1) Opis programowania agantowego

**1.1) 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.

* + 1. **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 problemów rozwiązywanych przez program. Ten fakt nadaje programowi cechę dostosowywania się do języka danego problemu. Oznacza to rozpatrywanie problemów 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. [?]

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]

* + 1. **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. [4] 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. [5][6]

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. [5]

|  |  |
| --- | --- |
| Cecha | Warianty |
| 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 również 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. Innymi cechami agenta mogą być między innymi:

* Mobilność (ang. mobility) – zdolność agenta do przemieszczania się po przez siec komputerową.
* Uczynność (ang. benevolance) – agenci 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. [5]
* 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. [8]

Na podstawie cech i zadań jakie mogą wykonywać agent można je podzielić między innymi według poniższej topologii: [8]

* Współpracujący agenci - główną cechą tego typu agentów są zdolności społeczne. Wiele agentów współpracuje by osiągnąć jeden cel.
* Agenci interfejsu - agenci ci wspierają użytkownika w interakcji z jedną lub wieloma aplikacjami po przez uczenie się od użytkownika lub innych agentów.
* Mobilni agenci - za wykorzystaniem sieci komputerowej ci agenci potrafią przemieszczać się w celu wykonywania zadań zleconych przez ich użytkownika.
* Agencji informacyjny - celem tych agentów jest manipulowanie lub kolekcjonowanie informacji pochodzących z wielu źródeł.
* Reaktywni agenci - prosty typ agenta reagujący na stany środowiska według schematu bodziec-reakcja.
* Hybrydowi agenci - hybryda różnych typów, stworzona w celu maksymalizacji ich silnych stron.
* Mądrzy agenci - agenci cechujący się autonomią, umiejętnościami kooperacji i zdolnością do zdobywania wiedzy.

Podziału agentów można również dokonać ze względy na ich architekturę: [8]

* Agenci bazujący na logice – ich decyzje podejmowane są na drodze dedukcji.
* Agenci reaktywni – ich decyzje stanowią formę słownika której to kluczem jest jakiś stan środowiska a wartością podejmowana akacja.
* Agenci przekonanie – pragnienie – intencja (ang. belief-desire-intention, BDI) –ten typ agenta podejmuje decyzje na podstawie manipulacji struktur danych reprezentujących trzy cechy: przekonania, pragnienia oraz intencje.
* Agenci oparci o architekturę warstwową – każda z ich decyzji jest oparta o warstwy które rozpatrują obecny stan środowiska na różnych poziomach abstrakcji.

Dla paradygmatu programowania agentowego powstało wiele narzędzi oraz języków które wspierają tworzenie systemów agentowych. Wiele z nich jak na przykład szkielety aplikacji (ang. framework) opierają się o języki takie jak Java czy C++, natomiast języki agentowe stanowią całkowicie nie zależne implementacje paradygmatu programowania agentowego.[8] Przykład programu wykorzystującego jeden z języków agentowych został przedstawiony na listingu 2.

PROGRAM "cleaning"

CAPABILITIES{

{ pos(P) } Goto(R) { NOT pos(P) , pos(R) },

{ pos(P) AND dirty(R) } Vacuum(R) { NOT dirty(R) },

{ pos(P1) AND box(P1)} Movebox(P1,P2) { NOT pos(P1), NOT box(P1), pos(P2), box(P2)},

{TRUE} IsClean() {clean()},

{TRUE} Transported() {transport()}

}

BELIEFBASE{

dirty(room1).

dest(room1).

box(room2).

pos(room3).

}

GOALBASE{ clean(), transport() }

PLANBASE{ }

PG-RULES{

clean() <- dirty(Room) |

{ Goto(Room);

Vacuum(Room);

if not dirty(R) then IsClean()

},

transport() <- box(Room) AND dest(Dest) |

{ Goto(Room);

Movebox(Room,Dest);

if box(Dest) then Transported()

}

}

PR-RULES{}

Listing 2. Przykład agenta zajmującego się czyszczeniem pokoju napisana w języku 3APL

Problematyka związana zagadnieniem języków i narzędzi agentowych została szerzej opisana w rozdziale 1.3 niemniejszego opracowania.

* + 1. **Porównanie programowania obiektowego i agentowego**

Pozornie paradygmat obiektowy nie różni się znacznie od agentowego. Z punktu widzenia inżynierii w obu przypadkach program składa się z elementów posiadających własne stany, zachowanie oraz zdolność do komunikowania się miedzy sobą. Pomimo tych podobieństw koncepcyjnie między obiektami i agentami istnieją ogromne różnice.

Pierwsza różnica dotyczy stanów i zachowań obiektów i agentów. Agent jest jednostką obdarzoną autonomią z czego wynikam, że żadna inna jednostka nie powinna posiadać bezpośredniego, ani też pośredniego dostępu do jej stanów. Zmiana stanu agenta może nastąpić tylko poprzez zmiany w jego środowisku. Ta sama zasada dotyczy zachowania agenta. Żadna inna jednostka nie powinna być w stanie wywołać bezpośrednio akcji agenta. Innymi słowy agent A nie może zmusić agenta B do działania, może za to wysłać żądanie do agenta B, który z kolei w zależności do swoich celów może je wykonać lub nie. Stanowi to przeciwieństwo podejścia obiektowego. Obiekty mogą chronić i kontrolować swój stan dzięki hermetyzacji jednak nie jest to w żaden sposób wymagane i jest jedynie wolą programisty. Sytuacja jest podobna w przypadku zachowań obiektów, na przykład jeśli w języku Java metoda lub funkcja zostanie opatrzona słowem kluczowym public każdy obiekt może ją wywołać bez żadnej kontroli.[5]

Kolejną różnicą jest sposób w jaki należy postrzegać stany agenta określone przez Yoav Shoham mianem stanów mentalnych. Stany obiektów nie są z góry określone, najczęściej są to dane wykorzystywane przez obiekty. Oznacza to że obiekty różnych klas mogą mieć stany wewnętrzne rożnego typu. Agenci posiada jedynie stany określonych typów na przykład przekonania czy wybory mające wpływ na ich zachowanie. [6]

Programowanie agentowe uściśla zachowania agentów poprzez nadawanie im takich cech jak pro-aktywność, reaktywność, czy autonomię, a co za tym idzie każdy agent musi posiadać co najmniej jeden wątek pozwalający mu na działanie niezależnie od reszty systemu. Programowanie obiektowe natomiast nie wymaga od obiektów by posiadały jakiekolwiek wyżej wspomniane cechy, a więc oznacza to brak konieczności posiadania przez nie wątków.[5]

Pomimo przedstawionych różnic oba paradygmaty mają wspólny cel jakim jest dekompozycja problemów rozwiązywanych przez program. Wspomniany cel realizują jednak na różnych poziomach abstrakcji. Koncepcyjnie agent w stosunku do obiektu jest na wyższym poziomie abstrakcji. Agent jako pojedyncza jednostka w programie reprezentuje jakiś cel lub role, na przykład agent sprzątający pomieszczeni, natomiast pojedynczy obiekt stanowi najczęściej jedynie niewielki element większej całości której celem może być również sprzątanie pomieszczenia.

Opisane porównanie został podsumowane w tabeli 2.

|  |  |  |
| --- | --- | --- |
|  | Programowanie obiektowe | Programowanie agentowe |
| Jednostka | obiekt | agent |
| Orientacja | obiekty | cele i role |
| Parametry definiujące stan jednostki | niezdefiniowane | przekonania ,intencje, wiedza, zobowiązania i inne |
| Sposoby komunikacji | przekazywanie komunikatów, wywoływanie metod/funkcji | przekazywanie komunikatów, wywoływanie metod/funkcji |
| Rodzaje wiadomości | niezdefiniowane | Oferta, informacja, prośba, obietnica i inne |

Tabela 2. Porównanie programowania obiektowego i agentowego.

**1.2) Systemy wieloagentowe**

Oprogramowanie składające się z jednego agenta jest ograniczone w swoim działaniu ponieważ pojedynczy agent powinien zajmować się tylko jednym zadaniem. W przypadku złożonych problemów, które wymagają wykonywania wielu różnych czynności konieczne jest wykorzystanie większej liczby wyspecjalizowanych agentów którzy współpracowali by ze sobą. System oparty o takie założenia nazywany jest systemem wieloagentowym (ang. Multi-Agent System, MAS). Możliwościami systemów agentowych są: [9]

* rozwiązywanie problemów zbyt dużych dla jednego zcentralizowanego agenta,
* komunikacja z wieloma systemami,
* rozwiązywanie problemów posiadających rozproszoną naturę,
* rozwiązywanie problemów związanych z wieloma rozproszonymi źródłami informacji,
* możliwość zwiększenia szybkości systemu,
* uproszczenie architektury systemu.

Ważną cechą agentów wchodzących w skład systemu wieloagentowego jest zdolność do interakcji a co za tym idzie komunikacji między sobą. Bez wspomnianych zdolności agenci nie byli by w stanie współpracować i osiągnąć zamierzonego dla systemu celu. Spośród wielu możliwości najbardziej znaczącymi sposobami komunikacji są protokoły komunikacyjne oraz utworzone do tego celu języki komunikacyjne agentów (ang. Agent Communication Languages, ACL). Oba typy posiadają różne zastosowanie, protokoły są najlepszym rozwiązaniem dla aplikacji biznesowych natomiast języki sprawują się najlepiej dla systemów wykorzystujących tożsamych agentów.[8] Przykładem języka komunikacyjnego może być oparty o wiadomości KQML(ang. Knowledge Query and Manipulation Language). Język KQML definiuje [5]

**1.3) Rozwój programowania agentowego**

Pojęcie programowania agentowego zostało ukute przez Yoav Shoham w 1990 roku[3], jednak sam termin agenta jest dużo starszy. Pierwsze koncepcje agenta pojawiły się już w połowie lat 50-tych za sprawą John McCarthy oraz Oliver G. Selfridge’a. Jeden z badaczy Hyacinth S. Nwana w 1996 roku podzielił rozwój technologii agentowej na dwa pasma. Pierwsze z nich rozpoczęło się w 1977 roku gdy Carl Hewittís stworzył matematyczny model nazwany modelem aktora wprowadzający koncepcje samowystarczalnego i interaktywnego obiektu. Pasmo to wywodzi się głównie z badań nad rozproszoną sztuczna inteligencją (ang. distributed artificial intelligence) i koncentruje się miedzy innymi nad takimi zagadnieniami jak interakcja i komunikacja miedzy agentami, dekompozycja i podziałem zadań, koordynacja i kooperacją, rozwiazywaniem konfliktów po przez negocjacje. Są to typowe zagadnienia dla systemów składających się z wielu agentów. Pasmo drugie rozpoczęło się około 1990 roku i kładzie nacisk na prace nad szerokim zakresem typów agentów. [9] [10]

Wraz z rozwojem programowania agentowego zaczęły pojawiać się narzędzia dla programistów oraz standardy wspierające tworzenie systemów agentowych. Jedną z organizacji zajmująca się standaryzacją technologii agentowych była FIPA (The Foundation for Intelligent Physical Agents) powstała w 1996 roku. FIPA działała do 2005 roku kiedy to połączyła się z IEEE (ang. Institute of Electrical and Electronics Engineers) globalną instytucją zajmującą się standaryzacją technologii informatycznych. Wiele narzędzi używających standardów FIPA jest przygotowana dla obiektowego języka Java, są to JADE (ang. Java Agent DEvelopment Framework), JACK Intelligent Agents oraz JIAC (ang. Java-based Intelligent Agent Componentware). Projekty tych narzędzi są aktywne do dziś. Dla skryptowego języka python utworzone zostało narzędzie SPADE (ang. Smart Python multi-Agent Development Environment) [11][12]

[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://www.msci.memphis.edu/~franklin/AgentProg.html

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

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

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

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

[9] - Software Agents: A review - Shaw Green, Leon Hurst ,Brenda Nangle, Dr. Pádraig Cunningham, Fergal Somers, Dr. Richard Evans

//1.3

[9] - Software Agents: An Overview - Hyacinth S. Nwana

[10] - Software Agents - Jeffrey M. Bradshaw

[11] - <http://www.fipa.org/>

[12] - http://en.wikipedia.org/wiki/FIPA

[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.*