Opis programowania agantowego

**Programowanie obiektowe kontra programowanie agentowe**

Programowanie obiektowego (ang. object-oriented programming, OOP) jest obecnie najpopularniejszym i najważniejszym paradygmatem stosowanym do tworzenia oprogramowania. Stanowi on również podstawę dla innych paradygmatów takich jak programowanie aspektowe (ang. aspect-oriented progrming, AOP) oraz programowanie generyczne (ang. generic programming). [1] Programowanie obiektowe może przybrać również bardziej wyspecjalizowaną formę nazywaną programowaniem agentowym (ang. agent-oriented progrming, AOP). [3] Poniższe akapity przedstawią oba paradygmaty oraz ich porównanie.

**Definicja i opis programowania Obiektowego**

Programowanie obiektowe jest paradygmatem, w którym do tworzenia programów wykorzystuje się elementy zwane obiektami. Obiekty stanowią abstrakcyjne byty programistyczne, które stanowią przestrzeń zagadnień rozwiązywanych przez program. Ten fakt nadaje programowi cechę dostosowywania się do języka danego zagadnienia. Oznacza to rozpatrywanie zagadnień we właściwych, bliższych myśleniu człowieka, kategoriach, w odróżnieniu od kategorii maszyny, na której uruchomiony jest program. Przykładem może być prosta układanka składająca się z pewnej liczby elementów przesuwanych po planszy przez użytkownika. Elementy układanki, jak i sama plansza mogą stanowić obiekty, które programista jest w stanie wykorzystać do napisania programu. Każdy obiekt może posiadać pewne dane wewnętrzne odpowiadające za jego stan, zestaw funkcji lub metod będącymi jego zachowaniem oraz możliwość odróżnienia danego obiektu od innego, zwana tożsamością. Informacje na ten temat przechowywane są w klasie. Zmienne należące do klasy nazywane są polami klasy, a obiekt utworzony na podstawie klasy jej instancją. Przykład kodu klasy dla języka Java został zawarty w listingu 1.

**public** **class** Account {

**private** **double** balance;

**public** **double** getBalance() {

**return** balance;

}

**public** **void** deposit(**double** depositValue)

{

balance += depositValue;

}

**public** **boolean** withdraw(**double** withdrawValue)

{

**if**(withdrawValue <= balance){

balance -= withdrawValue;

**return** **true**;

}

**return** **false**;

}

}

Listing 1. Klasa stanowiąca implementacje konta bankowego.

Przykładowa klasa zaprezentowana na listingu 1 posiada zmienną typu zmiennoprzecinkowego, która stanowi wartość depozytu bankowego. Każda instancja tej klasy posiadać będzie własną wartość tego pola. Zachowanie obiektów klasy Account zdefiniowane jest w funkcji withdraw i metodzie deposit służących do zmiany wartości konta oraz w metodzie getBalance zwracającej aktualny stan konta. Wspomniane metody i funkcje stanowią interfejs klasy Account, ponieważ określają jakie akcje lub żądania mogą zostać wykonane względem obiektów tej klasy. Z tego wynika, że zachowanie i rola obiektów w programie zależy od klasy którą dany obiekt reprezentuje. Obiekty tworzące program mogą komunikować się ze sobą w celu wykonywania zaprogramowanych dla nich zadań. Sposób w jaki to robią polega na wzajemnym wywoływaniu przez nie operacji . Najczęściej wykorzystywaną techniką umożliwiająca komunikację między obiektami jest posiadanie jednego obiektu przez inny. Powyższe twierdzenie może zilustrować może dowolna klasa posiadająca na przykład pole typ Account. Obiekty takiej klasy mogłyby komunikować się z posiadanymi przez nie obiektami poprzez wartości argumentów, wywoływanych przez nie, metod i funkcji .

Obecnie OOP dzieli się na dwa podtypy które odróżniają się odmiennym sposobem tworzenia obiektów. Jest to programowanie oparte o wspomniane wyżej klasy. Ten podtyp jest wykorzystywany w językach takich jak Java, C++, C# i wiele innych. [2] Drugim podtypem jest programowanie oparte o prototyp(ang. Prototype-based programming) gdzie nowy obiekt tworzony jest w oparciu o już istniejący, wykorzystywanym w głównej mierze w językach skryptowych, takich jak Perl, Java Script czy LUA. [4]

Najważniejsze założenia paradygmatu obiektowego to:

• Abstrakcja – polega na pomijaniu pewnych szczegółów w celu uproszczenia sposobu rozwiązywania problemów. Przykładem może być obiekt reprezentujący samochód, który w zależności od systemu może zawierać cechy takie jak barwa, numer rejestracyjny i dane właściciela lub szczegółowe wymiary, numer silnika i dane fabryki która go wyprodukowała. Oba przypadki odrzucają pewne cechy i skupiają się tylko na tych które program potrzebuje. [2]

• Hermetyzacja – pomaga chronić obiekty przed niepowołaną zmianą stanu lub użyciem funkcji poprzez wprowadzenie kontroli dostępu do wszystkich jej składników. To wymusza korzystanie z klasy tylko w założony przez jej twórcę sposób. Eliminuje to konieczność zabezpieczania programu przed skutkami ubocznymi użycia klasy w błędny sposób oraz pozwala na zmiany wewnętrznych, chronionych mechanizmów bez utraty kompatybilności z resztą programu. Z punktu widzenia użytkownika klasy, hermetyzacja pozwala na odróżnienie elementów dla niego istotnych, do których ma dostęp, od tych które powinien zignorować. W obiektowych językach programowania, takich jak Java lub C++ używa się najczęściej trzech modyfikatorów dostępu. Public oznacza, że dana funkcja lub pole jest widoczne poza klasą, private – tylko w obrębie danej klasy albo protected, kiedy funkcja, czy pole są chronione jedynie przed dostępem z zewnątrz.[2]

• Dziedziczenie – jest mechanizmem, który pozwala rozszerzyć lub zmienić zachowanie klasy tworząc na jej podstawie nową klasę lub klasy. Klasa, po której następuje dziedziczenie jest nazywana klasą bazową, a klasa dziedzicząca klasą pochodną. Dziedziczenie, nie tylko pozwala na rozszerzanie klas, ale tworzy też pewną hierarchę. Przykładem może być klasa „zwierzę” dla której klasami pochodnymi są klasy „słoń” , „lew” i „zebra”. Klasy pochodne (te które nie są chronione przed dziedziczeniem) posiadają cechy klasy „zwierzę” oraz dodają do nich swoje bardziej wyspecjalizowane cechy lub zachowania. Ukazaną hierarchię można określić jako przejście od jednego ogólnego typu do wielu wyspecjalizowanych typów. W zależności od języka programowania klasa pochodna może dziedziczyć tylko z jednej lub z wielu klas bazowych. [2]

• Polimorfizm – jest związany bezpośrednio z dziedziczeniem. Pozwala on na używanie instancji klas pochodnych, tak jakby były typu bazowego, jednak obiekt wciąż zachowuje swoje oryginalne zachowanie. Oznacza to że obecny typ obiektu ma wpływ jedynie na to jakie funkcje możemy użyć, a nie na to z jakiej klasy zostaną wywołane. Jest to możliwe dzięki mechanizmowi zwanemu późnym wiązaniem (ang. late binding) pozwalającym na wykonanie odpowiedniego kodu dopiero w czasie działania programu. Polimorfizm umożliwia bardzo elastyczne rozszerzanie programu o nowe typy pochodne, bez konieczności zmiany implementacji funkcji, które z nich korzystają. [2]

**Rozwój programowania obiektowego**

Programowanie obiektowe wywodzi się z grupy języków imperatywnych, które charakteryzują się wykonywaniem w ustalonej kolejności następujących po sobie instrukcji tworzących program. Bezpośrednim przodkiem OOP był paradygmat programowania strukturalnego, tj. program go wykorzystujący był podzielony na struktury zwane podprogramami posiadające jeden punkt wejścia i wyjścia. Programowanie obiektowe rozszerzyło paradygmat strukturalny o dodanie bardziej abstrakcyjnych elementów jakimi są klasy i obiekty.[5] **Implementację (zamienic na inne słowo)** OOP stanowią obiektowe języki programowania. Pierwszym językiem obiektowym była Simula 67, którego zadaniem było tworzenie symulacji metodą kolejnych zdarzeń (ang. discrete event simulation). Jego autorami byli Ole-Johana Dahla oraz Kristena Nygaarda z Norweskiego Ośrodka Obliczeniowego. Język ten wprowadził jako pierwszy pojęcie klas i obiektów jako instancji klas. Simula 67 została formalnie zdefiniowana w roku 1967, trzy lata później tworząc język Smalltalk firma Xerox PARC wprowadziła pojęcie programowania zorientowanego obiektowo. Pomimo inspiracją Simulą 67 nowy język był przede wszystkim dynamiczny, obiekty w nim mogły być dowolnie tworzone, zmieniane oraz usuwane. Smalltalk wyróżniał się także wprowadzeniem dziedziczenia oraz był pierwszym w pełni obiektowym językiem programowania[6][7]. Jednym z jego twórców był Alan Kay który podsumował pięć cech tego języka[2]:

1. *Wszystko jest obiektem.*
2. *Program jest zbiorem obiektów które po przez wysysanie komunikatów mówią sobie nawzajem co robić.*
3. *Każdy obiekt posiada swoja własną pamięć na którą składają się inne obiekty.*
4. *Każdy obiekt posiada swój typ.*
5. *Wszystkie obiekty danego typu mogą otrzymać te same komunikaty.*

OOP spopularyzowało się dopiero po roku 1980 za sprawą języka C z klasami (ang. C with classes), a od 1983 roku C++. Jego projektantem był duński informatyk Bjarne Stroustrup. Bezpośrednim poprzednikiem C++ był język C, z którego zaczerpnięto składnię, natomiast z języka Simula 67 funkcje wirtualne oraz dziedziczenie, które umożliwia klasie pochodnej posiadanie wielu klas bazowych. C++ jest językiem wieloparadygmatowym, co oznacza, że umożliwia tworzenie programów nie tylko obiektowo, ale także strukturalnie. Jest to pierwszy język, który wprowadził pojęcie konstruktora oraz destruktora, czyli specjalnych funkcji uruchamianych jeszcze przed stworzeniem i usunięciem obiektu. [3] W Zurychu Niklaus Wirth oraz jego współpracownicy rozpoczęli badania nad abstrakcją oraz programowaniem modułowym na bazie, których powstały język programowania Modula-2, a na jego podstawie Oberon. Podejście do OOP w Oberonie różniło się od tego znanego w C++. Na przykład pojecie klasy zostało zastąpione rekordem(ang. record type). Wraz z rozwojem programowania coraz większa ilość języków stawała się obiektowa, między innymi Ada, BASIC, Fortran i Pascal. Powodowało to jednak trudności w utrzymywaniu kodu oraz kompatybilności programów, które nie były tworzone według idei obiektowej. W kolejnych latach powstały dwa znaczące języki programowania podobne do C++ pod względem składni oraz połączenia programowania obiektowego z elementami paradygmatu proceduralnego. Była to Java stworzona przez Sun Microsystems, która zdobyła popularność głównie dzięki wieloplatformowości oraz C# firmy Microsoft. Cechą charakterystyczną obu języków, która wystąpiła po raz pierwszy w języku smalltalk, jest kompilacja do kodu bajtowego wykonywanego w odpowiednim środowisku uruchomieniowym zwanym wirtualną maszyną. Obecnie większość języków obiektowych jest nadal rozwijana, a języki takie jak PHP w kolejnych wersjach coraz bardziej adaptują paradygmat obiektowy.[6][7]

Rozwój programowania obiektowo wpłynął nie tylko na kolejne języki programowania, ale również na sposoby i narzędzia służące projektowaniu programów. Wraz z upowszechnianiem się paradygmatu obiektowego powstawały nowe, niezależne od siebie, metody i notacje wspomagające modelowanie systemów obiektowych. Ich liczba znacznie wzrosła w późnych latach 80-tych i wczesnych latach 90-tych. Większość była do siebie podobna semantycznie, jednak różniły się na poziomie wizualizacji i opisywały jedynie część problematyki modelowania. Doprowadziło to do pracy nad jedną zunifikowaną metodą. Jej twórcami byli Grady Booch ( twórca metody OOAD), James Rumbaugh (notacja OMT) i od 1995 roku Ivar Jacobson (autor metody OOSE). Łącząc cechy swoich metod stworzyli UML 1.0 (ang. *Unified Modeling Language*). Wydany w 1997 roku przez firmę Rational Software, a dzięki nawiązaniu współpracy już w 1996 roku z organizacja OMG język ten zyskał ogromną popularność. UML składa się z dwóch części, notacji i semantyki. Notacja odpowiada za elementy graficzne oraz składnię języka modelowania, semantyka natomiast za definicje pojęć języka i powiązania pomiędzy nimi. Modelowanie polega na tworzeniu diagramów z elementów graficznych reprezentujących na przykład klasy lub przypadki użycia. W UML 2.0 zdefiniowanych jest trzynaście rodzajów diagramów podzielonych na trzy kategorie(diagramy strukturalne, zachowań oraz interakcji). Język ten w dalszym ciągu jest rozwijany**.** [8][9][10]

Atuty OOP takie jak abstrakcja i enkapsulacja umożliwiły powstanie wzorców projektowych, które można zdefiniować jako opis komunikacji oraz zależności klas i obiektów, który rozwiązuje pewne problemy projektowe w określonym kontekście. Historia wzorców projektowych sięga wczesnych lat 80-tych, gdy najpopularniejszym językiem obiektowym był wciąż Smalltalk. W tym czasie pojawił się szkielet służący budowie aplikacji nazywany Model-View-Controller, który był jednym z pierwszych wzorców projektowych. MVC rozwiązywał problem ze stworzeniem interfejsu użytkownika dla języka Smalltalk-80. Dzielił się on na trzy części: model danych ,widok prezentujący graficzny interfejs użytkownika i kontroler interakcji między użytkownikiem, a widokiem. Popularność wzorcom projektowym przyniosła publikacja z roku 1995, „*Design Patterns - Elements of Reusable Software*” jej autorami byli Erich Gamma, Richard Helm, Ralph Johnson oraz John Vlissides (nazywanych bandą czworga). Opisywała ona 24 wzorce podzielone na trzy kategorie:

* kreacyjne – odpowiedzialne za sposoby tworzenia obiektów,
* strukturalne – opisujące struktury tworzące przez obiekty,
* behawioralne – będące charakterystyką zachowań i interakcji między obiektami.

Obecnie liczba wzorców wzrosła i stały się one podstawą do tworzenia wielu systemów opartych o programowanie obiektowe.[11][12][13]

**Definicja i opis programowania agentowego**

Programowanie agentowe można zdefiniować jako paradygmat programowania wykorzystujący idee agentów (ang. Software agents), wysoce wyspecjalizowanych jednostek obdarzonych pewnymi cechami sztucznej inteligencji. AOP stanowi rozwinięcie paradygmatu obiektowego i zmienia sposób postrzegania programu komputerowego, nie jako ciągu instrukcji wykonywanych przez maszynę, lecz jedną lub wiele samodzielnych jednostek wykonujących założone przez programistę cele. [3]

Pojęcie agenta jest znacznie trudniejsze do zdefiniowania, wynika to z braku jednoznacznej i ogólnie akceptowanej definicji czym jest agent. Wiele proponowanych definicji posiada wspólne elementy takie jak określenie agenta autonomiczną jednostka. [14] Jedną z możliwości prezentuje w swoim opracowaniu Michael Wooldridge:

*„Agent jest to system komputerowy, który znajduje się w jakimś środowisku i jest zdolny do samodzielnego działania w tym środowisku w celu spełnienia swoich celów.”* [A]

Natomiast według Yoav Shoham:

„*Agent jest jednostką której stan jest postrzegany jako składający się z elementów takich jak przekonania, możliwości, wybory i zobowiązania.”* [B]

Ustosunkowując się do powyższych definicji, słowa Wooldridge można potraktować jako ogólną definicję budowy i działania agenta, natomiast Shoham przedstawił wewnętrzny stan agenta, który ma wpływ na jego zachowanie. [14][15] Integralną częścią systemu agentowego jest otoczenie agenta, czyli środowisko wspomniane w definicji Wooldridge

AGENT

Akcja

Sensor

ŚRODOWISKO

Rysunek 1. Abstrakcyjny schemat agenta według Wooldridge.

Rysunek 1 prezentuje sposób interakcji agenta z jego środowiskiem, który na podstawie danych uzyskanych za pośrednictwem sensora wykonuje akcje mające wpływ na środowisko. W zależności od domeny, w jakiej wykorzystywane jest programowanie agentowe, agent może nie mieć całkowitej kontroli nad środowiskiem z powodu jego skomplikowania. Oznacza to, że wykonywane akcje mogą prowadzić do niedeterministycznych rezultatów co z kolei może doprowadzić do uzyskania przez agenta rezultatów niekorzystnych z perspektywy jego celów. Przykładem środowiska niedeterministycznego może być Internet lub świat realny. Środowiskiem, w którym akcje generują pojedynczy i za każdym razem identyczny rezultat jest środowiskiem deterministycznym, na przykład program komputerowy. Pozostałe cechy otoczenia agenta zostały przedstawione w tabeli 1.

|  |  |
| --- | --- |
| Dostępność | Środowisko dostępne – agent jest w stanie pobrać wszystkie i zawsze aktualne stany środowiska. |
| Środowisko niedostępne – część stanów środowiska jest niedostępna dla agenta od razu lub w ogóle. |
| Dynamizm | Środowisko statyczne - zmiana w środowisku następuje tylko po akcjach agenta. |
| Środowisko dynamiczne – jest to typ środowiska do którego maja dostęp inni agencji lub na przykład procesy co za tym idzie stan środowiska między akcjami agenta może ulec zmianie. |
| Ilość stanów | Środowisko dyskretne – posiada stałą i skończona liczbę stanów. |
| Środowisko ciągłe – posiada nieskończoną liczbę stanów. |

Tabela 1. Klasyfikacja cech środowiska w programowaniu agentowym.

Podobnie jak cechy środowiska, cechy agenta rówież zostały sklasyfikowane. Michael Wooldridge wraz z Nicolas R. Jennings dzielą je pomiędzy dwa pojęcia agenta. Koncepcja słabego pojęcia agenta prezentuje go z perspektywy urządzenia lub programu komputerowego posiadającego następujące cechy:

* Autonomie(ang. autonomy) – oznacza to że agent nie jest uzależniony od człowieka, jego działania nie wymagają żadnych bezpośrednich integracji z człowiekiem lub jego kontroli. Agenta sam kontroluje swój wewnętrzy stan.
* Zdolności społeczne(ang. social ability) – agent potrafi komunikować się z innymi agentami lub z człowiekiem.
* Reaktywność(ang. reactivity) – po przez sensory agent jest w stanie reagować na zmiany stanów zachodzące w jego środowisku. Na przykład urządzenie będące agentem i posiadające sensor w postaci kamery potrafi zareagować na zbliżający się do niego obiekt.
* Pro aktywność(ang. Pro-activeness) - jest to umiejętność agenta do działania bez żadnego bodźca zewnętrznego jakim może być działania człowieka czy zmiana stanu środowiska by osiągnąć własne cele .

Silne pojęcie agenta przedstawia go jako jednostkę posiadając ludzkie cechy takie jak:

* przekonania (ang. belief ),
* intencje (ang. intention),
* wiedza (ang . knowledge),
* zobowiązania (ang. obligation).

Przypisywanie agentom powyższych cech jest charakterystyczne dla większość badacz zajmujących się sztuczna inteligencją takich jak Yoav Shoham. [14] Innymi cechami agenta mogą być między innymi:

* Mobilność (ang. mobility) – zdolność agenta do przemieszczania się po przez siec komputerową.
* Uczynność (ang. benevolance) – agenty nie posiadają sprzecznych celów i zawsze starają się wykonywać to o co są proszone.
* Racjonalność (ang. rationality) – agent stara się działać w taki sposób by nie pozbawić się możliwości wykonania swojego celu. [16]
* Koordynacja (ang. Coordination) – wiele agentów może wykonywać swoje akcje w dzielonym środowisku.
* Uczenie się lub adaptacyjność (ang. learning or adaptivity) - na ta cechę składa się umiejętność agenta do: elastycznego reagowania na zmiany w środowisku, przejęcia inicjatywy zgodnie z wyznaczonym celem jeśli jest to konieczne, uczenie się z własnego doświadczenia, środowiska, interakcji.
* Wiarygodność (ang. veracity) –założenie według którego agent świadomie nie będzie przekazywał fałszywych informacji. [17]

Podział typów agentów wraz z ich krótką charakterystyką

**Rozwój programowania agentowego**

62112045-Agent-Oriented-Programming.pdf

AOP-Shohan

[1]- <http://www.americanscientist.org/issues/pub/the-post-oop-paradigm>

[2] - Think In Java B. Eckel

[3] – Agent-oriented progrming –Yoav Shoham

[4] - http://en.wikipedia.org/wiki/Prototype-based\_programming

[5] - http://informatyka.umcs.lublin.pl/files/bylina.pdf

[6] - http://en.wikipedia.org/wiki/Object-oriented\_programming

[7] - <http://www.exforsys.com/tutorials/oops/the-history-of-object-oriented-programming.html>

[8]-http://wazniak.mimuw.edu.pl/images/7/76/Io-5-wyk.pdf

[9] -<http://www.uml.com.pl/modules/articles/article.php?id=10>

[10]-<http://www.omg.org/gettingstarted/what_is_uml.htm>]

[11] - James W. Cooper, THE DESIGN PATTERNS JAVA COMPANION, 1998

[12] -<http://smurf.mimuw.edu.pl/external_slides/Wzorce_projektowe/Wzorce_projektowe.html>

[13] - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley Pub Co 1995

[14] - http://www.msci.memphis.edu/~franklin/AgentProg.html

[14] - An Introduction to MultiAgent Systems - Michael Wooldridge

[15] - An Overview of Agent-Oriented Programming- Yoav Shoham (brak tutułu dla książki)

[16] – Inteligent Agents: Theory and Practice – Michael Wooldridge, Nicolas R. Jennings

[17] - Software Agent Technology: an Οverview Application to Virtual Enterprises - Chrysanthi Ε. Georgakarakou, Anastasios A. Economides

[A] - An *agent* is a computer system that is *situated* in some *environment,*

and that is capable of *autonomous action* in this environment in order

to meet its design objectives.

[B] - An agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments.