**Czym jest AspectJ?**

AspectJ to język programowania,realizuje paradygmat programowania aspektowego (AOP) (czyli polegającym na rozwiązywaniu zagadnień wspólnych dla wielu niezależnych komponentów/modułów projektu). 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 opisuje zagadnienie, którego [implementacja](https://pl.wikipedia.org/wiki/Implementacja_(informatyka)) przecina więcej niż jedną klasę. 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. Kod źródłowy podzielony na klasy i aspekty jest kompilowany do kodu bajtowego, w który aspekty zostają wplecione. AspectJ jest pełnym językiem programowania: oprócz składni posiada także własne narzędzia – kompilator i debugger. Jest zintegrowany z Eclipsem.

**Różnica pomiędzy programowaniem obiektowym a aspektowym**

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.

Programowanie aspektowe definiuje dodatkowy mechanizm grupowania jakim jest aspekt , reprezentujący pojedyncze zagadnienie. 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.

**Punkty złączeń**

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 są:

•wywołanie i wykonanie dowolnej metody i dowolnego konstruktora

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

•statyczna inicjacja klasy

•obsługa wyjątku

**Punkty przecięcia**

Punkt cięcia jest zdefiniowaną kolekcją punktów złączenia. Punkty przecięcia grupują dowolne zbiory punktów złączeń.

Punkty przecięcia można podzielić według dwóch podstawowych kryteriów:

* Według złożoności:
  + proste punkty przecięcia – definiujące jeden punkt złączenia,
  + punkty złożone – zbudowane z kolekcji punktów złączenia powiązanych operatorami logicznymi (i, lub, negacja)
* według tożsamości:
  + nazwane punkty przecięcia - mogą być wielokrotnie używane w różnych miejscach aplikacji,
  + anonimowe punkty przecięcia - są tworzone w miejscu ich użycia i nie można odwołać się do nich z innej części systemu.

Punkt cięcia **call** reprezentuje wywołania metod i konstruktorów (przed wykonaniem)

Najpopularniejszym rodzajem punktu cięcia jest wywołanie metody. Poprzez wywołanie należy rozumieć moment po wydaniu polecenia wykonania tej metody, 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 (..).

Punkt cięcia **execution** reprezentuje wykonanie ciała 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.

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

Punkt cięcia **handler** reprezentuje kod obsługi wyjątku

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

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

Punkty cięcia within i withincode są osiągane w momencie, gdy sterowanie znajdzie się odpowiednio wewnątrz podanej klasy lub metody. Parametrem punktu within jest klasa, natomiast punktu withincode – sygnatura metody, które są analizowane.

Zmienna **thisJoinPoint** przechowuje informację o miejscu osiągnięcia punktu złączenia

Zmienna pozwala na odczyt m.in. sygnatury wywołanej metody, klasy i obiektu wykonującego obecnie kod, listy i wartości parametrów.

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

**Rady**

**Porada** jest fragmentem kodu programu wykonywanym przed, po lub zamiast osiągnięcia przez

program punktu cięcia.

Definicja rady w AspectJ składa się z określenia momentu uruchomienia porady, definicji punktu przecięcia oraz ciała rady.

Aby wybrać pożądany należy wybrać jedną z poniższych rad:

* **before()** - wykonywana jest tuż przed wystąpieniem punktu przecięcia. Ten typ rady może służyć na przykład do weryfikacji parametrów metody, zmiany sposobu sterowania etc.
* **after()** - wykonywana jest zaraz po wystąpieniu punktu cięcia. Sam moment wystąpienia punktu przecięcia jest nieprecyzyjny, dlatego można wyróżnić dwie sytuacje, które ta rada opisuje:
  + taką w której wykonanie oryginalnego kodu w punkcie cięcia zakończyło się poprawnie (sytuacja taka opisywana jest klauzulą after() returning)
  + taką w której efektem wykonania kodu było zgłoszenie wyjątku (opisywane klauzulą after() throwing)
* **around()** - rada ta w odróżnieniu od dwóch poprzednich, posiada możliwość zastąpienia oryginalnego kodu związanego z punktem cięcia treścią porady oraz możliwością zezwolenia lub odmówienia wykonania punktu przecięcia za pomocą polecenia proceed().

**Aspekt w języku AspektJ**

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.

**Tkacz**

Proces łączenia kodu aspektpwego z nie aspektowym (ze zwykłym kodem Javy) nazywany jest tkaniem. Zadaniem tym zajmuje się tkacz.

**Zasada działania/tkacz**

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.

**Aspekt Logger**

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.

**Wprowadzenie** pozwala na modyfikacje struktury klas (dodawanie pól i metod) i ich hierarchii (zmiana dziedziczenia).

Poza modyfikacją zachowania programu, aspekty mogą także zmieniać jego strukturę. Modyfikacje struktury kodu są znany pod nazwą wprowadzeń. 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. W podanym przykładzie aspekt Maniery dodaje do klasy Klasa prywatne pole czas oraz metodę ustawCzas().