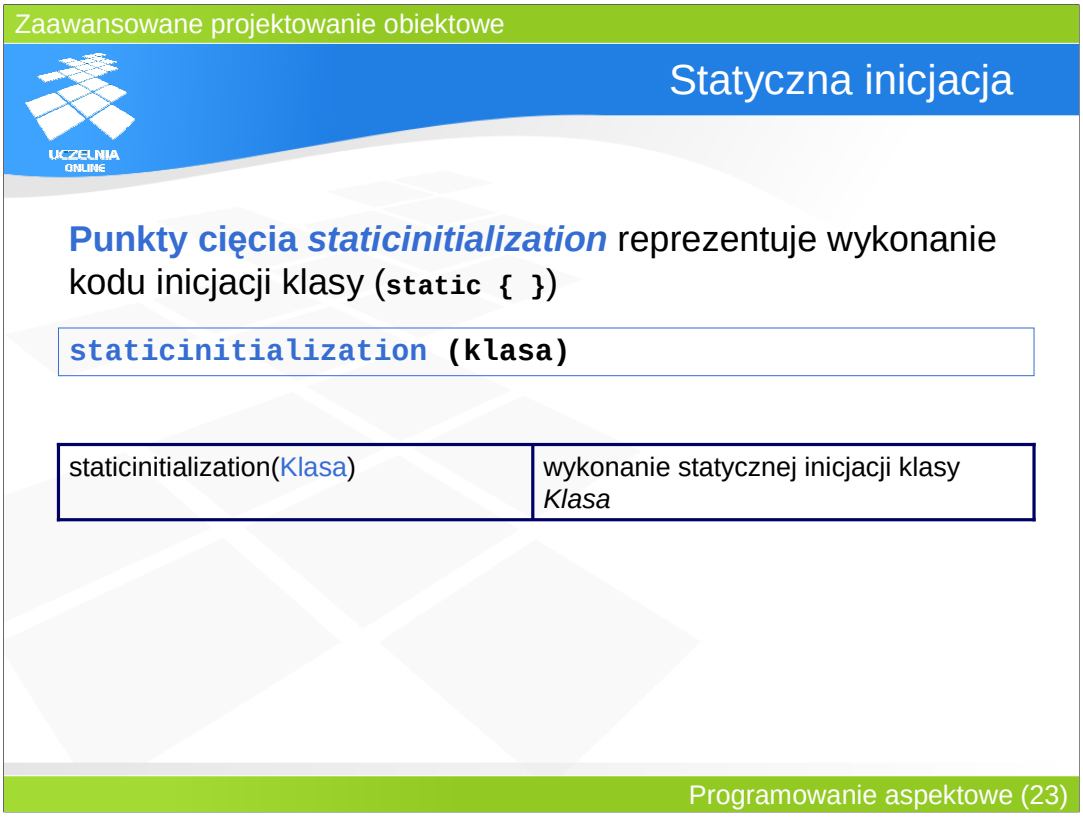
Parametrem tych punktów cięcia jest deklaracja pola, którego punkt dotyczy. Nazwa pola, jego klasy, typ i zasięg widoczności mogą być reprezentowane w postaci uproszczonych wyrażeń regularnych zawierających * oraz ..



Punkt cięcia *handler* jest osiągany przez program w momencie rozpoczęcia obsługi wyjątku danego typu.

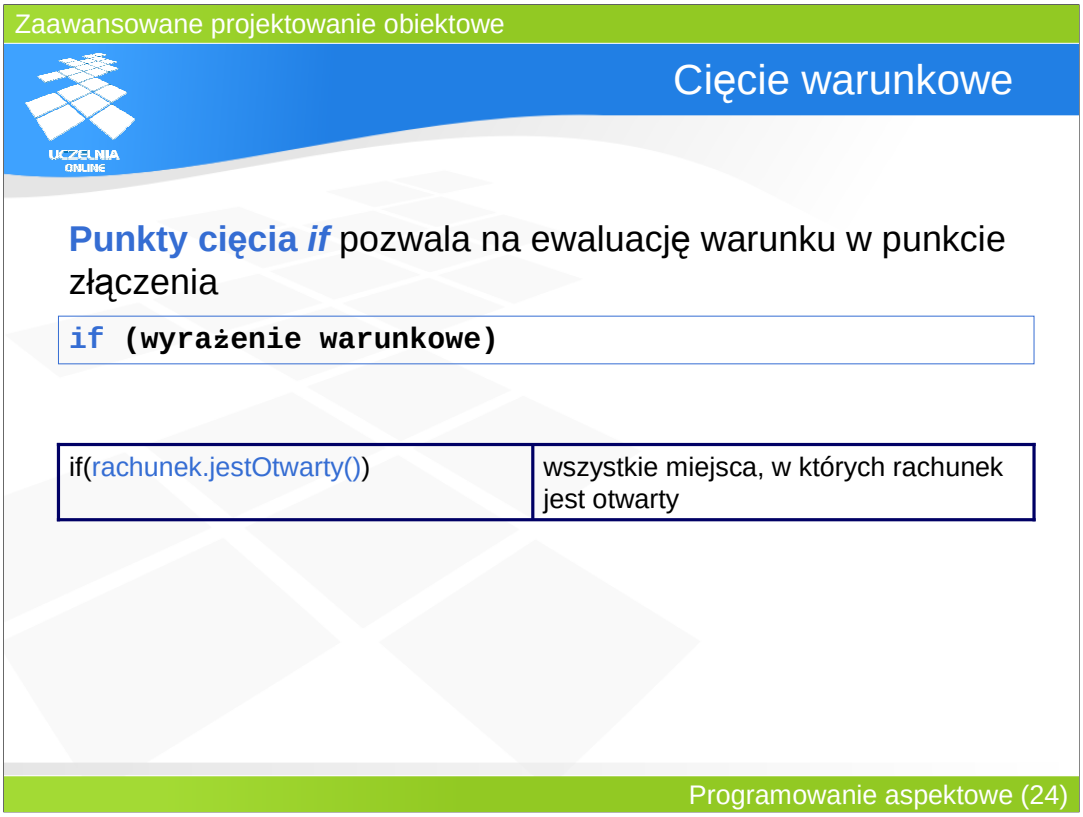
Parametrem tego punktu jest typ wyjątku, który również może być zapisany w sposób uogólniony. Na przykład zapis *handler(RuntimeException+)* oznacza obsługę wyjątku RuntimeException i wszystkich jego typów potomnych, a więc wyjątków niesprawdzanych (ang. *unchecked*).

W celu przechwycenia wartości wyjątku (czyli reprezentującego go obiektu) należy użyć punktu cięcia *args*.



Blok statycznej inicjacji klasy (o postaci static {...}) jest reprezentowany przez punkt cięcia *staticinitialization*.

Parametrem tego punktu jest klasa, w której blok się znajduje.



Szczególnym rodzajem punktu cięcia jest *if*, który pozwala na ewaluację warunku logicznego. Punkt ten jest osiągany, gdy warunek staje się prawdziwy.

Wyrażenie warunkowe użyte w deklaracji punktu cięcia może odwoływać się tylko do składowych statycznych, parametrów punktu cięcia lub porady oraz zmiennych *thisJoinPoint*. Nie może wywoływać metod niestatycznych aspektów lub korzystać z wartości lub wyjątków zgłaszanych przez poradę *after*. Należy także unikać efektów wynikających z ewaluacji warunku, ponieważ kolejność ewaluacji nie jest zdefiniowana w języku i jej nieoczekiwana zmiana może wywoływać nieprzewidziane efekty.

Leksykalne położenie

Punkt cięcia *within* reprezentuje punkty złączeń znajdujące się wewnątrz klasy (i jej klas wewnętrznych)

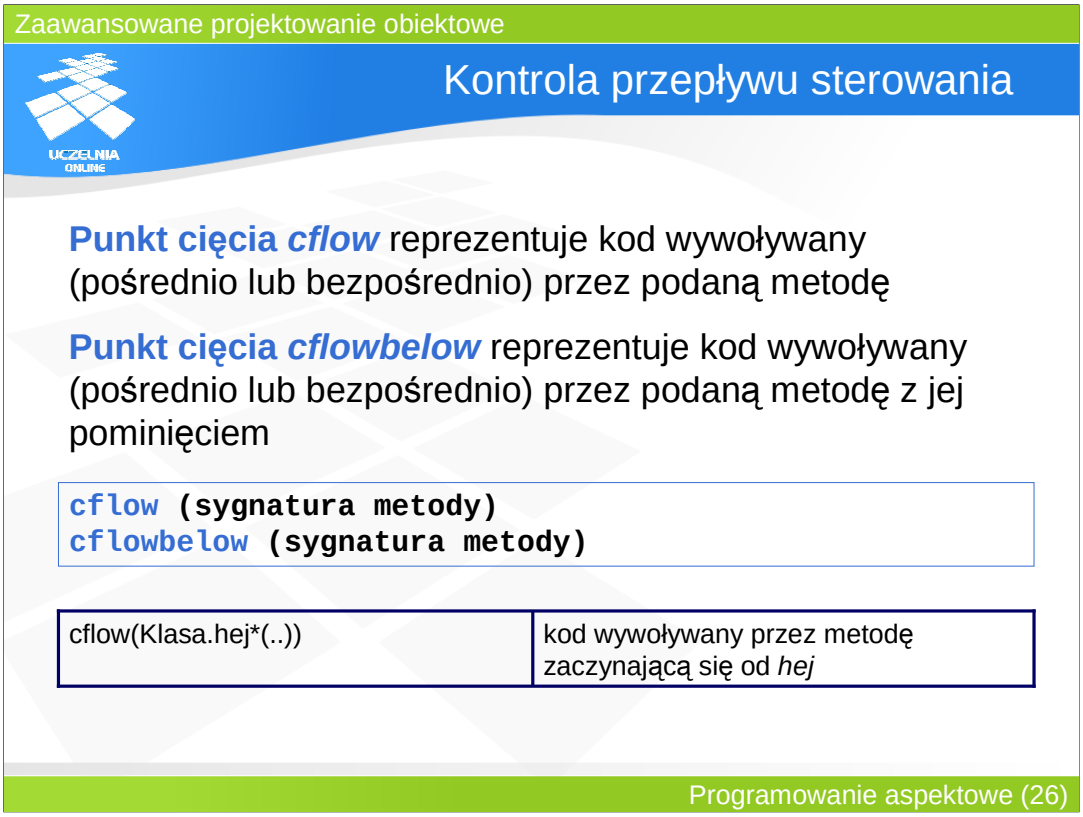
Punkt cięcia *withincode* reprezentuje punkty złączeń znajdujące się wewnątrz metody lub konstruktora podanej klasy

```
within (klasa)
withincode (sygnatura metody)
```

within(Klasa)	wszystkie miejsca wewnątrz klasy *Klasa*
withincode(Klasa.hej*(..))	wszystkie miejsca wewnątrz metod zaczynających się od *hej* w klasie *Klasa*

Dwa punkty cięcia *within* i *withincode* są osiągane w momencie, gdy sterowanie znajdzie się odpowiednio wewnątrz podanej klasy lub metody. Ponieważ są to punkty statyczne, dlatego ich osiągnięcie jest ewaluowane w trakcie kompilacji.

Parametrem punktu *within* jest klasa, natomiast punktu *withincode* – sygnatura metody, które są analizowane.



Za pomocą AspectJ można kontrolować także bardziej dynamiczne aspekty zachowania programu. Punkt cięcia *cflow* reprezentuje kod znajdujący się w obszarze sterowania wywoływanym przez podaną metodę, włączają ją samą. Podobny punkt cięcia *cflowbelow* jest jest poddrzewem *cflow*, jednak nie zawiera on samej metody, którą określono w sygnaturze punktu.

W obu przypadkach stan kontekstu można odczytać wykorzystując punkty cięcia *this*, *target* i *args*.

Należy pamiętać, że tych punktów cięcia nie można negować, czyli używać w konstrukcjach typu *! cflow(Klasa.metoda())*

Cel wywołania i argumenty

Punkt cięcia *this* reprezentuje odwołanie do własnego obiektu
Punkt cięcia *target* reprezentuje odwołanie do wybranego obiektu

```
this (klasa lub obiekt)
target (klasa lub obiekt)
```

Punkt cięcia *args* reprezentuje wywołanie metody o podanych typach argumentów

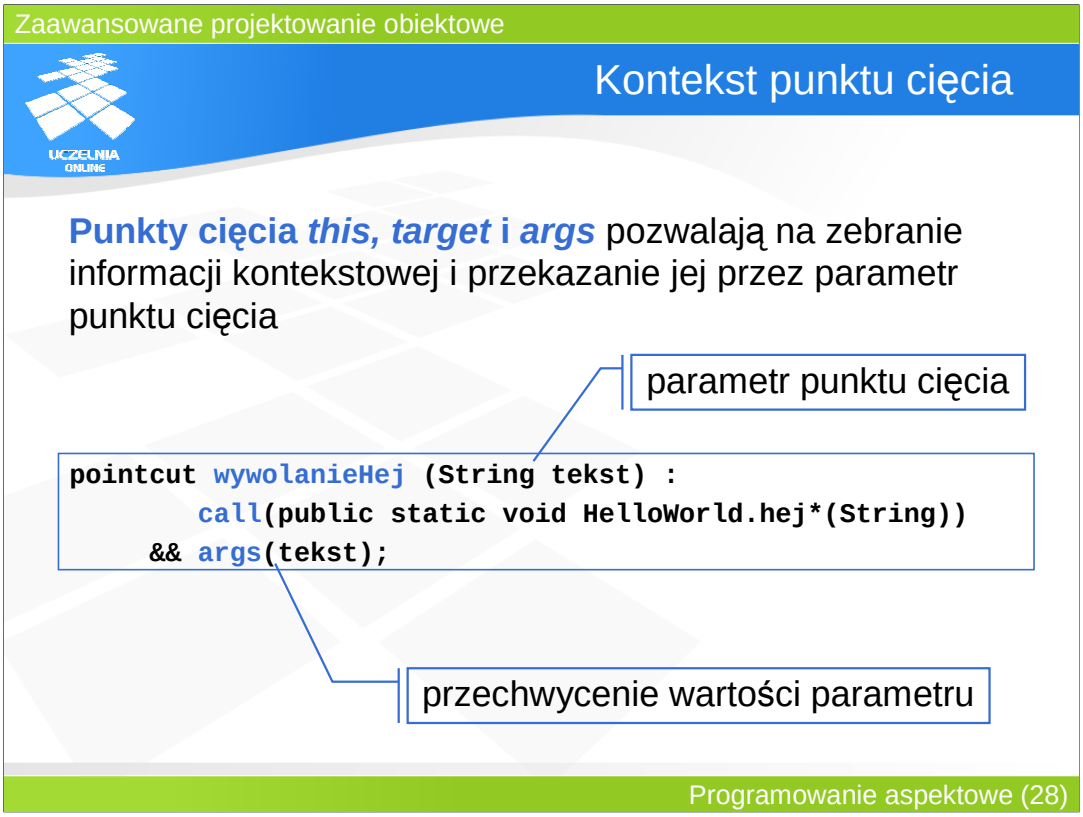
```
args (lista typów argumentów)
```


Trzy punkty cięcia: *this*, *target* i *args* służą do dynamicznej ewaluacji stanu program.

Punkt *this* jest osiągany w momencie, gdy bieżący obiekt jest instancją danej klasy.

Punkt *target* opisuje sytuacje, w których przedmiotem przetwarzania jest obiekt określonej klasy, to znaczy przekazywane jego do niego sterowanie. Zatem w punkcie wywołania metody (*call*) obiektem tym nie będzie bieżący obiekt, natomiast w punkcie wykonania metody (*execution*) będzie to ten sam obiekt.

Punkt *args* służy do stwierdzenia, czy parametry wywołania są określonych typów (dotyczy to zarówno wywołania metody, jak i np. obsługi wyjątku). Może on przechowywać listy typów lub zmiennych związanych wewnątrz punktu cięcia (a więc posiadających wynikający z tego typ).



Punkty cięcia *this, target i args* nie służą jednak wyłącznie do zawężania zakresu punktu cięcia. Ich podstawowym zadaniem jest zbieranie informacji o stanie kontekstu i przekazywanie jej do sygnatury punktu cięcia lub porady. Punkty *this* i *target* mogą służyć zapamiętania w zmiennej referencji do obiektu bieżącego lub obiektu, na którym wykonywana jest bieżąca operacja. Natomiast punkt cięcia *args* pozwala przechwycić i zapamiętać oryginalne wartości parametrów w bieżącym kontekście (zwykle związanym z wywołaniem metody).

Zmienne *thisJoinPoint* i *thisStaticJoinPoint*

Zmienna *thisJoinPoint* przechowuje statyczną i dynamiczną informację o miejscu osiągnięcia punktu złączenia

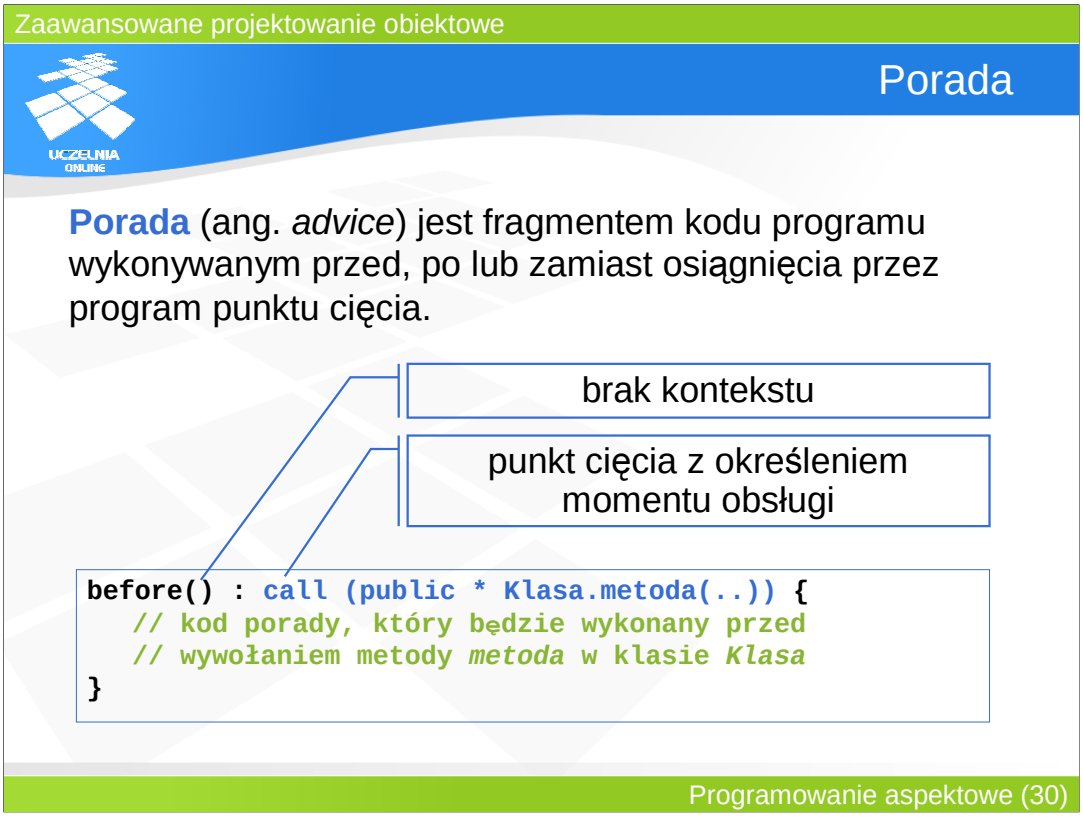
Zmienna *thisStaticJoinPoint* jest ograniczeniem *thisJoinPoint* do informacji statycznej

```
Signature thisJoinPoint.getSignature()
Object thisJoinPoint.getExecutingObject()
Class thisJoinPoint.getExecutingClass()
Object[] thisJoinPoint.getParameters()
SourceLocation thisJoinPoint.
   getCorrespondingSourceLocation()
```


Szczególną rolę w dostępie do kontekstowej informacji o punkcie cięcia pełnią zmienne z rodziny *thisJoinPoint*. Zmienna, od której wzięła nazwę ta rodzina, pozwala na odczyt m.in. sygnatury wywołanej metody, klasy i obiektu wykonującego obecnie kod, listy i wartości parametrów, a nawet danych dotyczących fizycznego położenia kodu źródłowego w pliku.

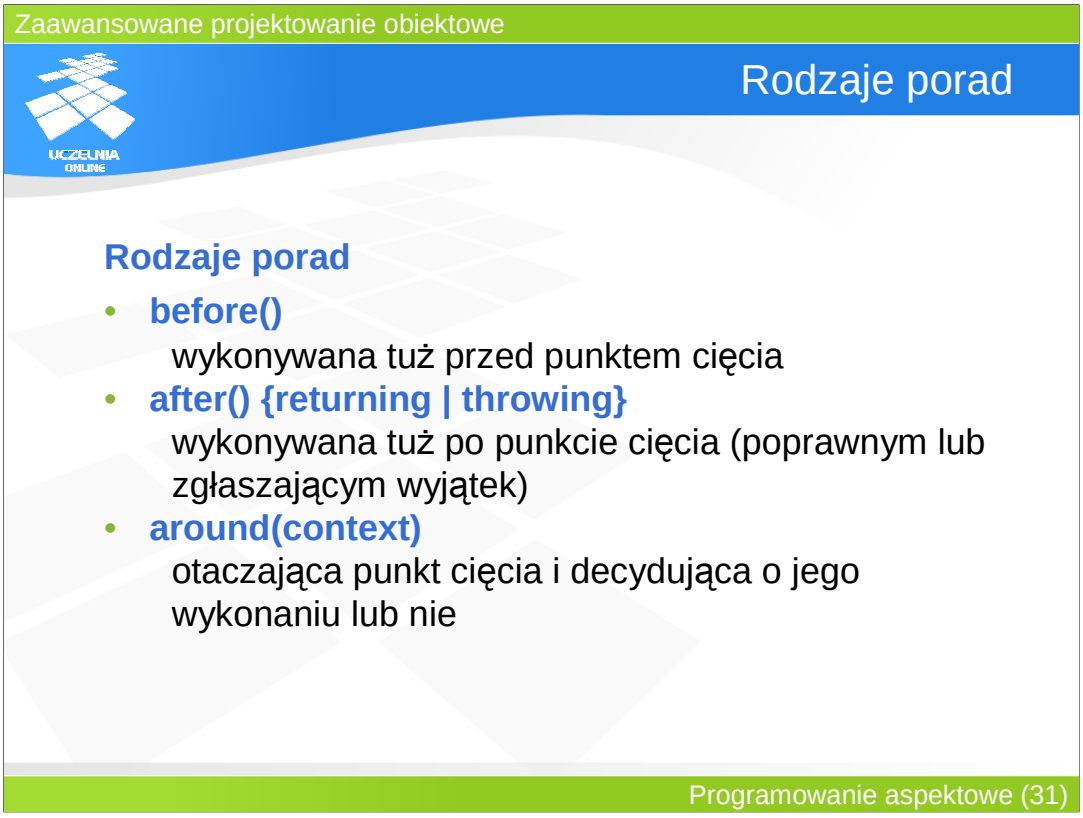
Zmienna *thisJoinPoint* posiada wiele odmian i podtypów, m.in. ograniczoną do informacji statycznej zmiennę *thisStaticJoinPoint.*

Ponadto możliwości tej zmiennej w dużej mierze zależą od kontekstu: może ona np. odczytywać typ przechwyconego wyjątku w punkcie cięcia *handler*.



Porada jest funkcjonalnym dopełnieniem punktu cięcia – określa akcję, jaką należy wykonać w momencie osiągnięcia punktu cięcia.

Definicja porady w AspectJ składa się z określenia momentu uruchomienia porady, definicji punktu cięcia oraz kodu.



Ze względu na moment połączenia porady z kodem w punkcie cięcia można podzielić je na trzy rodzaje*: before(), after()* i *around()*.

•Porada *before()* jest wykonywana tuż przed wystąpieniem punktu cięcia. Ten typ porady może np. służyć do weryfikacji parametrów metody, zmiany sposobu sterowania etc.

•Porada *after()* jest wykonywana tuż po wystąpieniu punktu cięcia. Ponieważ jednak jest to pojęcie zbyt ogólne, można wyróżnić dwie kolejne sytuacje: taką, w której wykonanie oryginalnego kodu w punkcie cięcia zakończyło się poprawnie (sytuacja taka opisywana jest klauzulą *after() returning*) oraz w której efektem wykonania kodu było zgłoszenie wyjątku (opisywane klauzulą *after() throwing*)

•Ostatnim rodzajem porady jest *around().* W odróżnieniu od dwóch poprzednich rodzajów, w tym przypadku istnieje możliwość zastąpienia oryginalnego kodu związanego z punktem cięcia treścią porady. Pozwala to m.in. na zmianę zachowania metod programu poprzez jego całkowitą zmianę lub rozszerzenie go. Ta ostatnia możliwość jest realizowana za pomocą polecenia *proceed()*.

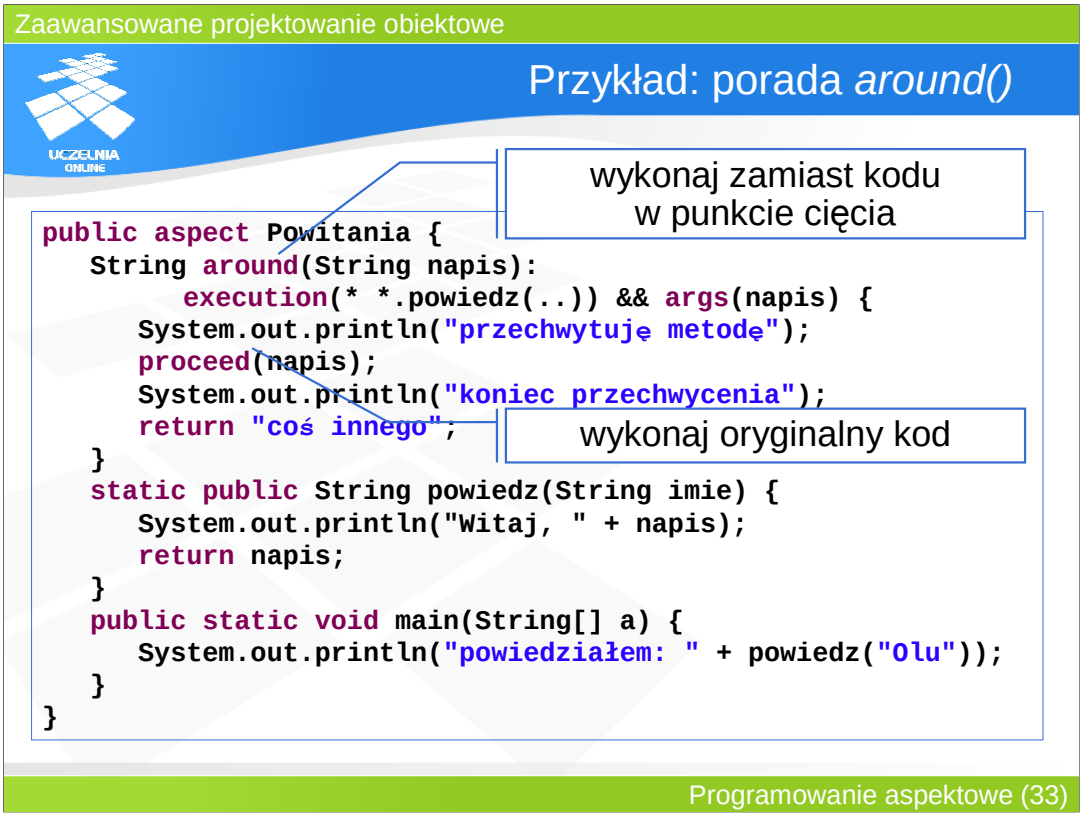
Przykład: aspekt logujący wywołania metod

```
public aspect Logger {
  pointcut wywolanieMetody() :
    call(* * (..)) && ! within(Logger);

  before() : wywolanieMetody() {
    System.out.println("Przed wywołaniem metody "
      + thisJoinPoint.getSignature());
  }
  after() returning : wywolanieMetody() {
    System.out.println("Po wywołaniu metody "
      + thisJoinPoint.getSignature());
  }
  after() throwing  : wywolanieMetody() {
    System.out.println("Metoda " +
      thisJoinPoint.getSignature() + " zgłosiła wyjątek");
  }
}
```


Na slajdzie przedstawiono przykład wspomnianego wcześniej aspektu służącego do logowania wywołań metod w kodzie programu. Aspekt Logger definiuje jeden punkt cięcia o nazwie *wywołanieMetody().* Dotyczy on wywołań wszystkich metod o dowolnym zasięgu widoczności, typie, znajdujących się w dowolnym pakiecie i o dowolnych parametrach, o ile nie znajdują się w aspekcie Logger. Warto zwrócić uwagę właśnie na to ograniczenie: jego brak spowodowałby uruchamianie porad także dla wywołań metod wewnątrz tych porad, co skończyłoby się zapętleniem lub przynajmniej niepożądanym zachowaniem programu.

W aspekcie znajdują się także trzy porady, dotyczące odpowiednio momentów przed osiągnięciem punktu cięcia *wywolanieMetody()*, po jego osiągnięciu w przypadku prawidłowego zakończenia oraz w przypadku zgłoszenia wyjątku. Wszystkie trzy porady wyświetlają stosowny komunikat związany z wywołaniem metody, powrotem z wywołania oraz zgłoszeniem wyjątku. Odwołania do zmiennej *thisJoinPoint* pozwalają na odczyt sygnatury metody, która aktualnie jest wywoływana.



Przykładem zastosowania porady *around()* jest przedstawiony na slajdzie aspekt. Porada *around()* zbiera kontekst, którym jest parametr wywołania oryginalnej metody o nazwie rozpoczynającej się od słowa *powiedz().* Treść porady składa się z polecenia wyświetlenia napisu "przechwytuję metodę", następnie wywołania oryginalnej metody dopasowanej przez poradę (służąca do tego klauzula *proceed()* może przyjmować parametry odpowiadające parametrami kontekstu, w jakim została wykonana porada), wyświetleniu kolejnego napisu i zwróceniu jako wyniku napisu "coś innego". W ten sposób została zdefiniowana alternatywna treść metody *powiedz().*

Metoda *powiedz()* przyjmuje parametr będący imieniem osoby do pozdrowienia, wyświetla pozdrowienia, a następnie zwraca przekazane imię.

W metodzie *main()* metoda *powiedz()* zostaje wykonana z argumentem "Olu". Powoduje to wywołanie wskazanej metody, jednak już po uruchomieniu jej osiągnięty zostanie punkt złączenia opisany w poradzie. Treść metody *powiedz()* będzie efektywnie zastąpiona przez treść porady. Ponieważ jednak porada wywołuje oryginalną metodę, więc efekt jej działania w postaci napisu także będzie widoczny. Zmianie ulegnie jednak wartość zwracana przez metodę: zamiast napisu "Olu" będzie ona brzmiała "coś innego".

Wynikiem uruchomienia tego aspektu będzie zatem sekwencja:

```
przechwytuję metodę
Witaj, Olu
koniec przechwycenia
powiedziałem: coś innego
```

Przykład: do czego służy ten aspekt?

Jaki jest efekt użycia tego aspektu?

```
aspect A {
   before(): call(* *(..)) {
      System.out.println("przed");
   }
   after(): call(* *(..)) {
      System.out.println("po");
   }
}
```

http://eclipse.org/aspectj

Po omówieniu podstawowych elementów aspektów oraz przedstawieniu przykładów, można przejść do bardziej zaawansowanych rozwiązań.

Na slajdzie przedstawiono aspekt, który przechwytuje wywołania wszystkich metod przed i po tym fakcie, wywołując w odpowiedzi metody *System.out.println().*

Co wydarzy się po uruchomieniu jakiegokolwiek kodu skompilowanego razem z tym z tym aspektem?

Otóż program ten nie wyświetli żadnego napisu i zakończy się. Przyczyną jest tzw. cichy błąd – zgłoszenie wyjątku, który jednak nie będzie zaprezentowany użytkownikowi.

Wynika to z konstrukcji punktu cięcia: osiągany jest on w momencie wywołania dowolnej metody w dowolnej klasie. A zatem, wywołanie w programie pierwszej metody spowoduje uruchomienie najpierw porady *before()* i związanego z nią kodu. Kod ten składa się również z wywołania metody, co również spowoduje uruchomienie porady *before()*... Spowoduje to rekurencyjne zapętlenie, aż do pojawienia się wyjątku związanego z przepełnieniem stosu wywołań metod. Jednak informacja o nim również nie pojawi się, ponieważ do akcji wkroczy porada *after()*, powodująca identyczne zapętlenie rekurencyjne. Skutkiem będzie zatem ciche zakończenie programu.

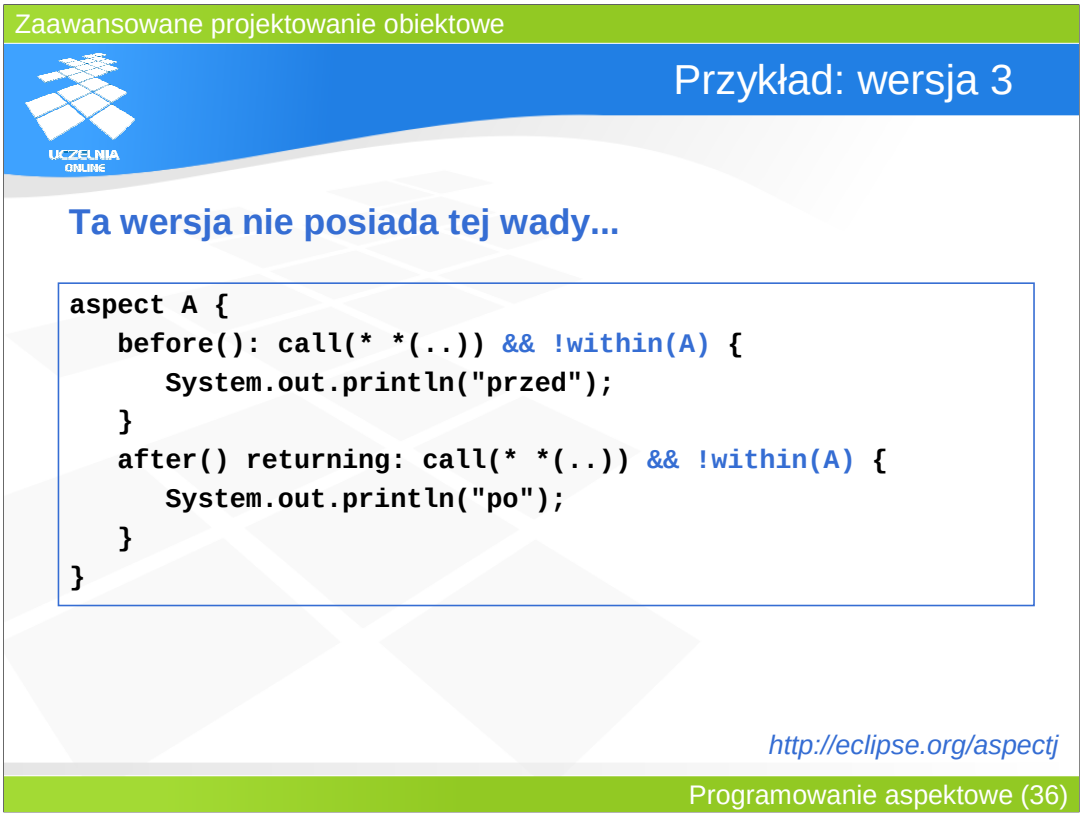
Przykład: wersja 2

Jaki jest efekt użycia tego aspektu?

```
aspect A {
   before(): call(* *(..)) {
      System.out.println("przed");
   }
   after() returning: call(* *(..)) {
      System.out.println("po");
   }
}
```

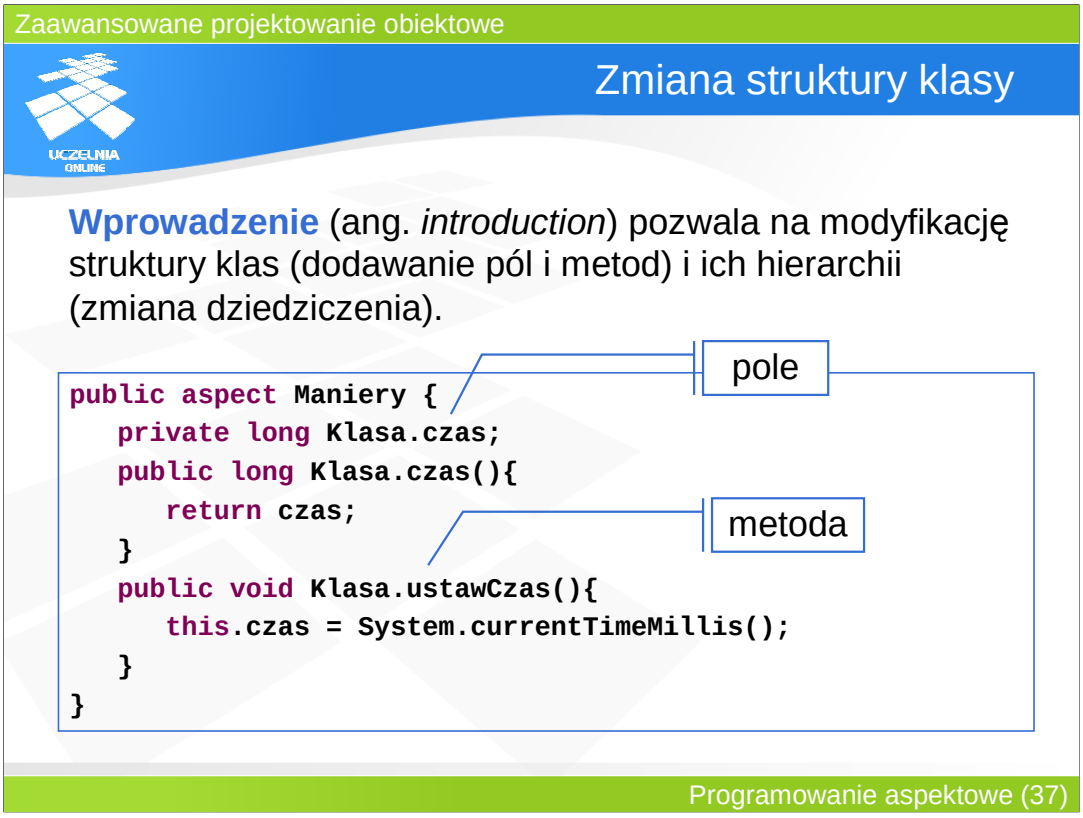
http://eclipse.org/aspectj

Pierwszym sposobem poprawy aspektu jest wprowadzenie do porady *after()* klauzuli *returning* – zostanie ona wywołana tylko po prawidłowym zakończeniu przechwytywanego kodu. W opisanym wcześniej przypadku przyczyną usunięcia informacji o wyjątku przez zapętlającą się rekurencyjnie poradę *after()*, dlatego zawężenie jej stosowania pozwoli uzyskać przynajmniej wydruk stosu śladu wyjątku.



Oczywiście, nie to jest zadaniem projektowanego aspektu. Osiągnięcie założonego celu, czyli wskazywania wywoływania metod w kodzie programu (a nie w aspekcie!) wymaga dodania ograniczenia *!within(A)*, które wyklucza kod znajdujący się wewnątrz tego aspektu spod analizy osiągnięcia punktu cięcia.

Tak zapisany aspekt poprawnie wykona zadanie.



Poza modyfikacją zachowania programu, aspekty mogą także zmieniać jego strukturę. Modyfikacje struktury kodu są znany pod nazwą wprowadzeń (ang. *introductions*).

Możliwe zmiany obejmują

•dodanie nowych składowych (konstruktorów, metod i pól) w wybranej klasie lub klasach

•dodanie implementacji interfejsu do wybranej klasy

•wprowadzenie dziedziczenia klasy po innej klasie

•zmianę poziomu widoczności składowych w klasie.

Aspekt dodaje składowe do klasy, podając ich deklaracje poprzedzone nazwą klasy. Specyfikując zasięg widoczności składowej należy pamiętać, że odnosi się on do bieżącego aspektu, a nie docelowej klasy. Oznacza to, że oryginalne pola i metody prywatne klasy nie będą miały dostępu do składowej zdefiniowanej w postaci aspektu.

Definiowanie nowych składowych rodzi również niebezpieczeństwo konfliktu w sytuacji, gdy składowa taka już istnieje w docelowej klasie. Kompilator aspektów zapobiega takiej sytuacji, sygnalizując błąd kompilacji. Konflikty pomiędzy różnymi aspektami są rozwiązywane w bardziej zaawansowany sposób, który jednak nie będzie omawiany.

W podanym przykładzie aspekt Maniery dodaje do klasy Klasa prywtne pole *czas* oraz metodę *ustawCzas()*.

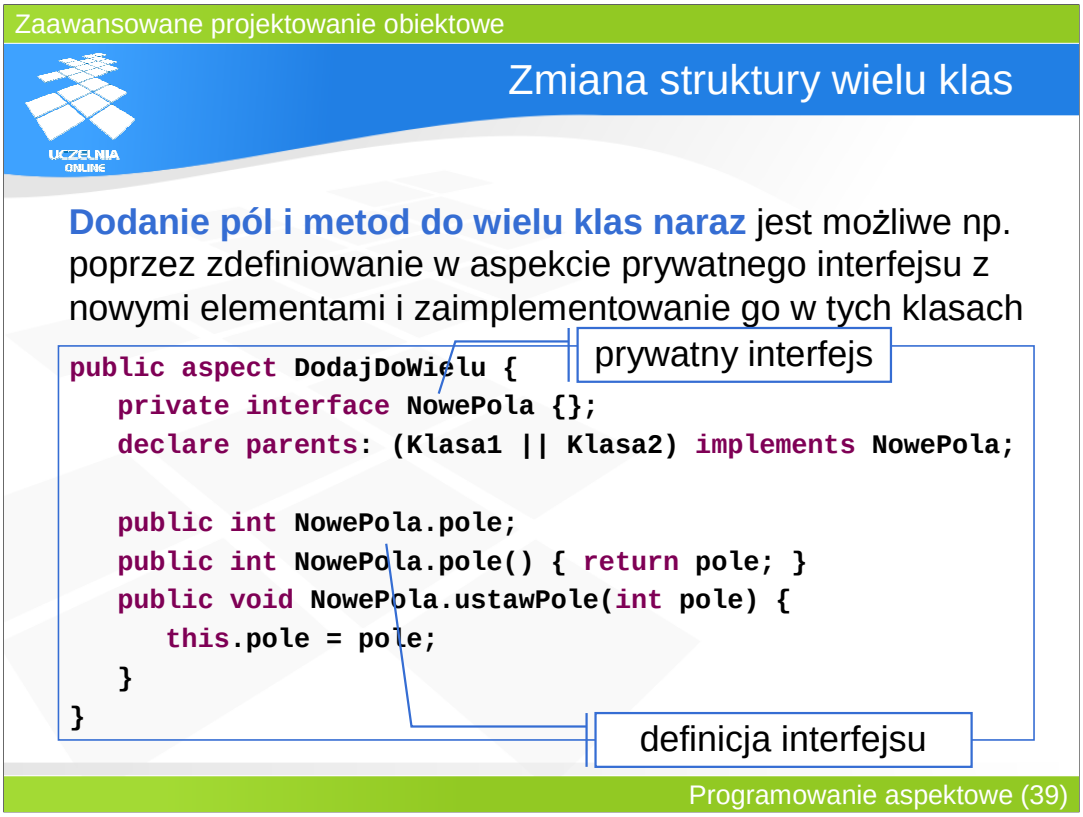
Zmiana hierarchii dziedziczenia

Polecenie *declare parents* pozwala zmienić hierarchię dzidziczenia wybranych klas.

```
public aspect DziedziczenieManier {
   declare parents: HelloWorld extends Zachowanie;
   declare parents: HelloWorld implements Observer;
}
```


Ustanowienie dziedziczenia w klasie za pomocą aspekty wymaga użycia nieco innej składni. Deklaracja zaczyna się od słowa kluczowego *declare*, po którym następuje określenie rodzaju deklaracji (w tym przypadku *parents*) i jej treść.

W przypadku dziedziczenia treścią deklaracji jest stwierdzenie, że klasa A ma dziedziczyć po B (*A extends B*). Implementacja interfejsu wymaga użycia konstrukcji *A implements B*.

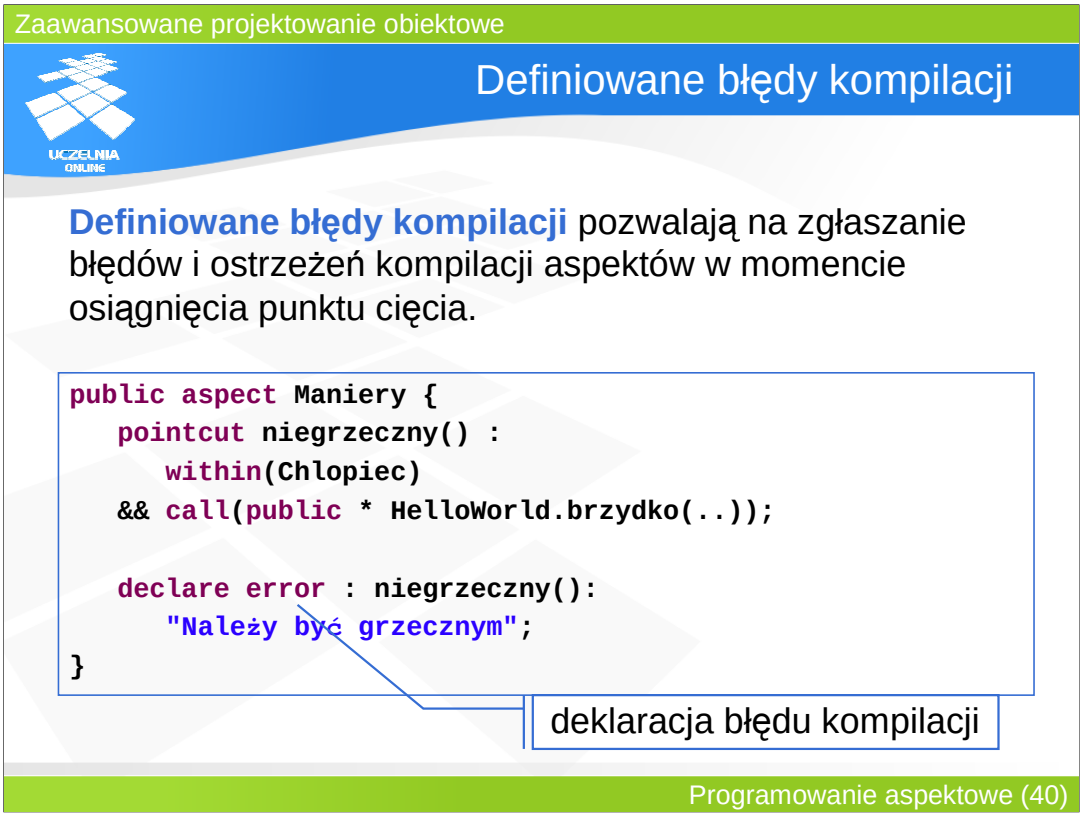


Wprowadzenie jest przewidziane jako metoda dodawania składowych do jednej klasy naraz, dlatego modyfikacje wielu klas zasadniczo wymagają zdefiniowania oddzielnych konstrukcji dla każdej z nich.

Jednak można zastosować też inne rozwiązanie: można zadeklarować pola i metody wewnątrz interfejsu, a następnie zadeklarować jego implementację w grupie klas.

Koncepcję tę wyjaśnia przykład: Pierwszym krokiem jest zdefiniowanie pustego prywatnego interfejsu o nazwie NowePola. Następnie następuje deklaracja implementacji tego interfejsu w wybranych klasach, określonych za pomocą alternatywy ich nazw. Ostatnim krokiem jest zdefiniowanie składowych nowego interfejsu za pomocą aspektów.

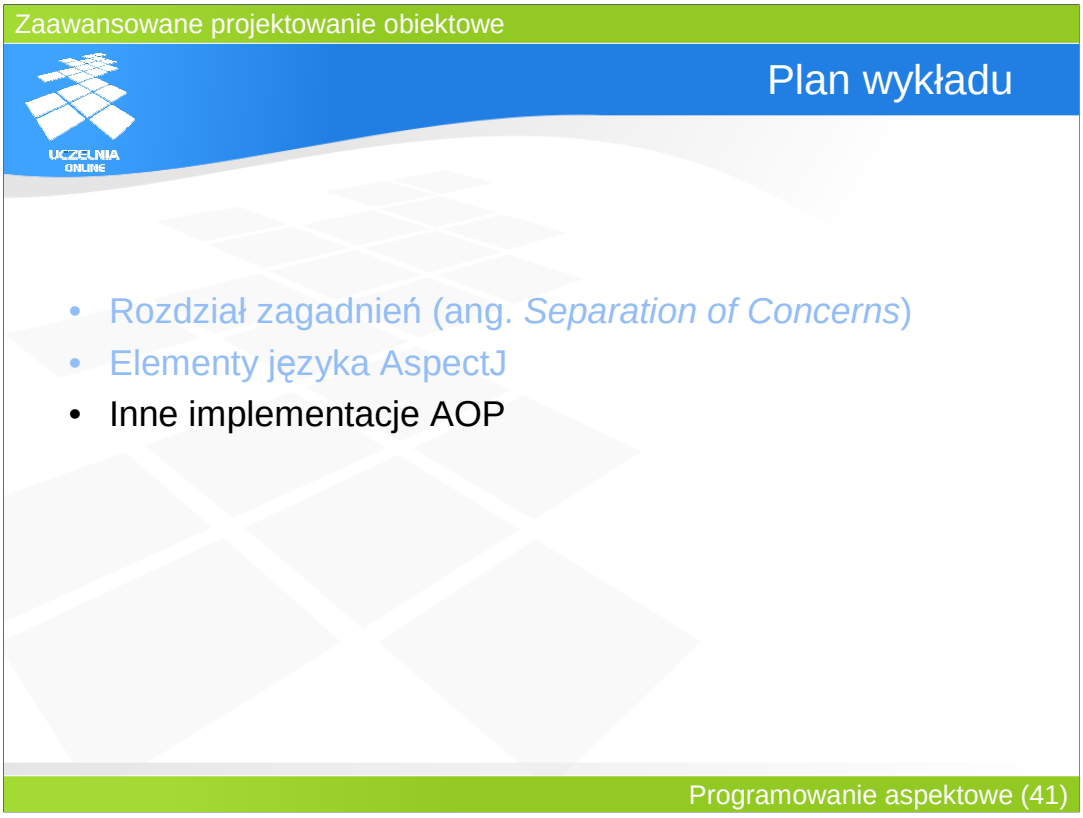
W ten sposób każda klasa implementująca nowy interfejs otrzymuje nowe pole oraz metody dostępu do niego.



Bardzo ciekawą cechą AspectJ jest możliwość przeniesienia niektórych błędów, które zwykle można wykryć dopiero w momencie uruchomienia kodu, na poziom kompilacji. W ten sposób można określić, które punkty cięcia nie powinny być nigdy osiągnięte, a ich osiągnięcie jest niepoprawnym stanem programu.

Służy do tego polecenie *declare error*, które określa punkt cięcia oraz komunikat wyświetlany przez kompilator. Podobnym poleceniem jest *declare warning*, które podczas kompilacji zgłasza ostrzeżenie.

Mechanizm ten ma ograniczenia: może służyć jedynie do wskazywania błędów kompilacji w stosunku do konstrukcji, które można określić statycznie. Można zatem w nim wykorzystać jedynie takie punkty cięcia, które posiadają tę właściwość: cflow, cflowbelow, this, target, args, if.



Na tym zakończymy omawianie najważniejszych cech systemu AspectJ. Obecnie przedstawione zostaną wybrane inne mechanizmy aspektowe dostępne dla programów napisanych w języku Java. Warto zauważyć, że pomimo różnic sposobie wyrażania poszczególnych elementów aspektowych oraz ich implementacji, podstawowe koncepcje pozostają niezmienione.

Szczegółowe porównanie większej liczby bibliotek aspektowych można znaleźć w artykule M. Kirstena pod adresem http://www-128.ibm.com/developerworks/java/library/j-aopwork1/

Inne systemy aspektowe: AspektWerkz

Spring AOP (http://aspectwerkz.codehaus.org)

- obecnie włączony do AspectJ 1.5
- brak własnego języka, aspekt jest klasą
- typowy zestaw punktów złączenia
- łączenie klas i aspektów na poziomie bajtkodu
- wysoka wydajność
- wsparcie dla anotacji języka Java (JDK 5.0+)
- definiowanie porad i punktów złączeń w XML

Przegląd innych rozwiązań aspektowych dla języka Java zaczyna się od biblioteki AspectWerkz. System ten był rozwijany i wspierany przez firmę Bea, jednak obecnie został on połączony z AspectJ. W przyszłości oba będą oferowały identyczną funkcjonalność, którą będzie można wyrazić na sposób charakterystyczny dla AspectJ lub AspectWerkz.

AspectWerks, w odróżnieniu od AspectJ, nie posiada specjalizowanego języka do wyrażania punktów cięcia i porad – aspekty są w nim zapisywane w postaci zwykłych klas języka Java. Do wyrażania elementów aspektowych wykorzystywane są anotacje, które definiują aspekty, punkty cięcia i porady. Oferuje typowy zestaw punktów złączenia, obejmujący przede wszystkim wywołanie metody, obsługę wyjątków, odwołania do pól etc., jednak weryfikacja wszystkich punktów cięcia jest wykonywana w trakcie uruchomienia program. Brak statycznych punktów złączenia jest poważnym brakiem, w efekcie którego prawdopodobieństwo popełnienia prostego błędu jest wyższe niż w przypadku AspectJ.

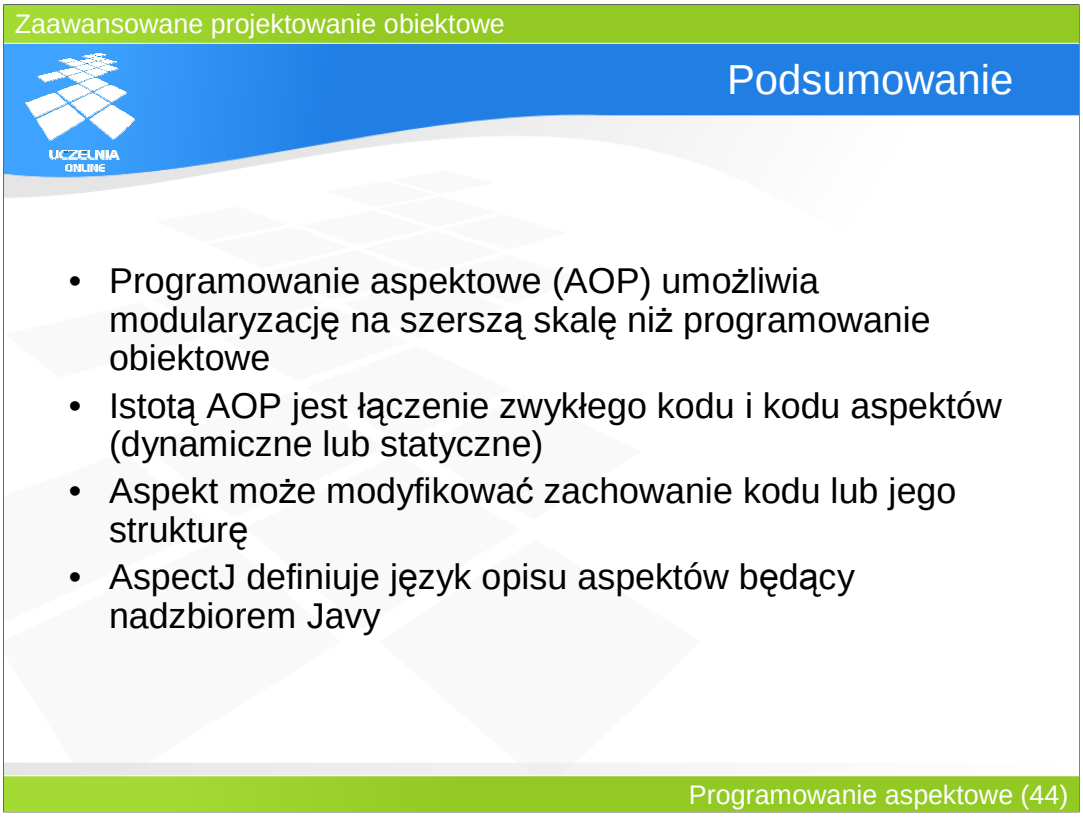
Inne systemy aspektowe: Spring AOP

Spring AOP (http://springframework.org)

- cel: integracja AOP wewnątrz Spring IoC
- aspekt jest komponentem Spring
- jeden punkt złączenia: wywołanie metody
- łączenie kodu na poziomie wykonania (*Java Dynamic Proxy*)
- konfiguracja poprzez kontener Springa

Mechanizm Spring AOP jest blisko związany z kontenerem IoC Spring (zob. wykład nt. programowania komponentowego). Aspekty są definiowane i konfigurowane identycznie jak inne komponenty Springa, w pliku definiującym rejestr kontenera.

Spring AOP stosuje inny mechanizm tkania niż przedstawione wcześniej systemy: punkty cięcia są obliczane w trakcie wykonywania kodu, a nie w momencie kompilacji, dlatego poszczególne komponenty pozostają od siebie binarnie niezależne. Jedynym dostępnym rodzajem punktu złączenia jest wywołanie metody, co stanowi poważny niedostatek tego systemu (choć, zdaniem twórców Spring AOP, była to ich świadoma decyzja projektowa). Do przechwytywania wywołań metod wykorzystywany jest wbudowany w maszynę wirtualną Javy mechanizm *dynamic proxy*.



Programowanie aspektowe stanowi nowy trend w programowaniu. Pozwala, w odróżnieniu od zwykłych mechanizmów obiektowych, stosować modularyzację, abstrakcję i hermetyzację na dużo większą skalę niż było to możliwe dotychczas. Istota programowania aspektowego polega na możliwości łączenia zwykłego kodu z kodem przecinającym, zgrupowanym w aspektach. W efekcie program jest wykonywany tak, jakby fizycznie stanowił połączenie różnych modułów, choć w rzeczywistości poszczególne części kodu są od siebie oddzielone. Łączenie może odbywać się na poziomie kompilacji kodu, jego ładowania lub wykonania.

Aspekty mają niemal nieograniczone możliwości w zakresie modyfikacji programu, poczynając od jego zachowania aż po zmianę jego struktury. Dzięki temu ich zastosowania nie są bardzo szerokie.

Pierwszą implementacją koncepcji aspektów stanowił język AspectJ, będący nadzbiorem języka Java. Jako jedno z nielicznych rozwiązań posiada on własny język opisu aspektów, wzorowany na języku Java. Jego cechami charakterystycznymi są silne wsparcie w środowisku IDE Eclipse oraz bogaty repertuar rodzajów punktów złączenia kodu i aspektów. Inne systemy aspektowe pozostają rozwiązaniami niszowymi.

Programowanie aspektowe przeżywa w ostatnich latach bujny rozwój, który prognozuje wzrost zainteresowania tą techniką także w przyszłości oraz powstanie nowych obszarów zastosowań.