UNIWERSYTET ADAMA MICKIEWICZA

Paweł Kaczmarek
nr albumu: d299053

Multimodalna komunikacja agentów programowych z użytkownikiem

Praca magisterska na kierunku:
WYDZIAŁ MATEMATYKI I INFORMATYKI UAM
Promotor:
Prof. dr hab. Zygmunt Vetulani

Poznań 2005

Streszczenie

Praca z zakresu interakcji człowieka z komputerowym systemem wieloagentowym. Praca dotyczy zagadnień interakcji pomiędzy agentem programowym a użytkownikiem z wykorzystaniem różnorodnych kanałów komunikacji jak HTTP/HTML, e-mail, WAP, itp. Nacisk położony został na stworzenie inteligentnego interfejsu umożliwiającego skuteczną interakcję pomiędzy programem agentowym a różnorodnymi urządzeniami końcowymi.

Słowa kluczowe

multimodalna interakcja, agent osobisty, system wieloagentowy, system wspomagania podróży

Spis treści

Wprowadzenie

Idea agentów programowych[1] narodziła się w latach osiemdziesiątych, ale prawdziwy rozkwit przeżyła w roku 1994, kiedy to pojawiło się kilkanaście kluczowych publikacji o tematyce agentowej. W pracy „Intelligent agents: the new revolution in software" (Warner & Guilfoyle), autor sugeruje powstanie gotowego oprogramowania do roku 2000. Patty Maes w klasycznym tekście „Agents that reduce work and information overload" (Maes 1994) równie optymistycznie podchodzi do tematu agentów personalnych. Agent taki, „zaglądając przez ramię" użytkownikowi, uczyłby się jego upodobań, nawyków - pomagałby mu w wykonywaniu codziennych, rutynowych czynności związanych z obsługą komputera osobistego. Z kolei Wooldridge i Jennings opublikowali artykuł (Wooldridge & Jennings 1995) na temat użycia agentów jako metody programowania przewyższającej poziomem abstrakcji programowanie obiektowe. Zgodnie z myślą zawartą w tym artykule, każdy agent, byłby wysoce niezależną jednostką programową, posiadającą swój cel działania. Agenci działaliby w kolektywie, udostępniając sobie nawzajem usługi. Ponadto, rola agentów rozszerzyłaby się do spoiwa pomiędzy spadkowymi, wzajemnie niekompatybilnymi technologiami.

Jednakże, Nwana i Ndumu zaliczający się do grona sceptyków w tej dziedzinie, w publikacji z roku 1999, pod tytułem „A Perspective on Software Agents Research" (Nwana & Ndumu 1999), słusznie krytykują powszechnie panujący optymizm, zderzając ideę z rzeczywistością. W tekście tym, przedstawiony jest rzeczywisty stopień rozwoju omawianej technologii skonfrontowany z przewidywaniami wyżej wymienionych optymistów. Autorzy artykułu doszli do wniosku, iż wiele z konceptów agentów programowych zostało gruntownie przestudiowanych, jednakże rozwój technologii w wielu przypadkach pozostawał w tyle. Można wymienić kilkanaście projektów (projekty FIPA[2] w szczególności), gdzie kluczową rolę odgrywali agenci, które jednak nie wyszły z wczesnej fazy projektowania. Z pewnością czynnikiem ograniczającym jest brak odpowiedniego oprogramowania, które wspierałoby rozwój tego typu przedsięwzięć. Pozostaje zatem wybór: czekać na rozwój technologii w celu stworzenia systemów całkowicie zgodnych z ideą lub iść za radą Nwana i Ndumu, i użyć już istniejących narzędzi do implementacji realistycznych systemów. Użycie dobrze ugruntowanej technologii wydaje się być bardziej trafne w przypadku aplikacji typowo przemysłowych. Opcja druga, powinna pomóc w szybszym i bardziej rzeczowym rozwoju technologii agentowej.

W pracy tej rozważam możliwości interakcji programu agentowego z użytkownikiem poprzez różnorodne media jak HTTP/HTML, WAP, etc. Konkretna realizacja teoretycznej myśli znalazła odzwierciedlenie w implementacji klienckiej strony agentowego Systemu Wspomagania Podróży (Gordon & Paprzycki 2005). Projekt ten zakłada istnienie jednego Agenta Osobistego (ang. Personal Agent) na jednego użytkownika. Użytkownik chcąc skorzystać z usług oferowanych przez system musi mieć więc możliwość interakcji z tym agentem.

Rozdział pierwszy zawiera opis omawianego systemu. W rozdziale drugim przedstawiłem problematykę komunikacji typu agent i nie-agent. Treść rozdziału trzeciego obejmuje kilka możliwych podejść do rozwiązania problemu przedstawionego w rozdziale drugim. W rozdziale czwartym i piątym opisuję użyte technologie i metody użyte podczas faz projektowania i implementacji podsystemu a w rozdziale szóstym wymieniam w szczególe kluczowe zagadnienia samej implementacji.

[1] Definicja agenta programowego - rozdział 5.2.

[2] FIPA - Foundation for Intelligent Physical Agents, http://www.fipa.org/

ROZDZIAŁ 1

System Wspomagania Podróży

1.1. Wspomaganie podróży a technologie internetowe

Współczesny, skomputeryzowany podróżny, chcąc zaplanować podróż, zamówić bilety lotnicze czy dowiedzieć się o rodzajach kuchni w Madrycie, wie, że nie stoi przed łatwym zadaniem. Ma do dyspozycji którąś z wyszukiwarek internetowych (na przykład Google). Jednakże prędkość, z jaką rośnie dzisiejsza sieć WWW, powoduje, iż tego typu mechanizmy wyszukiwania zaczynają zawodzić, gdyż zalewają użytkownika niezliczoną ilością odsyłaczy do nieinteresujących go stron. Poniżej, jako wstęp do opisu Systemu Wspomagania Podróży przedstawiłem trendy i technologie, które mają szansę zmienić obecny stan rzeczy. Wśród nich najważniejsze to: Semantyczny Internet oraz agenci programowi.

Semantyczny Internet

Nieskuteczność w odzyskiwaniu pożądanych informacji wynika z cech podstawowego narzędzia do prezentacji zawartości Internetu - języka HTML. Język ten pozwala na opis struktury dokumentu oraz sposobu jego wizualizacji. Oznacza to, że za jego pomocą można poinstruować przeglądarkę sieciową by napis „cukier" umieścił obok napisu „jest słodki" - nie ma natomiast łatwego sposobu przekazania treści tego komunikatu przetwarzającej go maszynie.

Semantyczny Internet[1] jest projektem prowadzonym pod kierunkiem konsorcjum W3C[2]. Zamierzeniem przedsięwzięcia jest stworzenie uniwersalnego sposobu wymiany informacji, poprzez dodanie zrozumiałej dla maszyn, semantycznej informacji do dokumentów sieciowych. Narzędziem do specyfikacji struktury dokumentów Semantycznego Internetu jest XML oraz XML Schema, natomiast RDF (Resource Description Framework)[3] oraz RDF Schema[4] są językami służącymi do semantycznego opisu dokumentów w sieci. Rysunek **??** przedstawia strukturę Semantycznego Internetu:

- URI (Uniform Resource Identifier) - najniższa warstwa reprezentuje fundament projektu. URI zapewnia statyczne mapowanie zasobów, tworząc ich „nawigowalną" przestrzeń[5].

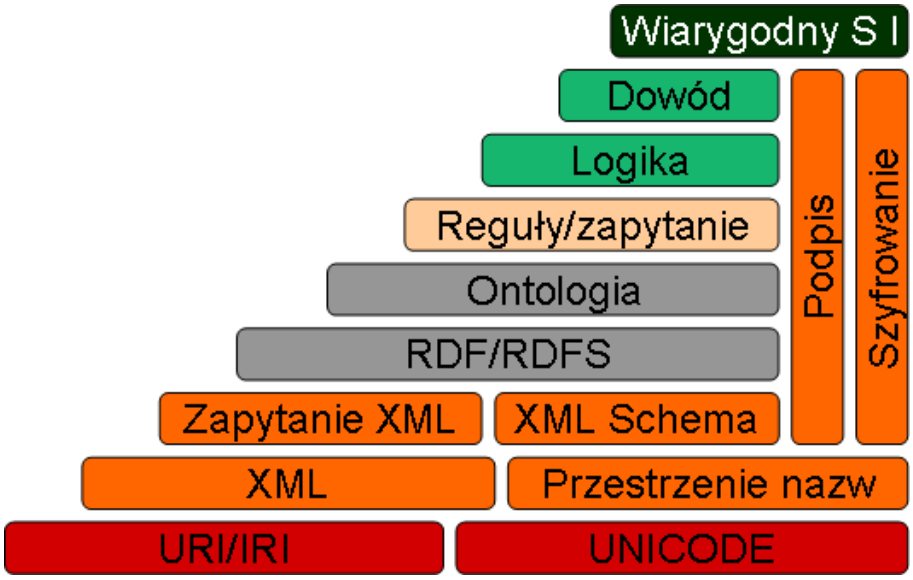
- RDF, RDF Scheme - przemysł informatyczny jest zgodny, że XML jest odpowiednim środkiem reprezentacji danych (dla ludzi i maszyn), gdyż dostarcza on syntaktycznej struktury dla opisu tych danych. Jednakże, przy użyciu XML istnieje kilka sposobów reprezentacji tych samych danych i z tego powodu język ten okazał się zbyt arbitralny by wesprzeć tak rozległą integrację danych, jaką jest Semantyczny Internet. Wizja Semantycznego Internetu (według pomysłodawcy Tima Bernersa-Lee'a) proponuje język Resource Description Framework[6] do opisu maszynowo przetwarzalnej informacji. RDF

[1] Semantic Web, http://www.w3.org/2001/sw/
[2] konsorcium W3C, http://www.w3.org/
[3] szerszy opis technologii w rozdziale 1.2.3
[4] szerszy opis technologii w rozdziale 1.2.4
[5] patrz rozdział 1.2.2
[6] patrz rozdział 1.2.3

to ustandaryzowana forma opisywania i zapytywania zasobów internetowych (tekst, audio, wideo, pliki) zapewniająca syntaktyczną interoperacyjność[7].

- Ontologie - Ponad interoperacyjnością syntaktyczną musi zachodzi interoperacyjność semantyczna, tak by współdziałające w Semantycznym Internecie aplikacje mogły wymieniać dane (informacje). Rozwiązaniem tego zagadnienia są sformalizowane modele domen (ontologie), które definiują użyte termy oraz relacje między nimi. Ontologie używają terminów jak klasa, podklasa oraz własność.

- Dowód, zaufanie i bezpieczeństwo - W przypadku globalnej bazy danych jaką ma się stać Semantyczny Internet, powstają problemy zaufania i bezpieczeństwa. Nie wszystkie zasoby będą powszechnie dostępne i równie niezawodne. Gdyby udało się do zwracanego wyniku zapytania dodać dowód (warstwa Dowód), w jaki sposób uzyskano odpowiedź, pytająca aplikacja mogłaby wykonać pewne wnioskowanie (poprzez środki warstwy Logiki) i ocenić wiarygodność źródła. Ponadto, wyekstrahowane ze źródła fakty mogłyby stanowić jego atrybut by z czasem mogły być poddane ocenie rzetelności (warstwa Wiarygodnego SI). Te ostatnie warstwy struktury Semantycznego Internetu zbadano w najmniejszym stopniu i stanowią źródło wielu przeszkód technicznych.



Rysunek 1.1. Stosowa struktura Semantycznego Internetu, opracowano na podstawie http://www.w3.org/Consortium/Offices/Presentations/SemanticWeb/34.html

Semantyczny Internet jest obietnicą technologii pozwalającej na komputerowe przetwarzenie zawartości dokumentów w sieci, co pozwoli na filtrowanie, separacje oraz syntezę elementów informacji (Fensel 2003).

Zgodnie z zamierzeniem twórców technologii Semantycznego Internetu, kluczowym komponentem oprogramowania zdolnego do interpretacji zawartości tego typu dokumentów są agenci programowi[8]. Rolę agenta, jako jednostki interpretującej semantyczną treść, zauważono i podkreślono między innymi w Intelligent Systems Journal (IEEE 2001). Uważa się,

[7]Interoperacyjność (ang. interoperability) to zdolność dwóch systemów lub komponentów do wymieniania informacji i używania tej informacji. Interoperacyjność w kontekście oprogramowania oznacza jego niezależność od platformy sprzętowej.

[8]Semantic Web - Big Picture, http://www.semanticweb.org/about.html

że działający niezależnie lub w zespole, prywatny agent użytkownika, bazując na semantycznie określonej zawartości, dzięki swojej inteligencji, podoła zadaniu dostarczenia klientowi przefiltrowanej oraz spersonalizowanej informacji.

Technologia agentowa

Podróżny, poza wyszukiwarką internetową, może posłużyć się dedykowanym serwisem ułatwiającym planowanie podróży. Takie serwisy uważa się jednak za jedne z najbardziej złożonych (IEEE Intelligent Systems Journal, 2002). Jednocześnie, technologię agentów programowych postrzega się jako doskonały środek rozbijania złożonych problemów na komponenty (Jennings 2001), które w tym przypadku są agentami lub kolektywami agentów.

Stosowność użycia podejścia agentowego w tego typu projekcie została ponadto zauważona przez Hyacinth S. Nwana oraz Divine T. Nduma (Nwana & Ndumu 1999) ze względu na takie cechy agentów jak mobilność kodu i stanu, świadomość kontekstu oraz personalizację. Ta ostatnia cecha w szczególności, pozwala na realistyczne odwzorowanie prawdziwej agencji podróży oraz agentów podróży. Należy także pamiętać, iż technologia agentów programowych wywodzi się z domeny sztucznej inteligencji i dzięki takim cechom agentów jak inteligencja oraz autonomia użytkownik może mieć nadzieje na bardziej zindywidualizowaną obsługę aniżeli w takich projektach jak Expedia[9], Orbitz[10] czy Travelocity[11].

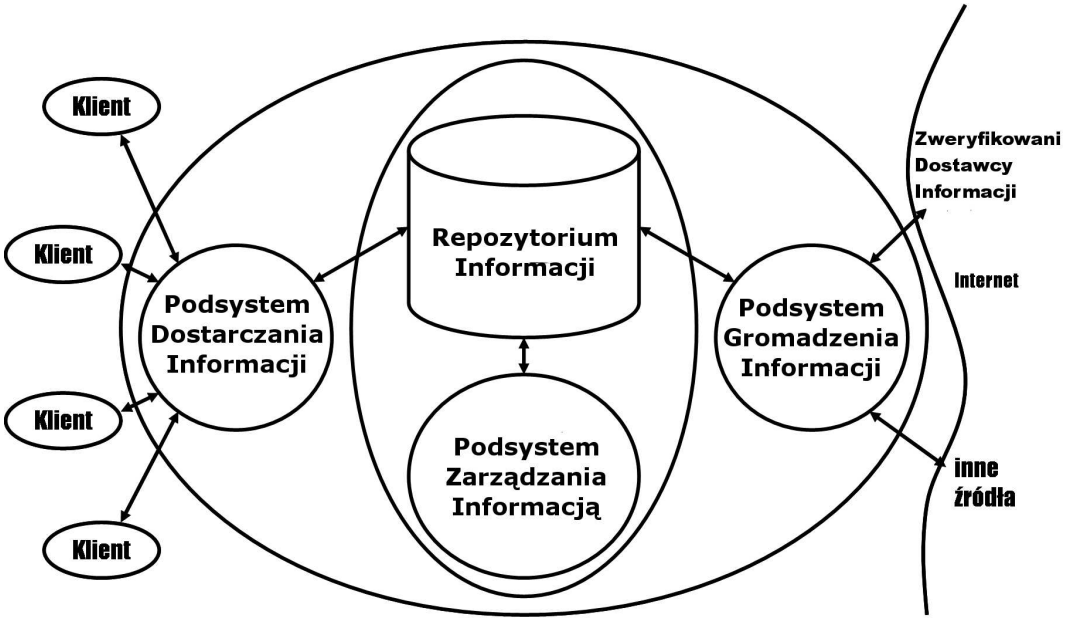
1.2. Wysokopoziomowy przegląd Systemu Wspomagania Podróży

Systemem Wspomagania Podróży (ang. Travel Support System) to projekt, który w zamierzeniu ma wykorzystywać technologię agentową oraz technologię Semantycznego Internetu. Rolą agentów będzie wyszukiwanie wartościowych, użytecznych i przede wszystkim spersonalizowanych informacji. W polu zainteresowań podróżnych są rezerwacje biletów lotniczych, godziny otwarcia muzeów, informacje na temat zwiedzanego zamku czy rodzaje kuchni w pobliskim lokalu. Budowany system musi dysponować tego typu danymi. Ponadto, dane te opisane będą przy użyciu języka RDF. Umożliwi to agentom przetwarzanie tych informacji na potrzeby wykonywanych zadań.

Prace nad Systemem Wspomagania Podróży zaczęły się w roku 2001 opublikowaniem [**?**, **?**, **?**]. Od tamtej pory, rozwój projektu trwa nieprzerwanie i w 2005 zaowocował opublikowaniem artykułu „Designing Agent Based Travel Support System" [**?**], zawierającego opis szkieletu systemu. Jego struktura opiera się na ogólnym schemacie projektu e-commerce (Galant, Jakubczyc & Paprzycki 2002) i wyróżnia trzy główne podsystemy: Podsystem Gromadzenia Informacji, Podsystem Zarządzania Informacją oraz Podsystem Dostarczania Informacji. Ogólną architekturę ilustruje rysunek 1.2. Wyróżnić można następujące podsystemy:

- **Zweryfikowani Dostawcy Informacji (ZDI)** - Pod tym pojęciem kryją się serwisy, gwarantujące:
  - rzetelne, godne zaufania dane,
  - pewien ustalony interfejs,
  - stałą dostępność w Internecie.

[9]Expedia: http://www.expedia.com/
[10]Orbitz: http://www.orbitz.com/
[11]Travelocity: http://www.travelocity.com/



Rysunek 1.2. Ogólna architektura systemu

opracowano na podstawie [**?**]

Grupy tego typu dostawców to:

- dostawcy oferujący zawartość w standardowym formacie, jak Rich Site Summary (RSS),
- statyczne strony, gdzie zachodzi potrzeba periodycznego ekstrahowania danych, na przykład za pomocą odpowiednio zaprogramowanych agentów.

Istnieje kilkanaście serwisów zaliczających się do kategorii ZDI, których przykłady można znaleźć w [**?**, **?**].

- **Inne źródła** - W przeciwieństwie do ZDI, są to źródła danych zawartych w Internecie o niepotwierdzonej wiarygodności.
- **Podsystem Gromadzenia Informacji (PGI)** - Funkcją tego komponentu jest przechowywanie semantycznie rozgraniczonych danych, skolekcjonowanych przez podsystem ZDI oraz inne źródła internetowe. Dane te opisane są w języku RDF. Repozytorium funkcjonuje w oparciu o system JENA[12] firmy Hewlett-Packard[13].
- **Podsystem Zarządzania Informacją (PZI)** - Komponent ten zarządza repozytorium danych. Do jego zadań należy między innymi:
  - uzupełnianie niepełnych danych (przykładowo - brakujący numer telefonu restauracji),
  - uzupełnianie danych, które się z czasem dezaktualizują (repertuar kina, godziny otwarcia muzeum).
- **Podsystem Dostarczania Informacji (PDI)**[14] - Agenci w tym podsystemie wyko-

[12] system do przetwarzania dokumentów RDF, patrz http://www.hpl.hp.com/semweb/jena.htm
[13] Hewlett-Packard, http://www.hp.com/
[14] Przedmiot mojej pracy.

rzystują dane „zaopatrzenia” w celu dostarczenia klientowi spersonalizowanej informacji.

- **Klienci** - Usługi systemu będą dostępne dla szerokiej rzeszy klientów wyposażonych w urządzenia z dostępem do Internetu: od komputera biurowego do telefonu komórkowego z usługą WAP. Klientem systemu może być także inny, spoza systemu, autonomiczny agent.

1.3. Klienci systemu

Dostępność usług omawianego systemu w dużej mierze zależy od rozmiaru puli urządzeń, za pomocą których użytkownicy będą mogli komunikować się ze swoim Agentem Personalnym. Pomimo znaczących różnic pomiędzy urządzeniami końcowymi jak telefon komórkowy, PDA (Personal Digital Assistant) czy komputer biurowy, należało ustalić zbiór minimalnych wymagań dla urządzeń chcących połączyć się z systemem. Z rozważań poczynionych w [**?**] wyłonił się następujących zbiór ustaleń.

Ze względu na fakt, iż wszelkie wymienione wyżej urządzenia mają dostęp do Internetu, głównym kanałem komunikacji jest wszechobecny protokół HTTP. Różnice pomiędzy klientami są natomiast w zawartości przesyłanych danych. Stąd klientów systemu podzielono na dwie obszerne grupy:

- *klienci nieinteraktywni*: klienci Web Services, inne systemy wspomagania podróży (OpenTravel Alliance[15]), agenci spoza omawianego systemu (Agentcities Network Services[16])
- *klienci interaktywni*[17]: klienci interpretujący HTML, WAP, czy inne języki znacznikowe.

HTTP jest przyjętym standardem wśród klientów-przeglądarek, które mają często jednak ograniczone możliwości interpretowania zawartości przesyłanej przez ten protokół. W celu dotarcia do możliwie największej grupy klientów wymagania odnośnie możliwości interpretacyjnych przeglądarki klienta nie mogą być wygórowane. Z tego względu, odrzucono wszelkie technologie zależne od platformy, urządzania czy przeglądarki takie jak animacja Flash czy aplet Java. Ponadto, ze względu na bezpieczeństwo, niektórzy klienci systemu mogą mieć administracyjne ograniczenia na instalowanie „obcego” oprogramowania, co skłoniło projektantów systemu do wykluczenia JRE (Java Runtime Environment) oraz technologii pozwalającej na interpretacje XML. Z tych samych powodów należało wykluczyć użycie *ciasteczek*.

Z powyższych rozważań wynika, iż klient (programowy) systemu musi mieć do dyspozycji jednokierunkowe połączenia Internetowe używając protokołu HTTP oraz być w stanie interpretować pewnego rodzaju język znacznikowy (HTML, WML, XHTML, etc.)

Należy jednak podkreślić, że zaimplementowane rozwiązanie z łatwością daje się zaadoptować na potężniejszych urządzeniach, na których Agent Personalny może egzystować. System został rozbity na komponenty, stwarzając tym samym możliwość używania jego poszczególnych „kawałków” przez bardziej wyszukanych klientów.

[15] OpenTravel Alliance, http://www.opentravel.org
[16] Agentcities Network Services, http://www.agentcities.net
[17] Ze względu na zakres tej pracy, zajmę się klientami interaktywnymi.

1.4. Repozytorium danych systemu

1.4.1. Pojęcie ontologii

Ontologią, w dziedzinie filozofii, nazywa się (według ??jakas ksiazka nie sieciowa??):

- gałąź metafizyki zajmującą się naturą oraz relacjami między bytami, lub,
- pewną teorię określającą naturę lub rodzaje bytów.

W przemyśle IT, kilkanaście lat temu wyłoniło się nowe znaczenie terminu „ontologia", jako produktu inżynierii zawierającego szczegółowe słownictwo opisujące rzeczywistość (część rzeczywistości) oraz pewien zbiór bezpośrednich założeń opisujących zamierzone znaczenie tego słownictwa - innymi słowy jest to „specyfikacja konceptualizacji" (Gruber 1993). Konceptualizacja jest abstraktem, uproszczonym spojrzeniem na świat, który chcemy reprezentować.

W procesie opisywania pewnej części świata (domeny), określamy pewne byty (konkretne rzeczy z tej domeny), ich własności oraz relacje między nimi. W terminach ontologi mamy:

- klasy (ang. classes) - pewne domeny zainteresowania
- instancje (ang. instances) - konkretne byty
- związki pomiędzy bytami (ang. relationships)
- własności (ang. properties) oraz ich wartości
- funkcje oraz procesy związane z bytami
- reguły i granice dotyczące bytów

W kontekście przemysłu informatycznego, ze względu na potrzebę przetwarzania maszynowego, wymaga się mocno sformalizowanej reprezentacji ontologii, zazwyczaj przy użyciu języka logicznego lub reprezentującego wiedzę. Język naturalny dostarcza jedynie etykiety (ang. labels), informujące człowieka o znaczeniu poszczególnych jednostek ontologii.

1.4.2. URI

URI jest standardem internetowym[18], który za pomocą zgodnego ze standardem łańcucha znaków umożliwia łatwą identyfikacje zasobów w sieci. URI składa się ze schematu URI i dwukropka, po którym następuje część zależna od schematu. Przykłady schematów to: „http", „ftp", „urn", etc. Syntaktyka części następującej po dwukropku zależna jest od specyfikacji związanej z danym schematem, która musi jednak spełniać pewne założenia zgodne ze specyfikacją URI.

URI umożliwiający lokalizację zasobu określa się mianem URL (Uniform Resource Locator) tego zasobu. Przykładowo URL:

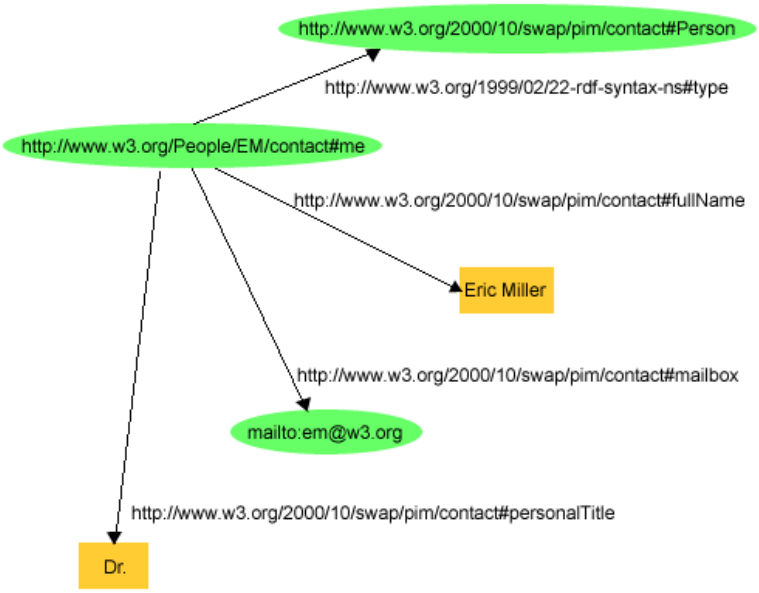
```
http://adres/opis.txt
```

wskazuje, że dokument o nazwie „opis.txt" jest dostępny na serwerze o nazwie „adres" poprzez protokół HTTP.

URN (Uniform Resource Name) jest podzbiorem URI i odnosi się do łańcuchów znaków identyfikujących zasoby poprzez nazwę. Nazwa ta powinna spełniać wymóg globalnej jednoznaczności oraz trwałości, nawet w przypadku niedostępności zasobu. Zasób może być opisany w przestrzeni nazw „`urn`" lub w innej zarejestrowanej przestrzeni. Oto przykład:

```
adres/opis.txt#dokladnyOpis
```

[18]Specyfikacja URI, patrz:http://www.ietf.org/rfc/rfc2717.txt



Rysunek 1.3. Graf RDF opisujący osobę Eric'a Millera

opracowano na podstawie [?]

URI może być jednocześnie lokalizatorem i nazwą

```
http://adres/opis.txt#dokladnyOpis
```

1.4.3. RDF

> RDF (Resource Description Framework) jest określoną przez W3C specyfikacją modelu metadanych, zazwyczaj implementowaną w postaci aplikacji XML. Założeniem RDF jest opis zasobu za pomocą wyrażenia składającego się z trzech elementów: podmiotu, predykatu i obiektu. W RDF podmiot stanowi opisywany zasób, predykat określa jaka jego własność jest opisywana, zaś obiekt stanowi wartość tej własności. Podstawowym mechanizmem wykorzystywanym przez RDF do identyfikacji podmiotu, predykatu i obiektu jest URI[19].

Celem RDF jest umożliwienie maszynowego przetwarzania abstrakcyjnych opisów zasobów w sposób automatyczny. Stwierdzenia w RDF można ilustrować za pomocą grafów. Dla przykładu, grupę stwierdzeń:

> Osoba identyfikowana przez: http://www.w3.org/People/EM/contact#me, której na imię Eric Miller, której adres e-mail to: em@w3.org, która ma stopień doktora,

przedstawia rysunek 1.3.

RDF udostępnia składnię opartą o XML (nazywaną RDF/XML). Grupa stwierdzeń odpowiadająca rysunkowi 1.3 zapisana w formacie RDF/XML ma formę:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
```

[19] definicja za Wikipedią, patrz http://pl.wikipedia.org/wiki/RDF

```
  <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me
     ">
    <contact:fullName>Eric Miller</contact:fullName>
    <contact:mailbox rdf:resource="mailto:em@w3.org"/>
    <contact:personalTitle>Dr.</contact:personalTitle>
  </contact:Person>

</rdf:RDF>
```

zaś w notacji N3[20]:

```
@prefix : <#> .

<http://www.w3.org/People/EM/contact#me>
      a       <http://www.w3.org/2000/10/swap/pim/contact#Person> ;
      <http://www.w3.org/2000/10/swap/pim/contact#fullName>
              "Eric Miller" ;
      <http://www.w3.org/2000/10/swap/pim/contact#mailbox>
              <mailto:em@w3.org> ;
      <http://www.w3.org/2000/10/swap/pim/contact#personalTitle>
              "Dr." .
```

1.4.4. RDF Scheme

Własności (properties) opisywane przez RDF mogą być użyte jako atrybuty zasobów lub do reprezentacji związków pomiędzy zasobami. Językowi RDF brakuje jednakże mechanizmu opisu tych *własności* czy opisu związków pomiędzy *własnościami* a zasobami. Lukę tą wypełnia język RDF Scheme (RDFS) [**?**]. RDFS rozszerza semantycznie RDF poprzez zdefiniowanie klas oraz *własności*, które mogą być użyte do opisu innych klas, *własności* oraz zasobów.

Model danych opisany przy użyciu RDFS jest taki sam jak model danych stworzony przy użyciu języków obiektowych jak Java. RDFS pozwala na zdefiniowanie klasy[21], obiektu (instancji klasy) czy klasy dziedziczącej z innej klasy.

Język RDFS zawiera nowe elementy gramatyki do określania zależności na poziomie klas, wśród których najważniejsze to:

- rdfs:Resource - określający zasób
- rdfs:Property - określający *własność*
- rdfs:Class - określający klasę

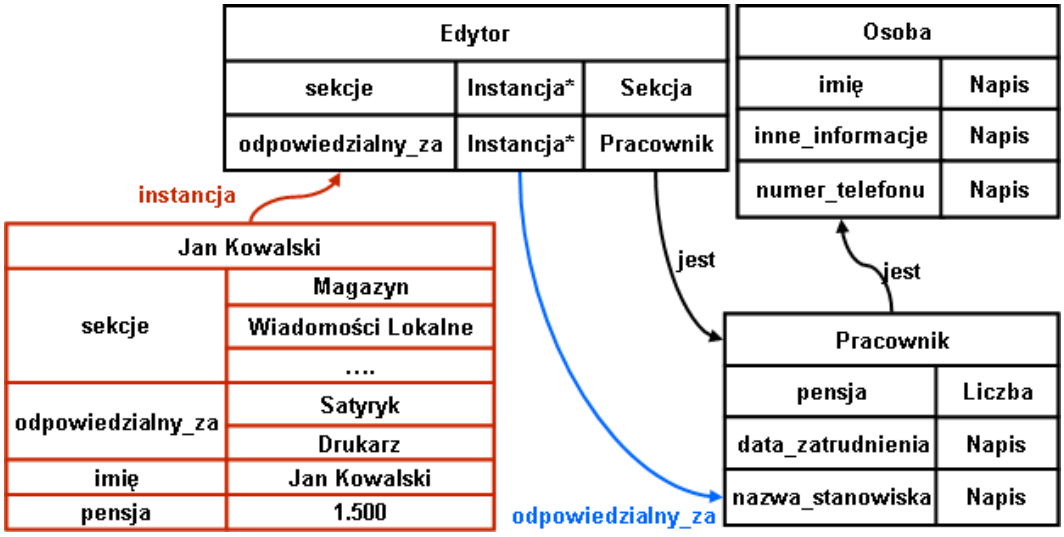
Ponadto, istnieją elementy gramatyki do wyrażania związków pomiędzy klasami i *własnościami* jak: bycie podklasą innej klasy (rdfs:subClassOf), przynależenie do domeny klasy (rdfs:domain), do ograniczenia wartości *własności* (rdfs:range). Przykład nietrywialnego modelu danych, stworzonego w języku RDFS obrazuje rysunek 1.4.

1.4.5. Ontologia systemu

Na potrzeby omawianego systemu stworzono ontologię restauracji [**?**]. Źródłem danych był projekt Chefmoz [**?**], zawierający semantycznie rozdzielone dane o ponad 260 tysiącach restauracji na całym świecie. W projekcie tym, restaurację może charakteryzować między innymi:

[20] http://www.w3.org/2000/10/swap/Primer

[21] „wzorzec" obiektów o wspólnych własnościach



Rysunek 1.4. Przykładowy model danych. Osoba, Pracownik oraz Edytor to klasy. Strzałka z etykietą „jest" oznacza rozszerzenie klas (dodanie nowych atrybutów). Strzałka z etykietą „instancja" łączy klasę z jej instancją. Strzałka z etykietą „odpowiedzialny_za" wskazuje na konkretną relację pomiędzy klasami. Przykład został wygenerowany na podstawie fikcyjnych danych zapisanych w języku RDFS przez aplikację: Protege 3.1.1 z wtyczką: Ontviz

.

- adres (podzielony na adres, dzielnice, miasto, stan, kraj i kod zip),
- kuchnia,
- miejsca parkingowe,
- akceptowane środki płatnicze.

Przykładowa instancja klasy restauracji przyjmuje następującą formę:

```
<res:Restaurant rdf:about="http://www.agentlab.net/db/restaurant-db
   #Poland_DS_Bielany_Wroclawskie_Mangia_Pizza1046496998">
    <city>Poznan</city>
    <country>Poland</country>
    <phone>+48 (61) 555 44 22 </phone>
    <state>DS</state>
    <streetAddress>ul. Czekoladowa 12</streetAddress>
    <zip>44-342</zip>
    <res:cuisine rdf:resource="http://www.agentlab.net/schemas/
       restaurant#ItalianCuisine"/>
    <res:cuisine rdf:resource="http://www.agentlab.net/schemas/
       restaurant#PizzaCuisine"/>
    <res:hours>Mon-Sat 10-21; Sun 10-20</res:hours>
    <res:locationPath>Poland/DS/Poznan</res:locationPath>
    <res:parsedHours>
            10-20|10-21|10-21|10-21|10-21|10-21|10-21
    </res:parsedHours>
    <res:title>Super Pizza</res:title>
</res:Restaurant>
```

Stworzona klasa restauracji zawiera listę własności (cech) restauracji. Każda z nich jest unikatowa i zawiera komentarz. Poniższy przykład własności „rodzaj kuchni":

```
<rdf:Property rdf:about="http://www.agentlab.net/schemas/restaurant
   #cuisine">
    <comment>
      The type of food a restaurant serves.
      We repeat this field up to three times.
    </comment>
    <domain rdf:resource=
                   "http://www.agentlab.net/schemas/restaurant#
                      Restaurant"/>
    <label>cuisine</label>
    <range rdf:resource=
                   "http://www.agentlab.net/schemas/restaurant#
                      CuisineCode"/>
</rdf:Property>
```

1.4.6. Interfejs użytkownika a ontologia systemu

Ze względu na nieustannie rosnące zasoby Internetu zaszła potrzeba by metoda semantycznego opisu tych zasobów nie naruszała już istniejących struktur. Dlatego, w przeciwieństwie do języków takich jak Java, struktury w języku RDF powstają w oparciu o technikę „bottom-up"- w pierwszej kolejności określa się własność a następnie klasę, do której ta własność należy. W przypadku ontologii opisującej restauracje umożliwia to na przykład łatwe dodanie pewnej cechy restauracji, której wcześniej nie uwzględniono, jak odległość od centrum miasta, czy ilość miejsc przy barze.

Budowany System Wspomagania Podróży uwzględnił zmienny charakter tego typu ontologii. Użytkownik będzie w ten sposób w stanie udzielić bardziej szczegółowych i co ważniejsze aktualnych instrukcji systemowi. Interfejs użytkownika uwzględnia aktualny stan ontologii (jest na jej podstawie każdorazowo generowany). Opis rozwiązania tego zagadnienia zawiera rozdział 6.4.

ROZDZIAŁ 2

Komunikacja agenta z użytkownikiem

Mając na względzie uwagi poczynione w podrozdziale 1.2, rozważę możliwości komunikacji agenta programowego z użytkownikiem. Wezmę pod uwagę następujące scenariusze:

- agent rezydujący w urządzeniu klienta,
- agent wymieniający wiadomości z użytkownikiem.

2.1. Agent rezydujący w urządzeniu klienta

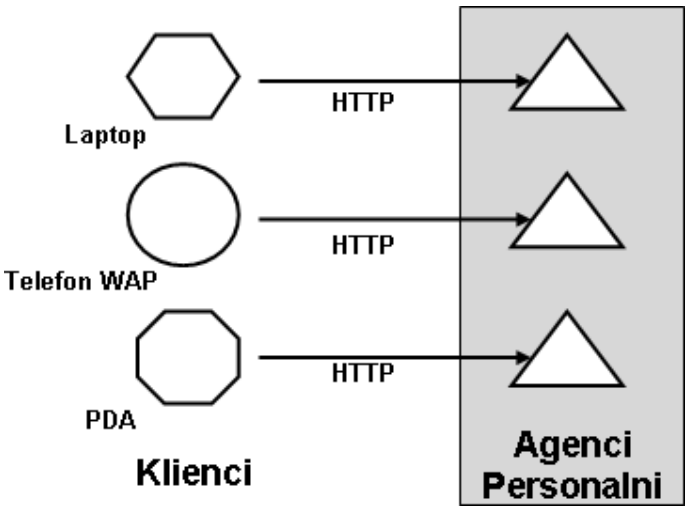
W podrozdziale 1.2 ustaliłem, iż interakcja użytkownika z omawianym agentowym Systemem Wspomagania Podróży oprze się na dobrze znanej architekturze typu „klient - serwer". Należy jednakże zauważyć, iż to rozwiązanie kłóci się z jedną z najważniejszych wytycznych dotyczących oprogramowania agentowego - agenci powinni być wszędzie. Agent powinien rezydować na urządzeniu klienta i reprezentować go w interakcjach z siecią w zastępstwie przeglądarki internetowej. W rzeczywistości jednak, agent, by rezydować na urządzeniu klienta, wymaga obecności platformy agentowej (agencji). Założenie obecności tak wyspecjalizowanego oprogramowania na przykład na telefonie komórkowym jest na dzień dzisiejszy nierealistyczne. Wysiłki FIPA [**?**] dopiero niedawno skupiły się na stworzeniu specyfikacji wszechobecnej platformy, która byłaby w stanie przyjąć zgodnego ze specyfikacją FIPA agenta a powstałe dotychczas platformy agentowe jak: Aglets, Grasshopper, Voyager, COncordia nie stosują się do tego standardu. Ponadto istnieje jedynie kilka projektów, które umożliwiłyby wkroczenie agenta na mobilne urządzenie. Najbardziej wyróżniający się projekt, LEAP (Lightweight Extensible Agent Platform) [**?**], stracił w lipcu 2002 roku dotacje Unii Europejskiej i przyszłość tego projektu jest niepewna. Powyższe fakty ilustrują jedynie przeszkody jakie występują podczas dystrybucji agentów na mobilne urządzenia.

Niemniej, zaimplementowałem agenta rezydującego na telefonie komórkowym z możliwością obsługi Java Micro Edition (J2ME). Możliwości obliczeniowe telefonów komórkowych i w konsekwencji możliwości J2ME nie pozwoliły jednakże na umieszczenie w telefonie wielowątkowego Agenta Personalnego. Zaimplementowany przeze mnie agent[1] korzysta z wersji JADE[2] (Java Agent DEvelopment Framework) [**?**] przeznaczonej na urządzenia przenośne: JADE-LEAP[3].

2.2. Bezpośrednia wymiana wiadomości pomiędzy klientem a agentem

Podążając w myśl poprzednich ustaleń, omawiany agentowy System Wspomagania Podróży ma udostępniać swoje usługi w konwencji klient-serwer tak, aby nie zmuszać użytkownika do instalacji dodatkowego oprogramowania czy dodatkowych wtyczek. Zważywszy, że rezydowanie agenta na urządzeniu jest obecnie utrudnione należy rozważyć możliwość bezpośredniej wymiany komunikatów pomiędzy agentem a użytkownikiem (rysunek 2.1).

[1] patrz rozdział 6.4
[2] patrz rozdział 4.3
[3] patrz rozdział 4.3.1



Rysunek 2.1. Bezpośrednia komunikacja klient-agent

opracowano na podstawie [**?**]

Obecnym opiniotwórcą i rzeczywistym liderem w dziedzinie standardów technologii agentowej jest FIPA[4]. Organizacja ta wydała szereg standardów i specyfikacji dotyczących komunikacji międzyplatformowej i międzyagentowej - w tym standard wiadomości ACL(Agent Communication Language) [**?**]. Jedną z bardziej rozpowszechnionych platform agentowych (zastosowana na przykład w sieci Agentcities Network Services [**?**]), zgodnych ze specyfikacją FIPA jest JADE. Niestety bliska zgodność platformy JADE ze standardami FIPA nie zapewnia wsparcia dla komunikacji typu agent i nie-agent, ze względu na fakt, iż w trakcie rozwoju standardów typu ACL, skupiano się wyłącznie na komunikacji międzyagentowej i międzyplatformowej. W efekcie, pomimo, że agenci platformy JADE wymieniają wiadomości ACL poprzez HTTP (jest to specjalny dialekt) klient omawianego systemu wspomagania podróżnych, nie jest w stanie komunikować się z agentem programowym bez dodatkowego oprogramowania, co przeczy założeniom z rozdziału 1.2. Sytuację komplikuje fakt, iż agenci rezydują w środowisku platformy agentowej, która odgradza ich od środowiska sieciowego. Dlatego też nie ma sposobu adresowania agentów poprzez numer IP i port. Bezpośrednia komunikacja urządzenia klienta z agentem poprzez HTTP jest zatem niemożliwa.

Agent jako serwer

Należy w tym miejscu rozważyć jeszcze możliwość uczynienia agenta serwerem. Warto jednakże zauważyć, iż jednostka programowa rezydująca w pewnym określonym, dobrze znanym miejscu w sieci, odpowiadająca na zapytania (jak proces-demon), nie mieści się już w żadnej z definicji agenta[5]. Ponadto, ze względu na wcześniejsze założenia odnośnie systemu, agent taki musiałby posiadać wsparcie dla każdego ze znanych języków znacznikowych (HTML, WML, XHTML, etc.) co z uwagi na rozmiar z pewnością ograniczyłoby mobilność tego agenta.

Z powyższych rozważań wynika, że wszystkie wcześniej zaproponowane, intuicyjne rozwiązania problemu komunikacji pomiędzy agentem a nie-agentem (agent na urządzeniu, agent komunikujący się z klientem, agent jako serwer) umniejszają potencjał i rolę użytej tech-

[4] patrz rozdział 4.2
[5] definicja - rozdział 5.2

nologii agentowej negując tym samym celowość jej użycia. W proponowanym rozwiązaniu (opublikowanym w [**?**]), skupiłem się na jak najpełniejszym wykorzystaniu zalet agentów programowych, zrzucając brzemię problemu komunikacji na barki innych technologii.

ROZDZIAŁ 3

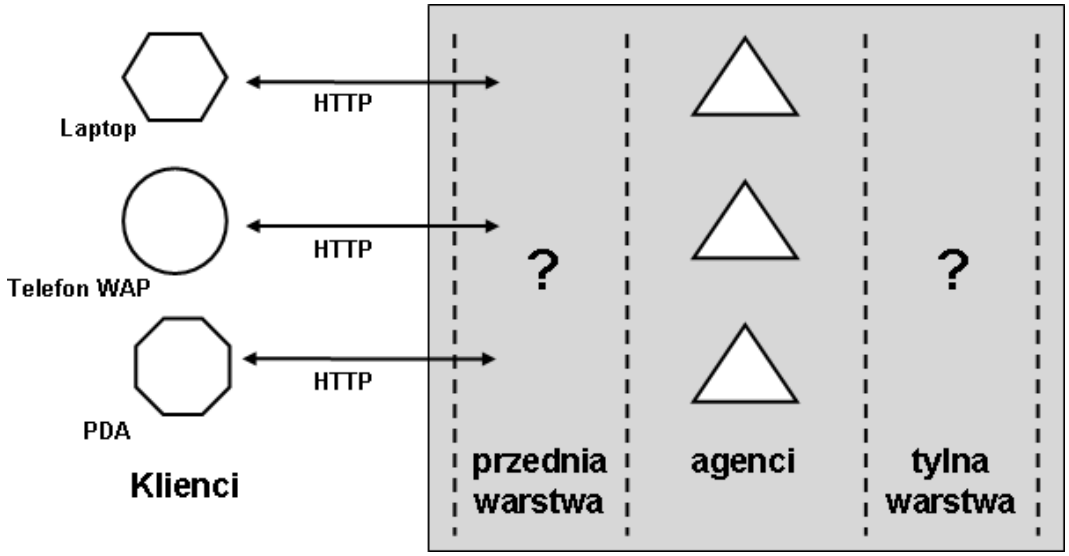
Rozwiązanie problemu komunikacji pomiędzy programem agentowym a użytkownikiem

3.1. Świat agentowy a świat nie-agentowy

Od pewnego czasu systemy wielowarstwowe stanowią rozwiązanie wielu problemów aplikacji przemysłowych. W architekturze tej rozdziela się wyspecjalizowane komponenty jak: interfejs użytkownika, warstwę logiki biznesowej czy warstwę zarządzającą repozytorium danych. Każdy z komponentów jest dobrze skapsułkowany i nie musi być „świadom" operacji wykonywanych w innych warstwach. W ten sposób możliwe jest równoczesne rozwijanie wielu warstw, ułatwiona jest obsługa błędów, co zwiększa niezawodność oprogramowania. Infrastruktura agentów może służyć jako middleware (oprogramowanie pośredniczące) [**?**], pozostawiając innym komponentom systemu obsługę komunikacji z użytkownikiem oraz interakcję z urządzeniami końcowymi (ang. back-end resources). Definicja middleware[1]:

> „Middleware (oprogramowanie pośredniczące) jest to rodzaj oprogramowania służący do komunikacji aplikacji użytkownika z bazami danych, lub innymi serwerami/usługami. Wykorzystanie middleware ułatwia tworzenie aplikacji, gdyż uniezależnia ją od rodzaju bazy danych. Obecnie, wobec rozwoju modelu aplikacji wielowarstwowych, middleware zyskuje na znaczeniu.
>
> Oprogramowanie middleware ma za sobą długą historię, zaś jego początki sięgają technologii mainframe. Rodzaje middleware obejmują takie kategorie jak:
>
> - monitory transakcyjne (Tuxedo, MQSeries)
> - procesory zapytań (rozmaitego rodzaju aplikacje integrujące interfejsy różnych komponentów aplikacyjnych, np. Enterprise Integration Portals, w których zapytania zadawane przez aplikację, np. CRM, są przetwarzane na zapytania obsługiwane przez inną aplikację, np. system bilingowy)
> - (ang. Data Driven Routing) pozwalający na budowę rozproszonej bazy danych w oparciu o bazy nie obsługujące rozproszonych transakcji
> - sterowniki baz danych:
>   - ODBC
>   - OLEDB
>   - JDBC (Java Database Connectivity)
>
> Obecnie termin middleware zaczyna mieć nieco szersze znaczenie, co wiąże się z wprowadzeniem oprogramowania pośredniczącego w transakcjach internetowych (CORBA, COM+) i innego oprogramowania łączącego klienta z usługodawcą (agenci)".

[1] definicja: http://pl.wikipedia.org/wiki/Middleware



Rysunek 3.1. Agenci jako middleware; klienci rezydują w Internecie; pozostałe warstwy pozostają po stronie serwera

opracowano na podstawie [**?**]

Rolę agentów jako middlware[2] uważa się za jedną z podstawowych, która wyrosła z filozofii tworzenia złożonego i rozproszonego oprogramowania. Ponadto, System Wspomagania Podróży, w którym wykorzysta się cechy agentów jak autonomia, mobilność czy inteligencja z pewnością w dużo większym stopniu będzie w stanie imitować prawdziwą agencję turystyczną czy też prawdziwego agenta podróży. Dlatego centralnym agentem jest Agent Personalny - jeden dla każdego użytkownika. On reprezentuje klientowi usługi agencji, zbliżając jego rolę do roli prawdziwego agenta podróży. Pozostali agenci zostali stworzeni jako komponenty rozproszonego systemu zapewniające określoną funkcjonalność (personalizacja, zapytywanie repozytorium danych), która wspomaga Agenta Personalnego w wykonaniu żądań użytkownika. Należy pamiętać jednak, iż Agent Personalny jest nadzorcą systemu i to pomiędzy nim a użytkownikiem zachodzi interakcja. To on jest także odpowiedzialny za wykonywanie poszczególnych aspektów zapytania klienta poprzez zarządzanie poszczególnymi agentami funkcjonalnymi. Rozwiązanie, gdzie agent personalny znajduje się po stronie serwera oraz jest częścią middleware posiada kilka znaczących zalet:

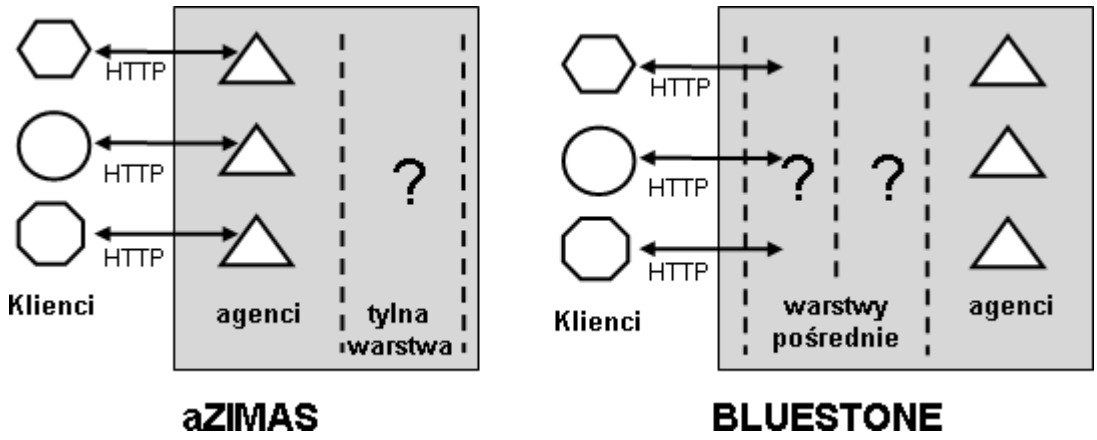
- Agent nie jest zależny od przepustowości łącza oraz systemu użytkownika.
- Dopóki agent utrzymuje kontakt z serwerem może korzystać ze swojej mobilności.
- Możliwa jest bezpośrednia interakcja agenta z bazami danych lub innymi repozytoriami (na przykład poprzez interfejs Java Database Connectivity).
- Ze względu na oddzielenie agentów od różnorodności „języków" używanych przez klientów, agenci ci, mogą być budowani w bardziej jednorodny sposób.
- Brak administracyjnych nakładów związanych z obsługą komunikacji z klientem w agencie upraszcza projektowanie jego funkcjonalności.

[2]oprogramowanie pośredniczące

Pytanie jakie się nasuwa, to jak zrealizować ideę Agenta Personalnego, komunikującego się bezpośrednio ze swoim użytkownikiem, wiedząc, że obecny stan technologiczny utrudnia lub uniemożliwia rezydowanie agenta na mobilnych urządzeniach. Odpowiedź to agent, który stanowi część middleware i „udaje" bezpośrednią komunikację z klientem. Proponowane rozwiązanie wspiera przeźroczystą integracje platformy agentowej (na przykład JADE) z konwencjonalnymi aplikacjami typu enterprise[3]. Aplikacja ma do dyspozycji zalety agentów jak rozproszenie, mobilność czy inteligencja, zaś problem komunikacji agenta i nie-agenta spada na barki aplikacji konwencjonalnej.

3.2. Agenci jako middlware

Agentów jako middlware użyto między innymi w systemie aZIMAS [**?**] oraz Bluestone[4] [**?**]. Ilustracje obu rozwiązań zawiera rysunek 3.2. Projekt aZIMAS użył agentów w warstwie prezentacji oraz w warstwie zajmującej się logiką aplikacji. Dokładniej, zapytanie HTTP klienta odbierane jest poprzez serwer sieciowy Apache, który ekstrahuje parametry z tego zapytania i przekazuje je odpowiedniemu agentowi. Rozwiązanie to cechuje wysoka wydajność ze względu na komunikację agent-użytkownik oraz wysokie zespolenie ze strukturami serwera. W celu użycia tego projektu, należy stworzyć zupełnie nową platformę agentową, co godzi w jedną z głównych wytycznych współczesnych systemów wieloagentowych, by łączyć współczesne platformy zgodne ze specyfikacją FIPA.



Rysunek 3.2. Porównanie rozwiązań aZimas oraz Bluestone

opracowano na podstawie [**?**]

Odmiennego podejścia użyli programiści rozwijający projekt Blustone [**?**]. Agenci stanowią w nim tylną warstwę (back-end tier) i są traktowani jako dostawcy usług. Usługi te udostępniane są poprzez ustalony, ogólny interfejs. Istnieje warstwa środkowa, pośrednicząca (Universal Listening Framework)[**?**], która nasłuchuje na różnorodnych kanałach komunikacyjnych (HTTP, SMTP), której zadaniem jest dostarczenie zapytania klienta do warstwy agentowej i odwrotnie, odpowiedzi agenta do klienta. Należy zauważyć, iż agenci odpowiadający na żądania klientów poprzez pewien ogólny, udostępniony interfejs tracą atrakcyjne cechy agentów jak mobilność czy autonomiczność.

[3] duże aplikacje przemysłowe
[4] firmy Hewlett-Packard

Podczas prac nad implementacją omawianego systemu wspomagania planowania podróży, wzięto pod uwagę [**?**] wyżej wymienione podejścia, jednakże położono szczególny nacisk by końcowy produkt wykorzystywał możliwie najpełniej zalety agentów, maskując jednocześnie ich wady. Konieczne okazało się stworzenie warstwy pośredniczącej, która oddziela Agenta Personalnego i resztę systemu od niskiego poziomu klienckich protokołów jak HTTP czy WAP. Pośrednik ten, jest konstrukcją modułów nasłuchujących. Każdemu klienckiemu protokołowi odpowiada jeden moduł, który potrafi odebrać żądanie klienta a następnie „przetłumaczyć" je na żądanie zrozumiałe dla Agenta Personalnego.

Konkretne moduły nasłuchujące obarczone są minimalną funkcjonalnością, wystarczającą do przyjęcia zgłoszenia użytkownika, przetłumaczenia zapytania na postać zrozumiałą dla Agenta Personalnego oraz odebrania odpowiedzi od systemu i przekazania jej klientowi. Moduły te nie przetrzymują stanu klienta (z wyjątkiem identyfikatora sesji). W ten sposób proces interakcji klienta obsługiwany jest w systemie przez Agenta Personalnego, dając mu możliwość działania jako rzeczywisty reprezentant użytkownika.

Należy w tym miejscu rozważyć problem przesyłania wiadomości do rozłączonego klienta. Zarówno telefon z WAP, jak i standardowa przeglądarka internetowa bez dodatkowego oprogramowania (jak aplet Java) nie są w stanie nasłuchiwać odpowiedzi. Problem ten jest dość powszechny wśród internetowych aplikacji, w których wykonanie zapytanie może być czasochłonne i użytkownik powinien być zawiadomiony o rezultacie zapytania. Używając protokołu HTTP wyróżnia się dwa podejścia do tego problemu [**?**]:

- *Klient blokuje, nie zrywając połączenia, do momentu pojawienia się odpowiedzi od systemu.* Rozwiązanie to jest dalekie od optymalnego, albowiem klient jest blokowany, co uniemożliwia mu dalszą interakcję z systemem. W niektórych wypadkach jednak może to być jedyne realistyczne rozwiązanie.

- *Klient zapytuje o gotową odpowiedź w regularnych odstępach czasu.* W podejściu tym zakłada się istnienie specjalnego modułu nasłuchującego, który zwracałby gotowe odpowiedzi lub informował o ich braku. Bardziej zaawansowani klienci mogliby w tle (w innym wątku) sprawdzać dostępność odpowiedzi. Wadą rozwiązania jest znaczne obciążenie serwera regularnymi zapytywaniami modułu nasłuchującego.

Ze względu na możliwości obecnych przeglądarek internetowych (w szczególności tych na telefonach komórkowych) zdecydowałem się na użycie pierwszego rozwiązania. W przyszłości rozwiązanie to może być sprzężone i rozwinięte za pomocą technologii jak AJAX[5] [**?**].

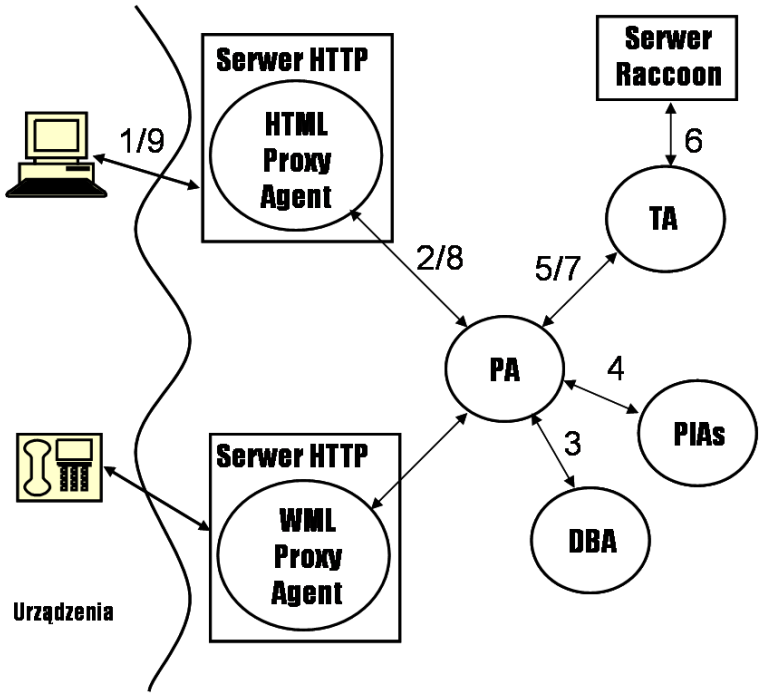
3.3. Przepływ danych

Architektura PDI (Podsystem Dostarczania Informacji) przedstawiona jest na rysunku 3.3. Funkcjonalność całego podsystemu jest rozbita pomiędzy kilku agentów, którzy działają „pod dowództwem" Agenta Personalnego, który jest centralnym elementem tego podsystemu. Wszelkie interakcje pomiędzy komponentami agentowymi odbywają się poprzez wymianę wiadomości ACL[6].

Użytkownik poprzez urządzenie z dostępem do Internetu, przekazuje zlecenie do *Agenta Proxy* (ang. Proxy Agent - PrA), który rezyduje za serwerem HTTP. Agent ten tłumaczy otrzymane zlecenie na formę zrozumiałą dla *Agenta Personalnego* (ang. Personal Agent - PA)

[5] ang. Asynchronous JavaScript Technology and XML

[6] więcej na ten temat - rozdział 7



Rysunek 3.3. Podsystem Dostarczania Informacji: PA – Agent Personalny, DBA – Agent „Bazodanowy", PIA – Struktura Personalizacyjna, PrA – Agent Proxy, TA – Agent Transformujący. Numery przy strzałkach oznaczają kolejność przepływu wiadomości w systemie.

opracowano na podstawie [**?**]

i przesyła je we wiadomości ACL. PA przekazuje zlecenie do Agentowej Struktury Personalizacyjnej (ang. Personalization Infrastructure Agents - PIAs) oraz do *Agenta „Bazodanowego"* (ang. DataBase Agent - DBA).

PIAs są strukturą agentów zajmujących się personalizacją. Dane potrzebne do personalizacji, uzyskiwane są z procesu interakcji użytkownika z systemem: magazynowane są wszelkie zlecenia klienta, odpowiedzi systemu oraz reakcje użytkownika na te odpowiedzi. Dane te wykorzystywane są do budowania profilu użytkownika z wykorzystaniem między innymi technik typu data-mining [7].

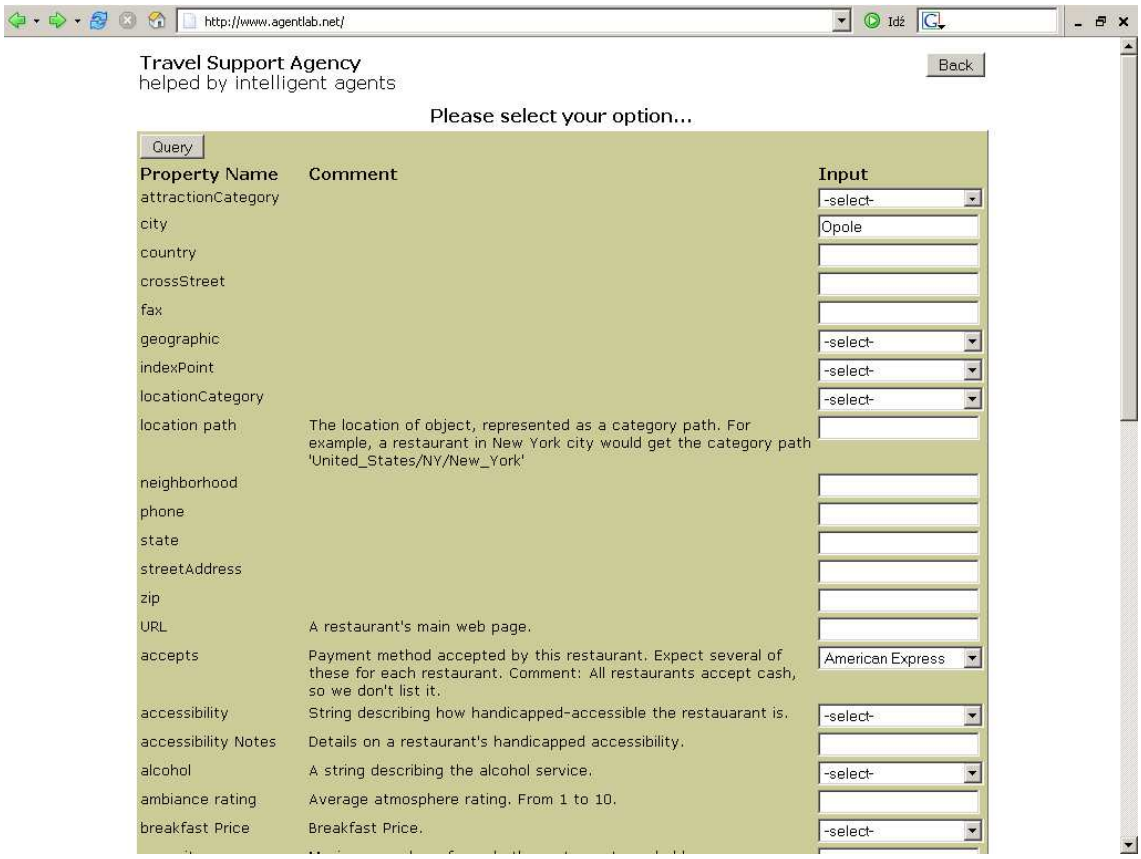
DBA stanowi interfejs systemu do repozytorium danych zarządzanego przez system JENA. Dokładniej, DBA tłumaczy otrzymane zapytanie na język RDQL (RDF Data Query Language) [**?**]. Następnie, używając JENA zapytuje źródło danych, uzyskując zbiór trójek RDF (w formacie RDF/XML) pasujących do zapytania. Rezultaty działania przekazywane są (we wiadomości ACL) PIAs w celu spersonalizowania odpowiedzi trafiającej do PA. W celu uzyskania czytelnej dla specyficznego protokołu formy odpowiedzi, PA przesyła otrzymane rezultaty do *Agenta Transformatora* (ang. Transformation Agent - TA), który wspiera się funkcjonalnością dostarczaną przez serwer Raccoon[8].

Uwzględniając zakres tej pracy, skupiłem się na implementacji PA, DBA, PrA oraz TA. Agent DBA operuje na przykładowej bazie danych restauracji oraz ontologii RDF opisujących strukturę tej bazy danych.

[7] data-mining
[8] patrz rozdział 4.4

3.3.1. Przykład

Założę, że użytkownik zalogował się, ustanowił połączenie ze swoim PA, zapytujący formularz HTML został poprawnie wygenerowany i wypełniony (rysunek 3.4) oraz użytkownik nacisnął przycisk wysyłający dane formularza (metoda GET protokołu HTTP).



Rysunek 3.4. Wygenerowany i wypełniony formularz zapytujący bazę danych restauracji. Wypełnione pola do *cuisine*, *city* oraz *accepts*.

Serwer odbiera żądanie i uruchamia wątek serwera wyznaczony do obsługi zlecenia klienta. Dla ustalenia uwagi niech zapytanie klienta będzie następujące:

1. city = Opole
2. cuisine = Italian
3. accepts = American Express Card

Przy uwzględnieniu ukrytych pól formularza nazwa użytkownika (dla ustalenia uwagi: użytkownikX), protokół kliencki (dla ustalenia uwagi:html_media), nazwa operacji (dla ustalenia uwagi: `get_data_operation`) odpowiada zapytaniu HTTP[9]:

```
http://www.agentlab.net/queryForm?
city=Opole
```

[9]W praktyce pola formularza identyfikowane są przez URI, wiec w zastępstwie parametru o nazwie „cuisine" jest parametr o nazwie „http://www.agentlab.net/schemas/restaurant#cuisine". W przykładzie zastosowałem uproszczenie dla poprawienia czytelności przykładu.

```
&cuisine=Italian
&accepts=AmericanExpressCard
&op_name=get_data_operation
&user_name=użytkownikX
&media_type=html_media
```

Z metadanych zapytania HTTP (querystring) ekstrahowane są parametry zapytania klienta (w postaci klucz-wartość) jak również separowane są dane przekazane z ukrytych pól formularza. Dane zapytania przekazywane są PrA, po czym wątek zasypia.

PrA, otrzymawszy dane zlecenia, formułuje odpowiednią wiadomość ACL i przekazuje ją PA użytkownika. PA odbiera wiadomość i na podstawie nazwy operacji wybiera kolejne kroki postępowania, co w przypadku ustalonej nazwy operacji oznacza przekazanie wiadomości ACL zawierającej parametry zapytania do DBA. Agent ten odbiera wiadomość i na podstawie otrzymanych parametrów (w postaci klucz-wartość) formułuje zapytanie w języku RDQL[10], które w omawianym przypadku ma następującą postać:

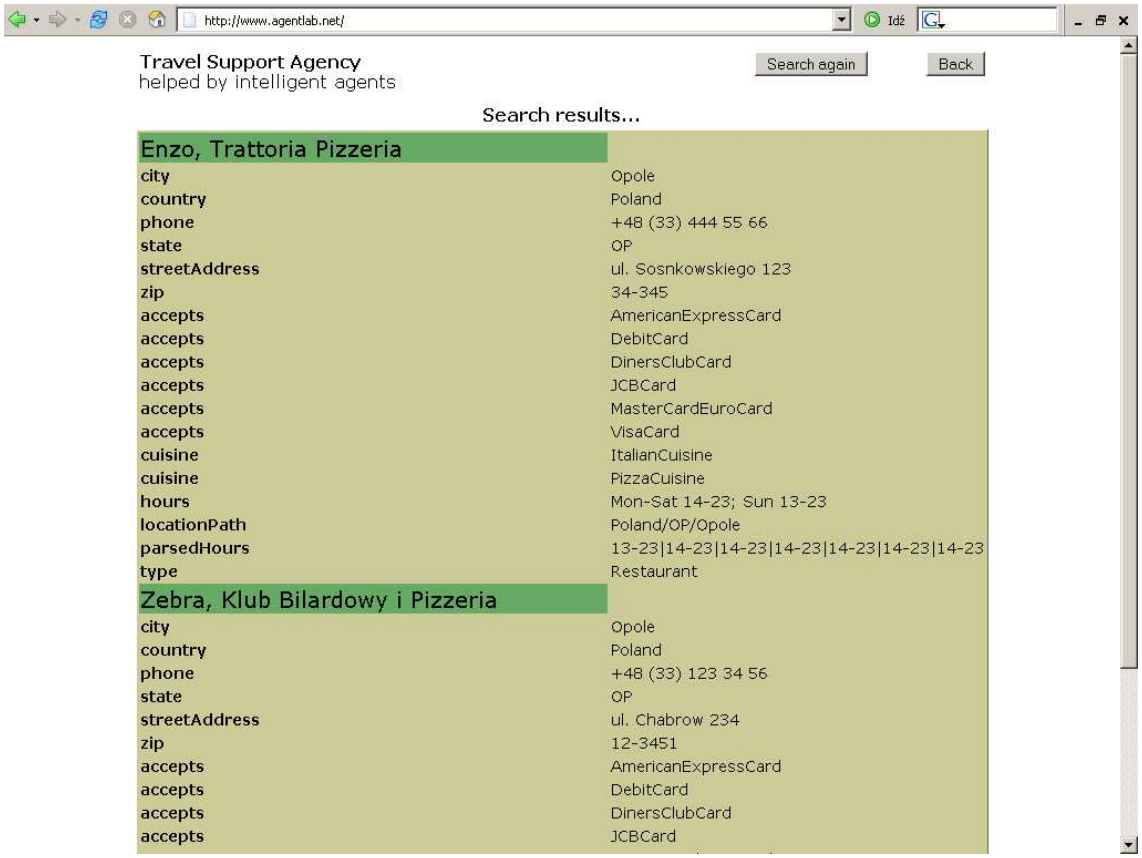
```
SELECT ?res WHERE
(?res,
   <http://www.agentlab.net/schemas/location#city>,
                                                'Opole'),
(?res,
   <http://www.agentlab.net/schemas/restaurant#cuisine>,
    <http://www.agentlab.net/schemas/restaurant#ItalianCuisine>),
(?res,
   <http://www.agentlab.net/schemas/restaurant#accepts>,
    <http://www.agentlab.net/schemas/money#AmericanExpressCard>)
```

Utworzone zapytanie, przy użyciu JENA, pozwala na odpytanie repozytorium danych systemu, co w efekcie daje następujący, przykładowy zbiór trójek RDF, które w postaci N3[11] mogą przyjąć postać:

```
:Poland_OP_Opole_Enzo__Trattoria_Pizzeria1044823399
  a       res:Restaurant;
  loc:city "Opole";
  loc:country "Poland";
  loc:#phone "+48 (33) 444 55 66 ";
  loc:#state "OP";
  loc:#streetAddress "ul. Sosnkowskiego 123 ";
  loc:#zip "34-345";
  res:accepts mon: VisaCard,
        mon: MasterCardEuroCard,
        mon: DebitCard,
        mon: JCBCard,
        mon: DinersClubCard,
        mon: AmericanExpressCard;
  res:cuisine res: ItalianCuisine,
        res: PizzaCuisine;
  res: hours "Mon-Sat 14-23; Sun 13-23";
  res: locationPath "Poland/OP/Opole";
  res: parsedHours "13-23|14-23|14-23|14-23|14-23|14-23|14-23";
  res: title "Extra Pizzeria".
```

Wyniki działania DBA, „zamknięte” we wiadomości ACL, wracają do PA, który kieruje je do dalszej transformacji, do TA. Agent ten na podstawie zawartości oraz na podstawie

[10] Język obsługiwany przez JENA do zapytywania modelu danych
[11] http://www.w3.org/2000/10/swap/Primer



Rysunek 3.5. Przykładowy wynik działania systemu

parametrów znajdujących się w otrzymanej wiadomości (protokół kliencki, nazwa operacji) wykonuje odpowiednie zapytanie do serwera Raccoon[12], otrzymując w efekcie dokument w formacie HTML wizualizujący odzyskaną przez system informacje (rysunek 3.5). Zakodowana we wiadomości ACL odpowiedź TA wraca do PA, który przekazuje ją do PrA. PrA przekazuje wiadomość od systemu odpowiedniemu wątkowi serwera, który, po przekazaniu jej przeglądarce klienta, ginie.

[12]szczegóły-rozdział 6.4

ROZDZIAŁ 4

Technologie

4.1. Wstęp

System Wspomagania Podróży budowany jest na bazie technologii agentowej, która:

- jest relatywnie młoda i mało ugruntowana (niedawno opublikowane standardy), co powoduje częsty brak interoperacyjności,
- nie posiada wielu narzędzi wspierających programistę,
- nie wspiera integracji z technologiami spadkowymi.

Budowanie aplikacji z użyciem tak „niestabilnych fundamentów" może jednakże mieć za zadanie zbadanie kresów możliwości czy też wytrzymałości tych fundamentów. Wiedza na temat najnowszych osiągnięć w dziedzinie technologii agentowej jest zatem niezwykle istotna. Niniejszy rozdział jest wglądem w istniejące standardy, technologie i narzędzia użyte w implementacji klienckiej strony tego systemu.

Zacząłem od opisu istotnej dla projektu części dorobku fundacji FIPA, która współcześnie odgrywa rolę wyroczni w kontekście standaryzujących specyfikacji dotyczących tej technologii. Następny podrozdział jest analizą, w pełni zgodnej ze specyfikacjami FIPA platformy agentowej, JADE (architektura, możliwości, agenci). Kolejny podrozdział „Publikowanie dokumentów RDF/XML" ma służyć jako wprowadzenie do metod transformacji treści w formacie RDF/XML na pewien, używany przez klienta język znacznikowy jak HTML, WML, XHTML, XUL, etc. Zacząłem od opisu platformy Cocoon, będącej najbardziej popularnym i rozpowszechnionym narzędziem służącym publikacji zawartości dokumentów XML, aby na zasadzie analogi opisać mniej znane narzędzie Raccoon, służące między innymi publikacji dokumentów RDF/XML. Ostatni rozdział, wprowadza w technologię XUL oraz Thinlet, których użyłem do dynamicznego generowania interfejsu użytkownika na urządzeniach jak telefon komórkowy czy PDA.

4.2. FIPA

FIPA (The Foundation for Intelligent Physical Agents) jest przedsięwzięciem tworzącym komputerowe standardy oprogramowania dla heterogenicznych, oddziałujących na siebie agentów programowych oraz systemów wieloagentowych. FIPA powstała w Szwajcarii, w roku 1996 jako organizacja niedochodowa, stawiając sobie ambitny cel zdefiniowania pełnego zbioru standardów, który miał służyć zarówno jako wyznacznik do implementacji platform agentowych jak i specyfikować, w jaki sposób agenci mają się ze sobą komunikować i oddziaływać. Do członków tej organizacji zalicza się kilkanaście instytucji naukowych oraz firm włączając: Hewlett Packard, IBM, British Telecom, Sun Microsystems, Fujitsu oraz wiele innych. W celu zintensyfikowania wysiłków dążących do integracji technologii agentowych i nie-agentowych, 8. lipca 2005. roku pieczę nad instytucją przejął komitet IEEE[1].

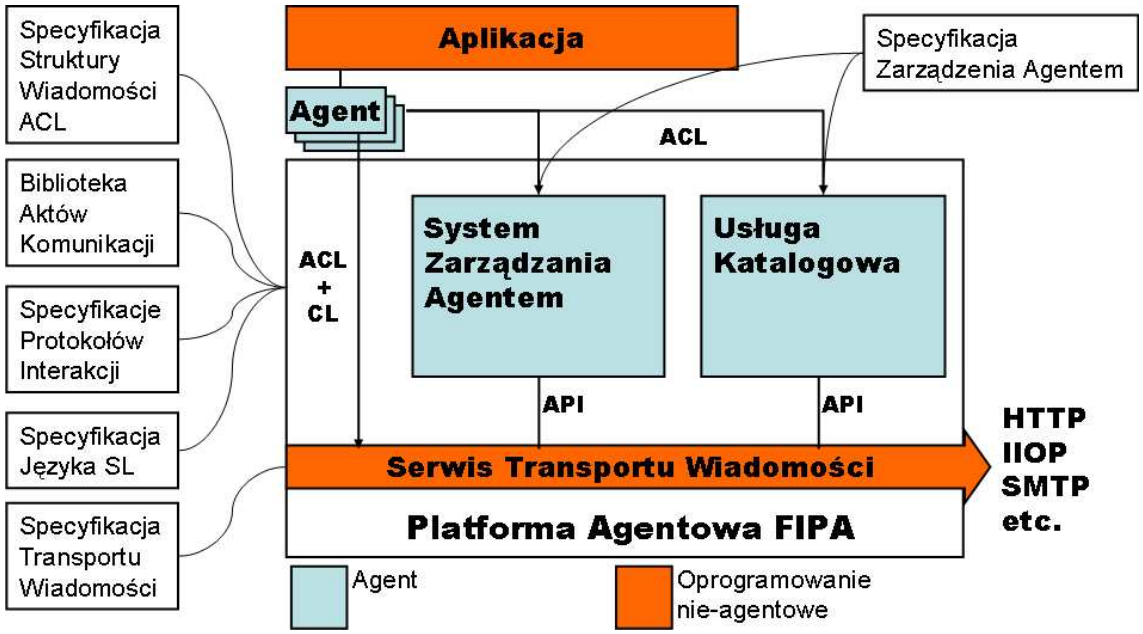
[1] IEEE Computer Society: http://www.computer.org/portal/site/ieeecs/

Pomimo, że przedsięwzięcie, wbrew oczekiwaniom, nigdy nie otrzymało komercyjnego wsparcia, FIPA uważana jest obecnie za rzeczywistego „przywódcę" w dziedzinie tworzenia standardów dla technologii agentowej, czego dowodem jest stale powiększający się zbiór komercyjnych i niekomercyjnych profesjonalnych projektów implementujących specyfikacje FIPA jak[2]:

- JADE
- FIPA-OS
- Grasshopper
- ZEUS
- April Agent Platform

4.2.1. Schemat platformy agentowej FIPA

Platforma agentowa w specyfikacji FIPA ma schemat ilustrowany przez rysunek 4.1. FIPA w ramach platformy wymienia trzy podstawowe domeny:



Rysunek 4.1. Platforma agentowa FIPA

opracowano na podstawie [?]

- Zarządzenie Agentem (ang. Agent Management - AM) - do której należą: System Zarządzania Agentem (ang. Agent Management System - AMS) oraz Usługa Katalogowa[3](ang Directory Facilitator - DF)

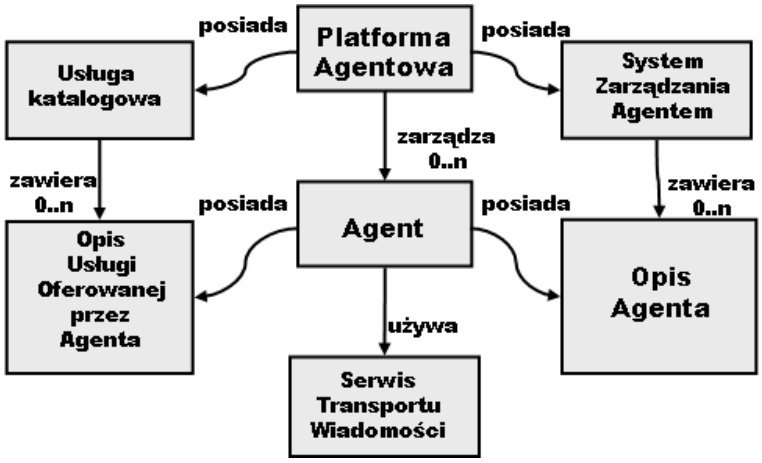
[2]http://www.fipa.org/resources/livesystems.html
[3]żółte strony usługi oferowane przez agentów

- Transport Wiadomości (ang. Message Transport MT) - do której należy: Kanał Komunikacji Agentowej (ang. Agent Communication Channel - ACC), Protokół Transportu Wiadomości (ang. Message Transport Protocol - MTP) oraz Serwis Transportu Wiadomości (ang. Message Transport Service - MTS)

- Język Komunikacji Agentowej (ang. Agent Communication Language - ACL) - do której należą: Akty Komunikacji (ang. Communication Acts - CAs), Protokoły Interakcji (ang. Interaction Protocols - IPs), Języki Treści Wiadomości (ang. Content Languages - CLs)

4.2.2. Zarządzenie Agentem

Każda platforma agentowa FIPA w ramach specyfikacji Zarządzenia Agentem posiada System Zarządzania Agentem oraz Usługa Katalogową (ilustracja zależności - rysunek 4.2).



Rysunek 4.2. Agent Management - zależności

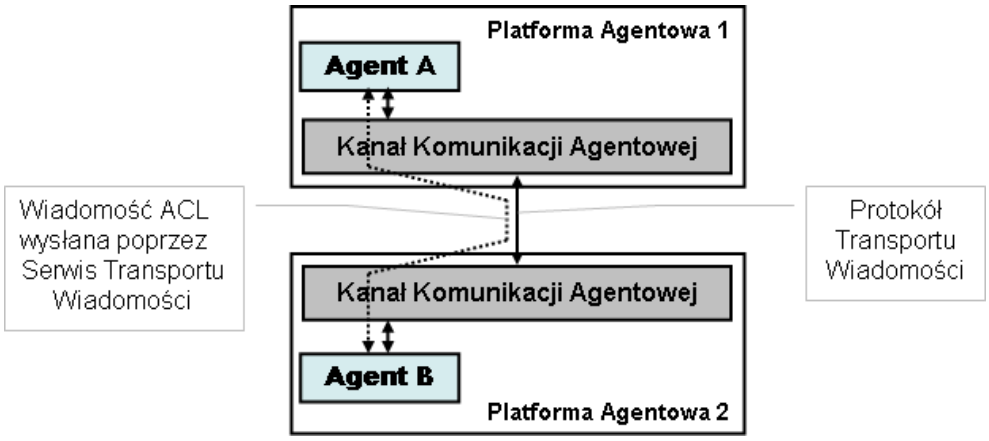
opracowano na podstawie [?]

- System Zarządzania Agentem - odpowiedzialny za operacje platformy agentowej takie jak:
  - tworzenie/usuwanie agentów,
  - rejestracja/wyszukiwanie agentów,
  - migracja agentów.

- Usługa Katalogowa - usługa dostarczająca funkcjonalność „żółtych stron" - każdy agent, chcąc upublicznić swoje usługi, powinien znaleźć odpowiedni „katalog" i poprosić o rejestrację swojego opisu agenta. DF wykonuje ponadto:
  - wyrejestrowanie usługi,
  - wyszukiwanie usługi,
  - modyfikację usługi.

4.2.3. Transport Wiadomości

Kanał Komunikacji Agentowej jest jednostką platformy, która dostarcza podstawową metodę kontaktu dwóch agentów będących na jednej lub różnych platformach. Wymiana wiadomości odbywa się na dwóch poziomach (rysunek 4.3):



Rysunek 4.3. FIPA: Transport Wiadomości - wymiana wiadomości pomiędzy platformami

opracowano na podstawie [**?**]

- Protokół Transportu Wiadomości - odpowiedzialny za fizyczne dostarczenie wiadomości pomiędzy dwoma Kanałami Komunikacji Agentowej.
- Serwis Transportu Wiadomości - dostarczany przez Kanał Komunikacji Agentowej. Agent na danej platformie, podłączony jest do Serwisu Transportu Wiadomości oferowanego przez tą platformę. Serwis Transportu Wiadomości odpowiedzialny jest za dostarczenie wiadomości ACL zarówno pomiędzy agentami w ramach tej samej platformy jak i różnych platform.

Struktura wiadomości ACL

Wszelkie wiadomości przesyłane są w formacie ACL, który ma strukturę składającą się z następujących pól:

- *performative* - typ aktu komunikacji
- *sender* - nadawca
- *receiver* - odbiorca
- *reply-to* - odbiorca odpowiedzi
- *content* - treść (zawartość) wiadomości
- *language* - język użyty do wyrażenia treści
- *encoding* - kodowanie znaków, użyte w treści
- *ontology* - ontologiczny kontekst treści odpowiedzi

- *protocol* - protokół do jakiego należy wiadomość
- *conversation-id* - identyfikator wiadomości
- *reply-with* - odpowiedź z tym wyrażeniem
- *in-reply-to* - akcja, dla której ta wiadomość stanowi odpowiedź
- *reply-by* - czas na otrzymanie odpowiedzi

4.2.4. Komunikacja Agentowa

Akt komunikacji

Akt komunikacji jest terminem zapożyczonym z Teorii Aktu Mowy (ang. Speech Act Theory). Wiadomości ACL modelowane są na podstawie takich aktów. Przykłady aktów komunikacji według specyfikacji FIPA to:

- *FIPA-request*: prośba o wykonanie akcji
- *FIPA-agree*: zgoda na wykonanie akcji
- *FIPA-refuse*: odmowa wykonania akcji
- *FIPA-inform*: poinformowanie odbiorcy, że dane stwierdzenie jest prawdą
- *FIPA-failure*: poinformowanie odbiorcy, że wykonanie akcji zakończyło się porażką

Protokoły interakcji

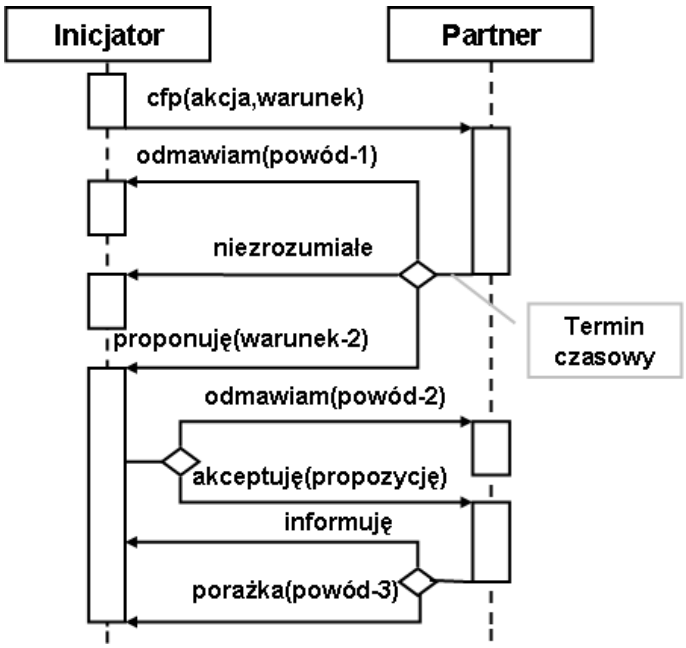
Rozmowa przebiegająca pomiędzy agentami często przybiera pewien określony schemat. Wyszczególnione wzorce wymian wiadomości określa się mianem *protokołów interakcji*. W takim przypadku, oczekuje się pewnej sekwencji następujących po sobie określonych wiadomości.

Podczas budowy systemu agentowego można zbudować agentów „świadomych" swych celów i mentalnych postaw oraz znających cel każdej wysłanej wiadomości. Sposób ten wymaga jednak znacznego nakładu implementacyjnego i może obciążać możliwości agentów. Bardziej pragmatyczny sposób zakłada użycie *protokołów interakcji*, tak, aby z łatwością implementowani agenci, skrupulatnie przestrzegający protokołu, mogli uczestniczyć w „głębszych" konwersacjach z innymi agentami.

FIPA określiła szereg standardowych protokołów interakcji, których można użyć jako wzorce przy budowaniu konwersacji pomiędzy agentami. Zgodna ze specyfikacja FIPA platforma agentowa JADE, dla każdego dialogu, wyróżnia rolę Inicjatora (Initiator) jako agenta, który rozpoczyna konwersację oraz rolę Partnera (Responder) jako agenta, który kontynuuje rozmowę zaczętą przez Inicjatora.

Wizualizacje oraz wsparcie przy projektowaniu stanowią diagramy protokołów interakcji. Diagramy te są diagramami sekwencji AUML[4] [**?**] (przykład protokołu interakcji - rysunek 4.4)

[4] rozszerzenie UML



Rysunek 4.4. Przykład diagramu AUML protokołu interakcji FIPA

opracowano na podstawie [?]

Języki Treści Wiadomości

Do wyrażenia treści wiadomości ACL można użyć jakiegokolwiek języka rozumianego przez nadawcę i odbiorcę, jak XML, RDF, Prolog czy SQL. FIPA wspiera języki: FIPA-SL, FIPA-CCL, FIPA-RDF oraz FIPA-KIF, spomiędzy których najbardziej rozpowszechniony jest FIPA-SL, wyróżniający trzy typy wyrażania treści:

- Stwierdzenie (ang. Proposition) - zgodne z gramatyką stwierdzenie, które w pewnym kontekście może być prawdą,
- Akcja (ang. Action) - do wyrażenia prośby o wykonanie pewnego działania,
- IRE (Identifying Reference Expression) - do wyrażania zdań typu: „takie x, dla którego stwierdzenie P(x) jest prawdą", gdzie P - jest pewnym stwierdzeniem (używane przy zapytywaniu innego agenta).

4.3. JADE

JADE (Java Agent DEvelopment Framework) jest platformą agentową w pełni zgodną ze specyfikacjami FIPA. Zgodność ze specyfikacjami FIPA oznacza, obecność obowiązkowych komponentów AMS, DF oraz ACC[5]. Platforma opiera swoje działanie na współdziałaniu kilku maszyn wirtualnych Java (Java Virual Machine). Każda maszyna wirtualna jest kontenerem w obrębie platformy, w której mogą rezydować agenci. Komunikacja pomiędzy kontenerami odbywa się przy pomocy zdalnych wywołań JAVA RMI[6], natomiast w obrębie jednego

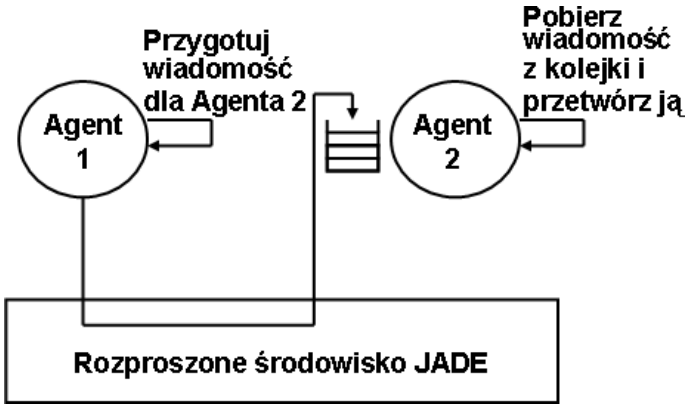
[5]patrz rozdział 4.2

[6]zdalne wywołanie metod, więcej na ten temat: http://java.sun.com/products/jdk/rmi/

kontenera używa się mechanizmu sygnałów. Rolę kontenera dla agentów AMS oraz DF pełni kontener o nazwie *Main-Container* (Kontener Główny), który powstaje pierwszy po uruchomieniu platformy. Po starcie platformy i uruchomieniu kontenera *Main-Container*, istnieje możliwość przyłączenia pozostałych kontenerów. Kontenery te mogą być dołączane także z innym komputerów w sieci, w łatwy sposób tworząc rozproszoną aplikację.

Wiadomości

Format wiadomości używanych w platformie JADE jest zgodny ze specyfikacją FIPA ACL. Przesyłanie wiadomości odbywa się w sposób asynchroniczny. Wiadomość wysłana do agenta trafia do jego „skrzynki na listy" (kolejka komunikatów) o czym odbiorca jest informowany (rysunek 4.5)



Rysunek 4.5. Przesyłanie wiadomości w platformie JADE

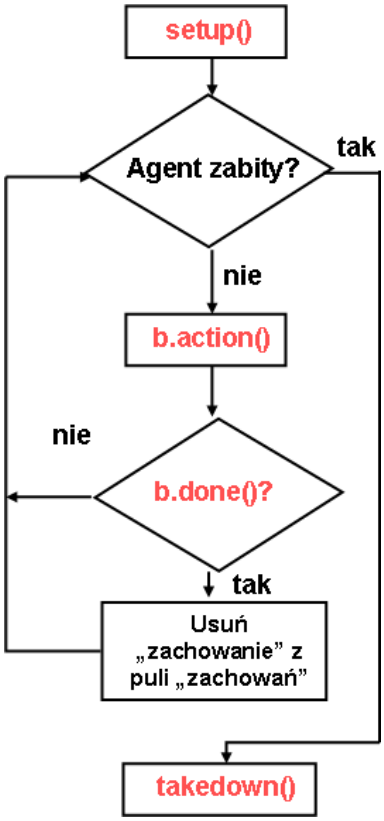
opracowano na podstawie [?]

Zadania agentów

Aplikacja oparta na JADE składa się z kompozycji agentów. Każdy agent w obrębie platformy ma unikalne imię oraz identyfikator AID (Agent IDentifier). Każdy agent wykonuje zadanie według schematu ilustrowanego przez rysunek 4.6, z którego można zauważyć, iż większość przejawów aktywności agenta zawarta jest w *zachowaniu* agenta, które jest implementowane przez klasę dziedziczącą z klasy `Behaviour` platformy JADE. Agent może „być zajęty" kilkoma zachowaniami (może mieć dodanych kilka instancji „zachowań"), które wykonują się jednakże w ramach jednego wątku. Zmiana zachowania następuje po wykonaniu się metody `action()` aktualnie „wykonywanego" zachowania. Zachowanie kończy się, gdy metoda `done()` zwróci wartość `true`.

Platforma dostarcza zbiór najczęściej potrzebnych zachowań a wśród nich:

- Zachowanie „jeden strzał" (`OneShotBehaviour`) - zachowanie kończy się po wykonaniu metody `action()`,
- Zachowanie cykliczne (`CyclicBehaviour`) – metoda `done()` zwraca zawsze `false`, więc zachowanie nigdy się nie kończy,



Rysunek 4.6. Ścieżka wykonywania zadania przez agenta. Na czerwono zaznaczone są metody, które programista musi zaimplementować.

opracowano na podstawie [?]

- Zachowanie złożone, wzorujące się na automacie stanów skończonych (Finite State Machine) (`FSMBehaviour`) - zachowanie wykonuje zarejestrowane wcześniej podzachowania[7] w oparciu o FSM zdefiniowane przez użytkownika.

Zarządzanie treścią wiadomości

Model komunikacji międzyagentowej FIPA oparty jest na założeniu, że agenci zaangażowani w dialog, będą dzielić wspólną ontologię, czyli kontekst rozmowy [?].

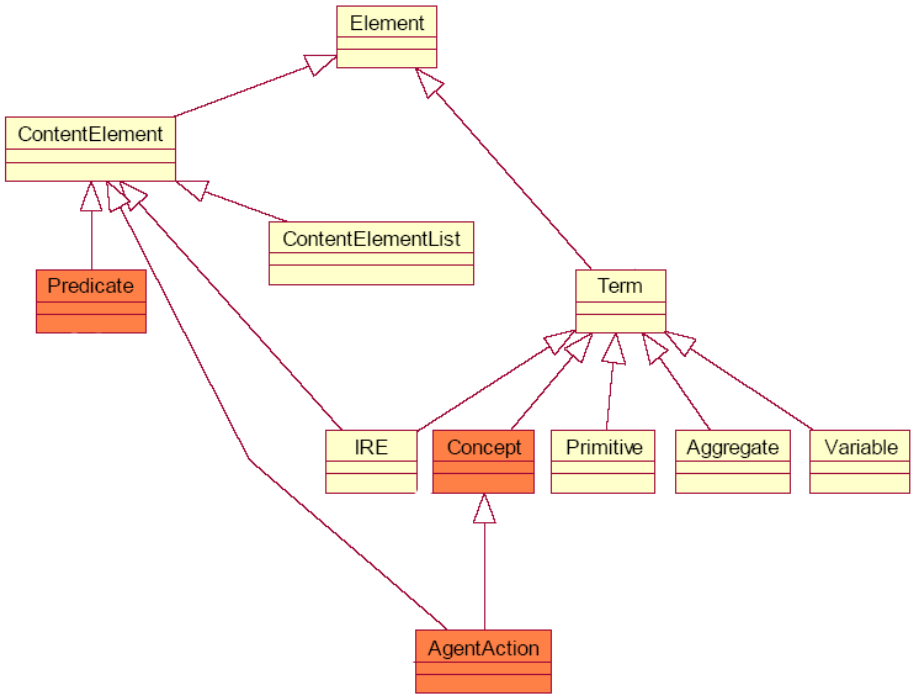
Platforma JADE dostarcza zbiór elementów (interfejsów), za pomocą których użytkownik może budować ontologie w postaci klas Java. Elementy ontologii przedstawia rysunek 4.7 i są to:

- Predykaty (ang. Predicates) - wyrażające pewną informację o stanie świata, która może być prawdziwa lub nie

- Termy (ang. Terms) - wyrażenia identyfikujące jednostki (abstrakcyjne lub konkretne), istniejące w opisywanej części świata, które można podzielić na:

[7] wolne tłumaczenie słowa „subbehaviour", które pojawia się w dokumentacji JADE i oznacza zachowanie wykonujące się w ramach innego zachowania

– Koncepty (ang. Concepts) - jednostki o złożonej strukturze, które mogą być zdefiniowane jako zbiór pól o danym typie, na przykład: (KOŁO (OBWÓD:30, KOLOR:20))
– Akcje agentów (ang. Agent Actions) - szczególny przypadek konceptu wskazującego na akcję, które ma być wykonana przez innego agenta, na przykład: (SPRZEDAJ (OSOBA:SŁAWEK) (ARTYKUŁ:JABŁKO))
– Prymitywy (ang. Primitives) - jednostki niepodzielne jak ciąg znaków czy liczba
– Grupy (ang. Aggregates) - jednostki określające inne jednostki jako grupy, na przykład: (SEQUENCE (PERSON :NAME JOHN) (PERSON :NAME BILL))
– IRE (Identifying Referential Expressions) - wyrażenie identyfikujące te jednostki dla których dany predykat jest prawdziwy, na przykład: (WSZYSTKIE ?X (PRACUJE-DLA ?X (FIRMA :NAZWA TILAB)) - identyfikuje wszystkie osoby pracujące dla firmy o nazwie TILAB.
– Zmienne (ang. Variables) - wyrażenia identyfikujące ogólny nieznany element



Rysunek 4.7. Elementy ontologii JADE

opracowano na podstawie [?]

Ontologię można budować za pomocą powyższych elementów. Każdy element musi mieć określony schemat, który definiuje jego strukturę. Do stworzenia konkretnej ontologii potrzeba, zatem dwóch kroków:

1. określenia schematu dla elementów ontologii,
2. zdefiniowania korespondujących klas reprezentujących te elementy.

4.3.1. JADE-LEAP

Obecnie platformy agentowe wspierają głównie sieci z ustalonym, stałym połączeniem. W konsekwencji rozwoju zarówno bezprzewodowych sieci jak GPRS, UMTS, WLAN, BLUETOOTH jak i możliwości takich urządzeń jak telefony komórkowe, aplikacje agentowe powoli wkraczają w świat mobilnych urządzeń.

Niestety, platformy JADE, w swojej tradycyjnej formie, nie można użyć na tego typu urządzeniach. Po pierwsze, obszar w pamięci komputera zajmowany przez JADE liczony jest w megabajtach. Po drugie, platforma JADE została zaimplementowana w JDK 1.4 podczas gdy większość urządzeń przenośnych wspierana jest przez Personal Java lub J2ME. Po trzecie, bezprzewodowe połączenia, które cechują: duże opóźnienia, niska przepustowość, częste rozłączenia oraz zmienność dynamicznie przydzielanych numerów IP, nie były brane pod uwagę podczas projektowania platformy.

Rysunek 4.8. Środowisko działania JADE-LEAP

opracowano na podstawie [**?**]

Wytyczne projektu LEAP (Lightweight Extensible Agent Platform) [**?**] sprowadzają się do stworzenia technologii umożliwiającej funkcjonowanie agentów zgodnych ze specyfikacją FIPA na urządzeniach, które obsługują język Java, mające „wystarczające"[8] zasoby oraz posiadają połączenie z siecią. Projekt ten przez dwa i pół roku finansowała Komisja Europejska oraz wspierany był przez centra badawcze firm z całej Europy: Francja (Motorola), Niemcy (Siemens, ADAC), Irlandia (Broadcom), Anglia (BT), Włochy (TILAB). Efekt pracy zespołu LEAP to platforma agentowa LEAP zgodna ze specyfikacją FIPA, działająca na urządzeniach z obsługą języka Java począwszy od telefonów komórkowych skończywszy na dużych serwerach obsługujących J2EE.

Platforma JADE-LEAP, jest hybrydą projektów JADE i LEAP, w której wdrożono biblioteki projektu LEAP, zmieniając rdzeń platformy JADE. Projekt ten, zakładał trzy różnie środowiska działania: J2SE, PJava oraz MIDP. Z tego względu powstały trzy różne implementacje platformy, mające jednak taki sam interfejs programisty, zapewniając tym samym warstwę jednorodności ponad różnorodnością cech urządzeń[9]. Dlatego też z punktu

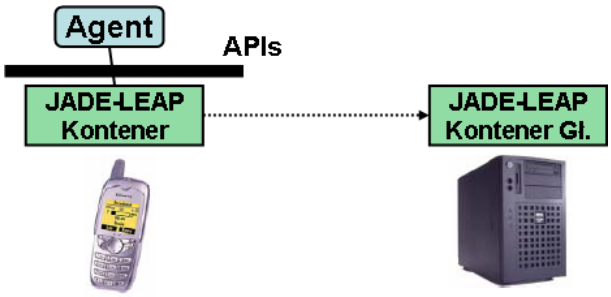
[8] minimalne

[9] z wyjątkiem JADE-LEAP dla MIDP, której brakuje kilku własności ze względu na ograniczenia języka J2ME

widzenia programisty (API) oraz użytkownika (administracja) JADE-LEAP dla J2SE jest niemal identyczny z JADE[10]. Środowisko działania JADE-LEAP ilustruje rysunek 4.8.

JADE-LEAP umożliwia działanie środowiska na urządzeniach przenośnych na dwa sposoby:

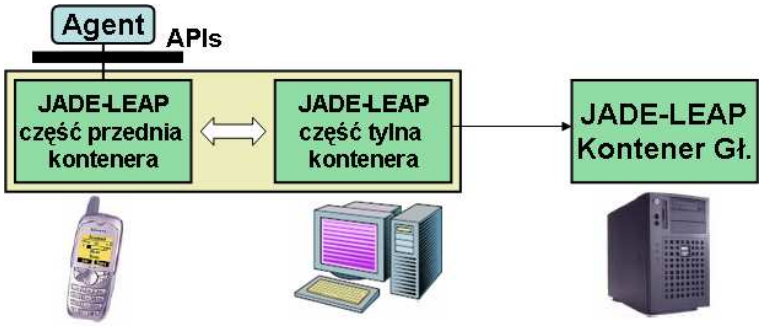
- *działanie niezależne* (ang. stand-alone) - Cały kontener uruchamiany jest na urządzeniu przenośnym (rysunek 4.9).



Rysunek 4.9. JADE-LEAP: działanie samodzielne

opracowano na podstawie [?]

- *działanie podzielone* (ang. split) - Kontener podzielony jest na *część przednią* (ang. FrontEnd), która operuje po stronie urządzenia oraz *część tylną* (ang. BackEnd), która operuje po stronie serwera. Obie części komunikują się poprzez połączenie sieciowe (rysunek 4.10). Rozwiązanie to ze względu na ograniczenia zasobów urządzeń przenośnych jest odpowiedniejsze, ponieważ:
  - przez sieć przesyłana jest *część przednia*, o niższej objętości,
  - inicjalizacja kontenera jest szybsza,
  - mniejsza ilość danych przesyłana jest przez bezprzewodowe połączenie.



Rysunek 4.10. JADE-LEAP: działanie dzielone

opracowano na podstawie [?]

[10] należy pamiętać, że kontenery pomiędzy JADE a JADE-LEAP nie mogą być mieszane w obrębie jednej platformy

4.4. Publikowanie dokumentów RDF/XML

4.4.1. Cocoon

Cocoon, część projektu Apache XML Project, jest elastyczną platformą służącą publikowaniu dokumentów XML przy użyciu XSTL (XML Stylesheet Language-Transformation). Projekt ten, poprzez użycie technologii XML (opis zawartości) oraz XSTL (transformacja zawartości) wspiera rozwój aplikacji z silną separacja zawartości, logiki i prezentacji.

Cocoon dostarcza szeroki wachlarz komponentów wielokrotnego użytku, których odpowiednia konfiguracja może tworzyć złożone zachowanie aplikacji z minimalnym wkładem programisty. Używając XML i XSLT, Cocoon jest w stanie:

- serwować strony statyczne,
- przetwarzać zapytanie użytkownika,
- wykonywać pojedyncze lub wielokrotne transformacje XSLT,
- dynamicznie przekazywać parametry do transformacji XSLT,
- generować wiele typów dokumentów, włączając formaty takie jak: XML, HTML, PNG, JPEG, PDF.

Architektura Cocoona opiera się na trzech konceptach:

- Przetwarzanie dokumentu XML podzielone jest na kilka kroków. Kombinacja tych kroków opisuje pewną sekwencję przetwarzania (ang. pipeline). Sekwencja składa się z wejścia, jednostek przetwarzających i z wyjścia.
- Każdy krok sekwencji można opisać za pomocą komponentu odpowiedniego typu. Cocoon dostarcza kilka takich typów komponentów i ich implementacje.
- Formułowanie odpowiedzi dla użytkownika wiąże się z identyfikacją zapytania z odpowiednią sekwencją a następnie przekazanie tej sekwencji danych wejściowych.

Cocoon wyróżnia kilka ogólnych komponentów sekwencji:

- *Generators, Readers* - komponenty wejścia - odpowiedzialne za dostarczenie zawartości (na przykład zawartości pliku) do sekwencji przetwarzającej.
- *Transformers, Actions* - komponenty przetwarzające - transformują treść najczęściej poprzez transformator XSLT.
- *Serializers* - komponenty wyjścia - punkty wyjścia z sekwencji, których odpowiedzialność to odebranie wyników działania poprzednich w sekwencji komponentów i przetworzenie tych danych na rządany format.
- *Matchers, Selectors* - komponenty warunkujące sekwencje przetwarzania - pozwalają na tworzenie nietrywialnych sekwencji zawierających instrukcje warunkowe *if* lub *if-then-else*.

Sekwencja Cocoon, musi zawierać, co najmniej jeden *Generator* i *Serializer* oraz dowolną liczbę jednostek przetwarzających.

4.4.2. Raccoon

Głównym narzędziem używanym przez Agenta Transformatora[11] jest serwer Raccoon [**?**]. Raccoon jest prostą aplikacją-serwerem, używającą modelu RDF jako źródła danych, działającą w pewnej analogii z platformą Cocoon. Aplikacja ta dostarcza zarówno ogólny mechanizm do odpowiadania na arbitralne zapytania, jak i wspiera zapytania określonego typu jak HTTP czy XML-RPC. Raccoon można zanurzyć w inną aplikację (wywołując odpowiednią komendę z linii poleceń) lub uruchomić oddzielny serwer HTTP udostępniający jego funkcjonalność.

Na potrzeby aplikacji Raccoon powstały między innymi języki:

- RxPath [**?**], syntaktycznie identyczny z XPath 1.0 [**?**], który służy do zapytywania modelu RDF.
- RxSLT [**?**], syntaktycznie identyczny z XSLT 1.0 [**?**], który jest procesorem dla modelu RDF, analogicznym do procesora XSLT dla dokumentów XML.

Aplikacja oparta o Raccoon określa sekwencje wyrażeń RxPath, które służą jako reguły łączące zapytanie z konkretną akcją aplikacji. Dokładniej, przychodzące zapytanie dopasowywane jest do sekwencji *Akcji*. *Akcja* składa się z dwóch części: jednego lub wielu *wyrażeń dopasowujących* (wyrażenia RxPath) oraz z *funkcji akcji*. W przypadku, gdy metadane odpowiadają *wyrażeniom dopasowującym* danej *Akcji* wykonywana jest związana z tą *Akcją funkcja akcji*. Wynik działania *Akcji* przekazywany jest do kolejnej *Akcji*. Wynik działania ostatniej *Akcji* traktowany jest jako odpowiedź dla klienta.

Rdzeniem serwera Raccoon jest klasa *RequestProcessor*. Każda instancja tej klasy jest powiązana ze źródłem danych jako Obiektowy Model Dokumentu (ang. Document Object Model) w postaci RxPath modelu RDF. Raccoon kojarzy metadane zapytania ze zmiennymi XPath.

Raccoon dostarcza gotowe elementy budujące *Akcje* aplikacji. Przykładowo, dostępny jest zbiór komponentów przetwarzających zawartość modelu, dostępny jako *funkcje akcji*. Możliwe jest między innymi przetwarzanie zawartości modelu przez procesory jak: XSLT [**?**], RxSLT [**?**], Python [**?**] oraz RxUpdate [**?**].

4.5. XUL

XUL (XML User-interface Language) jest międzyplatformowym językiem służącym do opisu interfejsu użytkownika. Poprzez ustandaryzowaną formę, XUL pozwala na stworzenie ujednoliconego interfejsu użytkownika niezależnie od platformy (komputer biurowy PC, PDA czy telefon komórkowy). W celu ułatwienia projektowania w XUL osobom niedoświadczonym w programowaniu i projektowaniu, język ten umożliwia korzystanie z technologii jak CSS, JavaScript, DTD czy RDF.

Interfejs w XUL opisany jest zazwyczaj przez trzy komponenty:

- Treść - dokument XUL zawierający definicje interfejsu oraz układu graficznego
- Powłoka Graficzna - pliki: CSS, graficzne, które definiują *powierzchowność* aplikacji
- Lokalizacja - dokument DTD (Document Type Declaration), definiujący encje[12] dla łatwej internacjonalizacji interfejsu - poniższy przykład prezentuje użycie encji w przypadku przycisku:

[11] agent transformujący zawartość opisaną w języku RDF na postać czytelną dla urządzenia użytkownika
[12] cecha języka XML; na przykład encja &lt oznacza znak mniejszości

```
<button label="&findLabel;"/>
```

Tekst, który się ukaże na etykiecie (ang. label), będzie wartością encji `&findLabel`. Można tę cechę wykorzystać przy projektowaniu interfejsu wielojęzycznego. W wersji angielskiej `&findLabel` będzie miało wartość „find", w polskiej zaś „szukaj".

Specyfikacja XUL zapewnia szeroką gamę elementów interfejsu:

- *elementy wysokiego poziomu* - okna, strony, okna dialogowe, etc.
- *kontrolki* - etykiety, przyciski (zwykłe, „radio" lub „check"), pola tekstowe, listy, menu, drzewa, podziałki, etc.
- *układ, rozmieszczenie* - pudło, siatka, kopiec, poziomowy, etc.
- *zdarzenia i skrypty* - skrypty, komendy, przyciski, etc.
- *źródła danych*

XUL został opracowany podczas rozwoju przeglądarki internetowej Mozilla[13]. Zastosowano go między innymi w Mozilla Amazon Browser, która oferuje bogaty interfejs użytkownika do wyszukiwania książek na Amazon.com[14].

4.5.1. Thinlet

Thinlet [**?**, **?**] jest implementacją technologii XUL całkowicie zaimplementowaną w języku Java. Narzędzie to powstało z myślą o wdrożeniu interfejsów użytkownika opisanego za pomocą języka XUL w apletach Java oraz na małych urządzeniach przenośnych takich jak telefony komórkowe (całkowita objętość to mniej niż 40 kilobajtów). Narzędzie to pozwala na odseparowanie prezentacji (opisanej w XUL/XML) od metod aplikacji pisanych w Java. Kod projektu zachowuje zgodność z wersją Java 1.1 oraz nie wymaga bibliotek Java Swing[15]. Kod projektu upubliczniony jest pod licencją GNU LGPL[16] (Lesser General Public License) i posiada odmiany dla Java SDK, Personal Java oraz Java MIDP.

4.6. MIDP i CLDC

Java, w początkowym zamierzeniu miała wspierać integrację urządzeń użytkowników. Nie trzeba było długo czekać na gwałtowny rozwój tego języka, który zapewnia interoperacyjne środowiska od przywoływaczy (ang. pagers) do wieloprocesorowych serwerów.

Przedsięwzięciem, które miało zapewnić wkroczenie języka Java na przenośne urządzenia był początkowo projekt Spotless [**?**] firmy Sun[17], który następnie ewoluował w J2ME. Projekt ten dostarczył środowisko budowy aplikacji umożliwiające budowę oprogramowania o niewielkich rozmiarach a także: przenośnego, bezpiecznego, niezależnego od technologii sieciowej, oferującego dynamiczną dostawę usług.

Proces standaryzacji wyłonił dwa poziomy organizacji w architekturze J2ME:

[13] http://www.mozilla.org/
[14] http://www.amazon.com
[15] Standardowe klasy Java służące do budowania graficznego interfejsu użytkownika
[16] www.gnu.org/copyleft/lesser.html
[17] http://java.sun.com/

- *Konfiguracja* - specyfikacja, definiująca minimalną funkcjonalność platform Java (podstawowe biblioteki, możliwości maszyny wirtualnej, specyfikacja bezpieczeństwa, ogólny szkielet połączenia sieciowego) dla pewnej rodziny urządzeń. Wyróżniamy dwa typy *konfiguracji*:
  - Connected Device Configuration (CDC) - konfiguracja dla małych urządzeń o większych możliwościach: systemy nawigacji, niektóre PDAs, etc.
  - Connected, Limited Device Configuration (CLDC) - konfiguracja dla małych, przenośnych urządzeń o ograniczonych możliwościach typu: telefon komórkowy, dwukierunkowy pager, etc.
- *Profil*
  - Mobile Information Device Profile (MIDP) - kolekcja interfejsów Java (API), która wzbogaca *konfigurację* o możliwości specyficzne dla danej grupy urządzeń.

Kluczowe elementy różniące specyfikacje CLDC ze specyfikacją JVM to:

- brak operacji zmiennoprzecinkowych,
- brak finalizacji[18],
- ograniczenia obsługi wyjątków,
- brak natywnych interfejsów,
- brak refleksji[19],
- brak grup wątków,
- brak słabych referencji[20],
- brak możliwości przeciążania klas systemowych.

Wirtualna maszyna Java, ponad którą działa CLDC to Kilo Virtual Machine (KVM) - zaprojektowana z myślą o urządzeniach o znacznych ograniczeniach zasobów. KVM zajmuje w pamięci operacyjnej urządzenia jedynie 60 kilobajtów a jej wymagania to:

- 160 kilobajtów pamięci,
- 16/32 bitowy procesor,
- prędkość procesora: 8-32 MHZ.

Podstawowym modelem aplikacji MIDP jest MIDlet. Model ten wykazuje podobieństwa z apletem Java (bazuje na GUI). Podczas działania, MIDlet przechodzi od stanu do stanu:

- Start – otrzymanie zasobu i rozpoczęcie działania
- Pause – zwolnienie zasobu i czekanie
- Destroy – zwolnienie wszystkich zasobów, zabicie wątków i wszelkiej innej aktywności

[18] możliwość dodania funkcjonalności, wywoływanej przed zniszczeniem obiektu
[19] możliwość badania cech klas: http://java.sun.com/developer/technicalArticles/ALT/Reflection/
[20] więcej na ten temat: http://java.sun.com/developer/technicalArticles/ALT/RefObj/

ROZDZIAŁ 5

Metodologia

5.1. Wstęp

Implementowana przeze mnie kliencka strona Systemu Wspomagania Podróży jest częścią wieloagentowego systemu. Faza projektu musiała uwzględniać przyszły, intensywny rozwój systemu w szczególności z uwagi na fakt, iż strona kliencka (Podsystem Dostarczania Informacji) powstała jako pierwsza.

Rozdział ten jest zestawem opisów metod użytych przeze mnie podczas prac zarówno nad projektem jak implementacją. Zacząłem od zagadnień programowania zorientowanego agentowo w podrozdziale pierwszym. Podrozdział drugi zawiera opis użytej przeze mnie metodologii projektowania systemów wieloagentowych - Prometheus. Ostatni podrozdział, wprowadza w zagadnienie wzorców projektowych w kontekście programowania obiektowego oraz opisuję próbę adaptacji wybranego wzorca na potrzeby programowania agentowego[1].

5.2. Programowanie zorientowane agentowo

W każdej z nowoczesnych metod programowania (proceduralne, obiektowe, agentowe) utrzymuje się tendencja by separować jednostki programowe (procedury, obiekty, agenci) ze względu na wykonywane funkcje. W przypadku programowania proceduralnego, duże procedury rozbija się na kilka mniejszych, prostszych i bardziej wyspecjalizowanych. Dobrze określona ziarnistość pomaga bowiem na niemal każdym kroku realizacji programu (aplikacji):

- *faza projektowania* - ułatwione zarządzanie poszczególnymi jednostkami
- *faza implementacji* - ułatwione wyszukiwanie usterek[2]
- *faza zarządzania*[3] - ułatwia ewentualną podmianę „przestarzałej" jednostki lub dodanie zupełnie nowej jednostki

W celu osiągnięcia pożądanej funkcjonalności aplikacji, konieczna jest interakcja pomiędzy powstałymi wyspecjalizowanymi jednostkami. W przypadku procedur jest to wywołanie jednej procedury przez drugą. W przypadku programowania zorientowanego agentowo jest to wymiana wiadomości. Pod pojęciem programowania agentowego kryje się zatem zbiór oddziałujących ze sobą agentów, czyli system wieloagentowy (MultiAgent System).

Programowanie zorientowane agentowo przewyższa poziomem abstrakcji programowanie zorientowane obiektowo [**?**]. Z tego względu, może ono znaleźć zastosowanie w przedsięwzięciach o niespotykanej złożoności jak: kontrola ruchu lotniczego, kontrola procesów przemysłowych, telekomunikacji, etc. Zacznę jednak od podstaw.

Źródłem wielu wskazówek dla metod programowania zorientowanego agentowo daje precyzyjna definicja agenta, która według autorów [**?**] jest następująca:

[1] podrozdział Agent-Controller
[2] ang. debugging
[3] ang. manage

Agent programowy jest dobrze skapsułkowanym systemem komputerowym rezydującym w pewnym środowisku, zdolnym do elastycznego i autonomicznego podejmowania akcji w tym środowisku, aby sprostać założonym celom istnienia.

Agenci posiadają następujące cechy:

- są to identyfikowalne jednostki rozwiązujące określone zadania z dobrze określonymi granicami i interfejsami,
- komunikują się ze swoim środowiskiem poprzez sensory (odbieranie sygnałów od otoczenia) i efektory (oddziaływanie na otoczenie),
- mają określone cele do wypełnienia,
- są autonomiczni - mają pełną kontrolę nad swoim stanem wewnętrznym i nad swoim zachowaniem,
- są reaktywni - są w stanie reagować na zmiany środowiska,
- są proaktywni - są w stanie rozwiązywać nowe, zaadoptowane cele.

Problem zaadoptowania filozofii programowania agentowego w złożonych systemach komputerowych szybko nasuwa pośredni wniosek, iż należy użyć wielu agentów [**?**], głównie w celu właściwego odwzorowania:

- zdecentralizowanej natury problemu,
- wielu punktów kontroli,
- wielu perspektyw,
- współzawodniczących zadań.

Ponadto, agenci muszą być w stanie oddziaływać na siebie ze względu na przykład na wykonywane działanie lub z powodu innych zależności wynikających z bytowania we wspólnym otoczeniu. Może to być zwykła wymiana informacji, prośba o wykonanie jakiegoś zadania, działania koordynacyjne, działania kooperacyjne czy negocjacja. Omawiane podejście różni się w tym momencie od innych paradygmatów inżynierii oprogramowania - interakcje pomiędzy jednostkami programowymi wzniosły sie na poziom wymiany wiadomości, czyli na poziom wiedzy [**?**].

Nie ma żadnych konkretnych danych wykazujących wyższość podejścia agentowego nad innymi. Można jednakże próbować teoretycznie wykazać, że podejście to jest jakościowo lepsze. Wyzwania tego podjął się Jennigs w [**?**], konfrontując dobrze znane techniki rozwiązywania złożonych problemów inżynierii oprogramowania z kluczowymi cechami podejścia agentowego, analizując w ten sposób stopień ich pokrewieństwa.

Do głównych metod pokonywania trudności ze złożonymi systemami należą [**?**]:

- dekompozycja - dzielenie problemu na mniejsze, łatwiejsze do wykonania „kawałki", które cechuje względna izolacja
- abstrakcja - proces wyróżniania pewnych detali systemu i zaniedbywania innych
- organizacja - proces identyfikacji i zarządzania relacjami pomiędzy komponentami

Skądinąd z [**?**] wiemy, że:

- Złożoność systemu często przyjmuję formę hierarchii podsystemów. System składa się ze wzajemnie połączonych komponentów. Struktura każdego komponentu zachowuje formę hierarchii. Komponenty najniższe w hierarchii wykonują najbardziej prymitywne zadania. Wyizolowano kilka ogólnych schematów takich hierarchii jak architektura klient-serwer.

- Wybór komponentów wykonujących prymitywne zadania jest stosunkowo arbitralny i dyktowany jest celami obserwatora.

- Systemy hierarchiczne ewoluują szybciej.

- Możliwe jest odróżnienie interakcji pomiędzy podsystemami i interakcji wewnątrz podsystemu. Interakcje wewnątrz podsystemów są częstsze i łatwiejsze do przewidzenia. Daje to podstawy by rozważać podsystemy jako wysoce niezależne i rozwijać je równolegle.

Agentowo zorientowana dekompozycja

Złożone systemy przyjmują formę hierarchii komponentów. Na każdym poziomie tej hierarchii elementy podsystemu działają w kolektywie by osiągnąć funkcjonalność rodzica. Naturalny wydaje się, więc podział na komponenty, ze względu na wykonywane zadania. Ponadto, w złożonych systemach obserwuje się brak scentralizowanej kontroli [**?**]. W kontekście dekompozycji ukierunkowanej na wykonanie zadania, skłania to do umiejscowienia tych punktów kontroli wewnątrz komponentów. Oznacza to, iż każdy podsystem ma kontrolę nad swoim zachowaniem (akcje, wybory, etc.).

Elementy podsystemu muszą ze sobą współpracować. W przypadku złożonych systemów, przewidzenie wszystkich możliwych tego typu interakcji na poziomie projektu i implementacji jest bardzo trudne. Istnieje, zatem potrzeba by komponenty posiadały zdolność elastycznej obsługi interakcji w czasie działania programu. Rozwiązanie to ułatwia łączenie jednostek programowych. Elementy podsystemu, w razie nieprzewidzianego zlecenia mogą „spontanicznie" poprosić o asystę. Ponadto, ze względu na fakt, iż interakcje odbywają się na poziomie wiedzy, łączenie jednostek programowych odbywa się także na poziomie wiedzy (nie ma problemów syntaktycznych związanych z nieprzewidzianymi interakcjami).

Agent, jako jednostka autonomiczna i posiadająca dobrze określone zadania do wykonania, idealnie dopasowuje się do pojęcia komponentu podsystemu. Kolektyw agentów odpowiada pojęciu podsystemu, jako hierarchii współpracujących komponentów.

Agentowo zorientowana abstrakcja

Znalezienie właściwego modelu odwzorowującego problem to połowa sukcesu. Dobre abstrakcje minimalizują semantyczną przestrzeń pomiędzy konstrukcjami użytymi w projekcie a jednostkami konceptualizującymi problem. W przypadku złożonych systemów, tymi jednostkami są: podsystem, komponenty podsystemu, interakcje oraz organizacje. Poniższa tabela jest zestawieniem abstrakcyjnych pojęć (składających się na złożony system), z ich odpowiednikami agentowymi.

pojęcie abstrakcyjne	podejście agentowe
podsystem	organizacja współdziałających agentów
komponenty podsystemu	agenci
interakcje	wymienianie wiadomości, protokoły interakcji
powiązania, organizacja	protokoły do tworzenia, utrzymywania i niszczenia grup działających w kolektywie w celu osiągnięcia wspólnego celu.

Agentowo zorientowana organizacja

Struktury organizacyjne są podstawowym elementem systemów agentowych. Ponadto, systemy agentowe posiadają zdolność do tworzenia, utrzymywania i niszczenia organizacji. Pozwala to na eksploatacje dwóch ważnych aspektów złożonych systemów. Po pierwsze, pojęcie prymitywnego komponentu może się zmieniać w zależności od obserwatora. Na jednym poziomie, cały podsystem może być uznany za prymitywny, zaś na innym za prymitywne uznać można komponenty tego podsystemu. Po drugie, struktury organizacji agentowych ułatwiają tworzenie form pośrednich. Dzięki temu, agent lub grupa agentów może rozwijać się we względnej izolacji i w razie potrzeby być dodawany do systemu.

5.3. Prometheus

Prometheus jest metodologią, wspomagającą proces szczegółowego projektowania i implementacji inteligentnych systemów wieloagentowych. Projekt ten rozwijany jest już od kilku lat na bazie doświadczenia zarówno akademickiego jak i przemysłowego. Składa się on z trzech podstawowych faz: specyfikacja systemu, projektowanie architektury oraz projektowanie szczegółowe. Schemat poszczególnych faz obrazuje rysunek 5.1.

5.3.1. Specyfikacja systemu

Ze względu na proaktywność agentów widoczną na niemal każdym poziomie systemów wieloagentowych, specyfikację zaczyna się od wyznaczenia celów (System Goals) i podcelów systemu. Na ich podstawie określa się *percepty*[4] (Percepts), które z kolei wspomagają proces konceptualizacji *akcji* (Actions). Powyższy proces pozwala na łatwiejsze wyznaczenie ról (Roles) oraz *scenariuszy* (Scenarios).

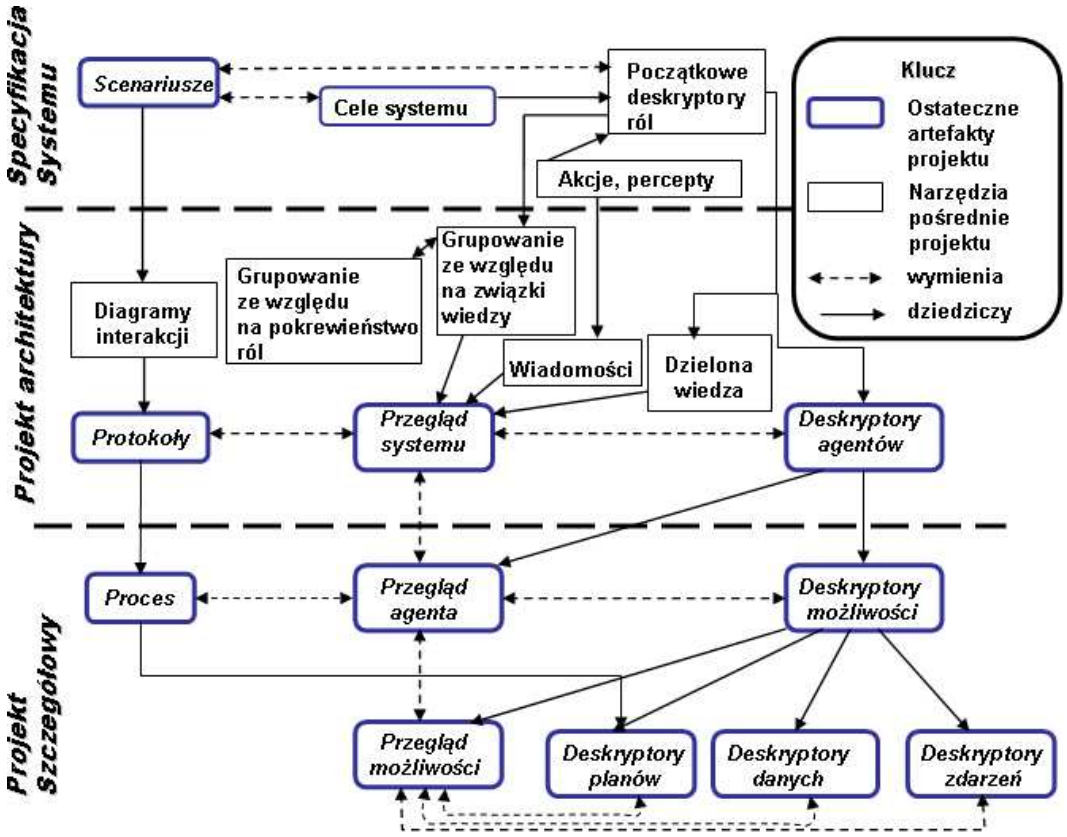
5.3.2. Projektowanie architektury

Typy agentów

Projektowanie architektury systemu można zacząć od wyznaczenia typów agentów. Proces rozpoczyna się na analizą ról mającą następujące fazy:

- grupowanie ról ze względu na używaną lub modyfikowaną wiedzę (przykład - rysunek 5.2)
- rewizja powstałych grup ze względu na pokrewieństwo ról (przykład - rysunek 5.3)
- rewizja powstałych grup ze względu na częstotliwość interakcji (przykład - rysunek 5.3)

[4] zdarzenia zauważane przez agentów



Rysunek 5.1. Schemat poszczególnych faz należących do procesu projektowania metodą Prometheus

opracowano na podstawie [?]

Role nie powinny znajdować się w jednej grupie jeśli: role nie są spójne, znajdują się na różnych maszynach lub gdy ich liczebności w systemie są różne. Bazując na wydzielonych grupach, tworzy się prototypy agentów. Każdy z takich prototypów wymaga określenia cyklu życia, liczebności w systemie oraz innych szczegółów związanych z jego egzystencją, czego podsumowaniem jest stworzenie deskryptora agenta[5] (Agent Descriptor).

Interkacje między agentowe

W kolejnym kroku projektowania określa się interakcje pomiędzy poszczególnymi agentami. Źródłem przesłanek dla tego procesu są *scenariusze*, *percepty* i *akcje* agentów. Do wizualizacji wyników tej konceptualizacji używa się diagramów interakcji AUML[6]. Wydzielone diagramy interakcji są następnie generalizowane do protokołów interakcji (rysunek 5.4).

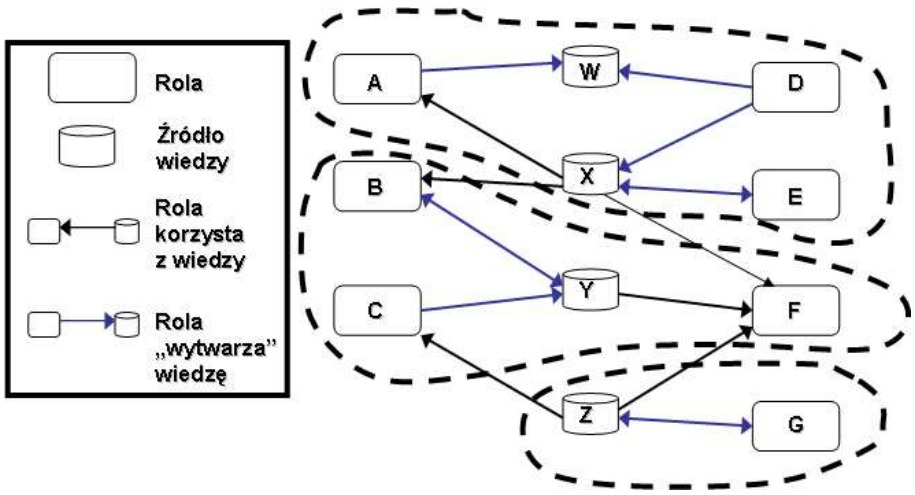
Przegląd systemu

W celu sfinalizowania tej części projektu należy ponadto:

- opisać wszelkie relacje pomiędzy *perceptami*, *akcjami* oraz agentami

[5]lista cech agenta

[6]Diagramy interakcji AUML wzorują się na diagramach sekwencji UML.



Rysunek 5.2. Przykład grupowania ról ze względu na związki modyfikujące wspólne źródło wiedzy (czerwone strzałki). Powstałe grupy zaznaczone są krzywą.

- opisać wszystkie źródła danych (SHARED DATA/KNOWLEDGE)
- opracować diagram stanowiący przegląd systemu (SYSTEM OVERVIEW)

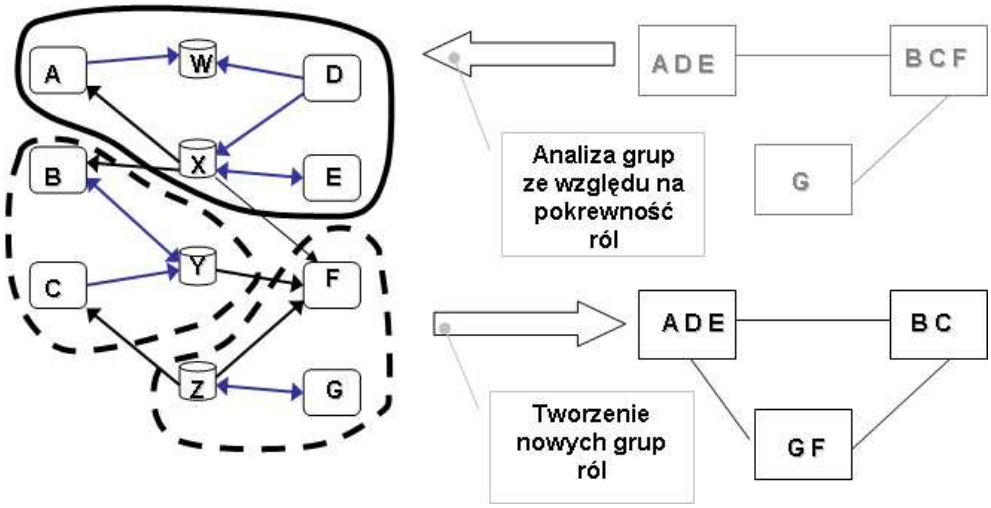
5.3.3. Szczegółowy projekt

Ostateczny krok metodologii Prometheus to projektowanie szczegółowe. W tej fazie należy określić strukturę agenta, która jest konstruowana z uwzględnieniem kompetencji danego agenta. Kompetencje agenta wyznacza się na podstawie jednej z grup ról powstałej we wcześniejszej fazie. Każde zadanie (w ramach określonej kompetencji) agenta musi mieć określony plan akcji.

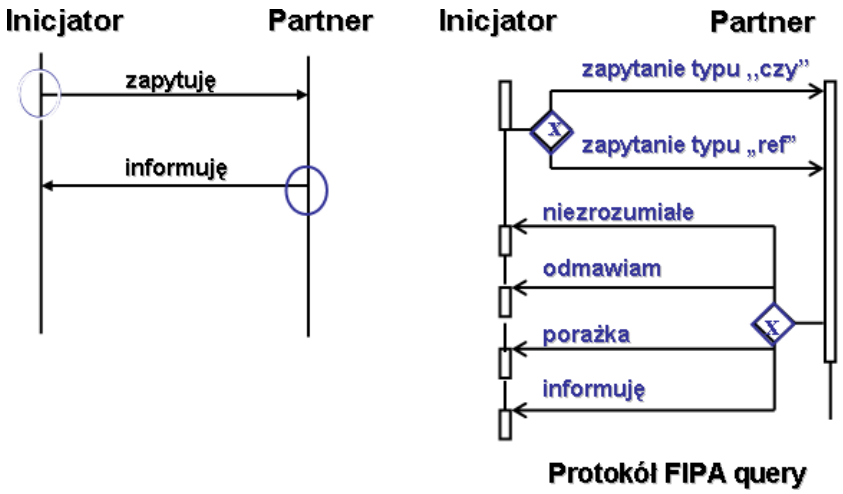
Elementów modelowania metodologii Prometheus używa się do reprezentacji podstawowych praw rządzących w programowaniu zorientowanym agentowo. Brakuje jej, jednakże dobrej dokumentacji (formalnej specyfikacji) co utrudnia poprawne łączenie poszczególnych konstruktów modelujących. W niedalekiej przyszłości planuje się jednakże integrację metodologii Prometheus z FIPA AUML.

5.4. Wzorce projektowe

Wraz z powstaniem idei programowania obiektowo zorientowanego pojawiła się nadzieja na oprogramowanie bardziej niezawodne, bezpieczniejsze i łatwiejsze w utrzymaniu. Jedną z najważniejszych zalet tej techniki miała być możliwość wielokrotnego użycia tego samego kodu. Dziś nie jest tajemnicą, iż pisanie takiego kodu nie jest łatwe. Należy wyznaczyć trafnie obiekty a następnie klasy o odpowiedniej ziarnistości, zdefiniować interfejsy, hierarchie dziedziczenia oraz ustanowić odpowiednie powiązania między nimi. Rozwiązanie problemu powinno być nastawione na specyfikę problemu, jednocześnie zachowując odpowiedni stopień ogólności by ułatwić przyszły rozwój systemu. Wśród ekspertów oprogramowania panuje opinia, że stykając się z zupełnie nową klasą problemów nie jest możliwe stworzenie optymalnego, elastycznego kodu za pierwszym podejściem. Różnica pomiędzy nowicjuszami a ekspertami w tej dziedzinie polega właśnie na różnicy w doświadczeniu.



Rysunek 5.3. Powstałe grupy (rysunek 5.2) są rewidowane ze względu na stopień pokrewieństwa (na przykład częstotliwość interakcji pomiędzy rolami).



Rysunek 5.4. Proces generalizacji diagramów interakcji

Wzorce projektowe są swego rodzaju „doświadczeniem w pigułce”. Każdy wzorzec projektowy nazywa, wyjaśnia i szacuje ważny i powtarzający się schemat w systemach zorientowanych obiektowo. Użycie wzorca, można postrzegać w kategorii dodania kolejnej warstwy abstrakcji - techniki podstawowej dla projektowania obiektowego. Pierwsza publikacja na temat wzorców wyszła spod ręki trzech autorów Ericha Gamma'y, Richarda Helma, Ralpha Johnsona oraz Johna Vlissidesa [**?**].

5.4.1. Wzorzec projektowy: Kontroler

W książce [**?**] wzorzec `Kontroler` (ang. Controller) opisany jest jako komponent, który odpowiada za interakcję z klientem w tym za zażądanie jego zapytaniami. Używany jest jednakże zarówno w warstwie prezentacji jak i w warstwie biznesowej. W ogólności wzorzec ten:

- akceptuje zapytanie,

- wykonuje operacje na zapytaniu,
- wybiera odpowiedniego delegata do wykonania zadania,
- przekazuje zapytanie klienta delegatowi,
- może dostarczyć wysokiego poziomu mechanizm obsługi błędów i wyjątków.

Wzorzec ten jest bardzo popularną architekturą i znajduje szerokie zastosowanie w różnorodnych projektach. W przypadku aplikacji sieciowych, `Kontroler` może przykładowo akceptować zapytanie HTTP, ekstrahować parametry, ciasteczka oraz nagłówki czyniąc owe elementy zapytania łatwiej dostępne. Bazując na zawartości zapytania `Kontroler` używa odpowiednich komponentów systemowych by wypełnić zlecenie. Istotą działania wzorca jest przekazanie wykonania zadania do innego obiektu-delegata. Dzięki temu rozwiązaniu w łatwy sposób (nie wymagający zmian w klasie Kontrolera) można modyfikować obsługę zleceń oraz dodawać obsługę nowych zadań. Oto zbiór interfejsów składający się na ten wzorzec:

- ```
  public interface Request{
      String getName();
  }
  ```

  Interfejs ten reprezentuje pojęcie zlecenia dla `Kontrolera`. Jedyna metoda zwraca unikalną nazwę operacji do wykonania.

- ```
  public interface Response;
  ```

  Interfejs odpowiada abstrakcyjnemu pojęciu odpowiedzi systemu.

- ```
  public interface RequestHandler{
    Response process(Request request) throws Exception;
  }
  ```

  Interfejs klasy będącej delegatem zadania. Jedyna metoda przyjmuje zlecenie i zwraca odpowiedź komponentu. Klasa to spełnia pomocniczą funkcję i wykonuje właściwą pracę związaną z konkretnym zleceniem. Metoda ta deklaruje abstrakcyjny wyjątek, który może nastąpić podczas operacji związanych z wykonaniem zlecenia.

- ```
  public interface Controller{
    Response processRequest(Request request);
    void addHandler(Request request, RequestHandler requestHandler);
  }
  ```

  Definiuje wysokiego poziomu metodę `processRequest`, przetwarzającą nadchodzące zapytania. `Kontroler`, po zaakceptowaniu zlecenia, przesyła je do odpowiedniego delegata (`RequestHandler`). Metoda ta nie deklaruje żadnych wyjątków, gdyż jest metodą znajdującą się na szczycie stosu i obsługuje wszelkie wewnętrzne wyjątki. Druga metoda kojarzy obiekt delegata z instancją zlecenia. Pozawala na rozszerzenie klasę `Kontrolera` bez zmiany kodu źródłowego.

Agent - Kontroler

W ogólności, agent funkcjonalny, ma zadanie przyjąć pewne zadanie, wykonać je i zwrócić rezultaty akcji. PrA otrzymuje żądanie od klienta-przeglądarki, przekazuje odpowiednią wiadomość do PA by po pewnym czasie otrzymać odpowiedź i przekazać ją użytkownikowi.

Klientem TA jest natomiast PA, który przekazuje mu nieprzetransformowane dane od systemu oczekując w odpowiedzi na dane gotowe do zaprezentowania klientowi. Pojawia się zatem schemat wzorca projektowego `Kontroler` opisanego powyżej w tym rozdziale.

Rozpatrzenia wymaga pojęcie delegata[7] do zadania. W ujęciu programowania czysto obiektowego, jest to zwykły obiekt oddelegowany do wykonania pewnego zadania. Obiekt taki, jako delegat, musi mieć możliwość interakcji z pozostałymi elementami systemu na równi z `Kontrolerem`. `Kontroler` jest w naszym przypadku agentem, zatem delegat powinien mieć możliwość wysyłania i odbierania wiadomości w języku ACL. Jedna z możliwości to uczynić delegata agentem. Rozwiązanie to jednak jest mało wydajne ze względu na relatywnie wysoki koszt stworzenia agenta. W przypadku jednoczesnego zgłoszenia się wielu użytkowników do systemu spowodowałoby to znaczne obciążenie serwera. Ponadto, każdy agent w obrębie jednej platformy posiada unikalne imię i tworzenie wielu agentów wymagałoby opracowania metody przydzielania imion nowo stworzonym agentom.

W implementacji tego zagadnienia postanowiłem użyć *zachowań* (ang. behaviours) agentów. W *zachowaniu* agenta powinny zawrzeć się wszelkie przejawy jego aktywności, co w naturalny sposób odpowiada pojęciu wykonania zadania. Ponadto, klasa `Behaviour`[8] jako jedna z nielicznych, posiada bezinterwencyjny dostęp do publicznych metod agenta (ze względu na jego autonomię), które w szczególności mogą służyć do komunikacji z resztą świata agentowego.

Ze względu na asynchroniczność oddziaływań pomiędzy agentami wzorzec `Kontroler` wymaga zmian. Agent A by wykonać wyznaczone zadanie, może być zmuszony do skorzystania z pomocy agenta B (wysyłając mu wiadomość ACL), który odpowie mu w sposób asynchroniczny (też przez wysłanie wiadomości ACL). Istnieją, więc dwa przypadki wykonania zadania przez Agenta-Kontrolera (w obu przypadkach używa się delegatów):

- Samodzielne wykonanie zadania.
- Powierzenie wykonania części zadania innemu agentowi.

W pierwszym przypadku, delegat wykonuje powierzone mu zadanie i zwraca wynik do swojego pracodawcy. W drugim zaś, wykonanie zadania rozdzielone jest na części. Zachowanie agenta, posługuje się aparatem komunikacji agenta, by sukcesywnie, odbierając cząstkowe rezultaty od swoich „agentów-pomocników", zrealizować powierzone mu zadanie.

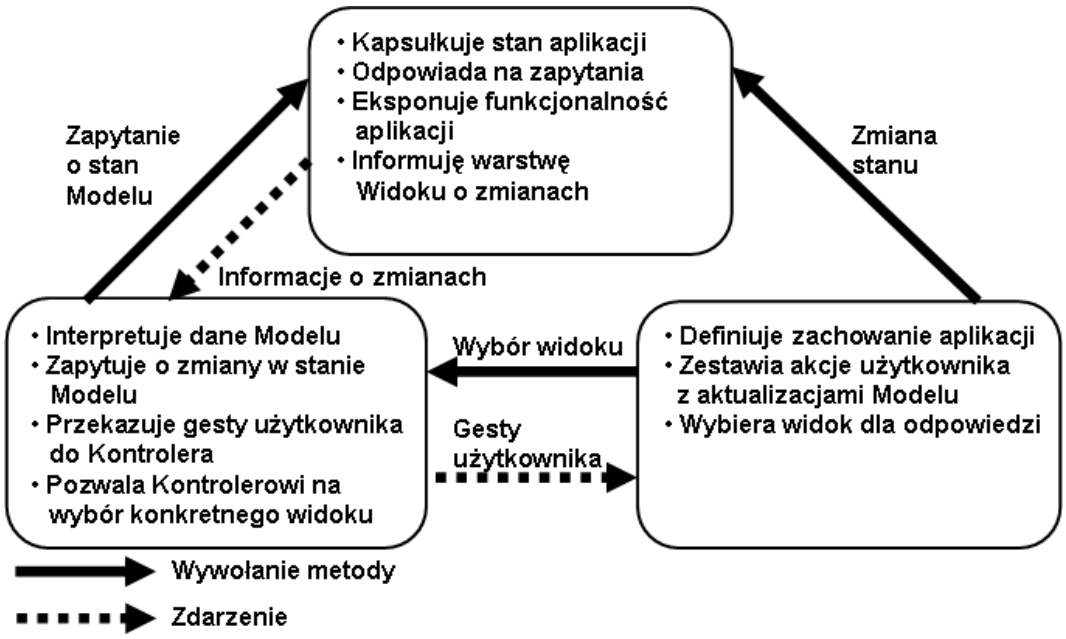
5.4.2. Wzorzec projektowy: Model-Widok-Kontroler

Architektura Model-Widok-Kontroler (ang. Model-View-Controller) jest szeroko stosowana w aplikacjach interaktywnych. Rozdziela ona bowiem prezentację danych od zarządzania danymi, minimalizując w ten sposób stopień powiązań między niejednorodnymi obiektami. Rozwiązanie to dzieli aplikacje na trzy warstwy: Model, Widok i Kontroler różnicując je pod względem wykonywanych zadań. Każda warstwa ma specyficzne zadania i odpowiedzialności:

- Model - Warstwa ta reprezentuje dane biznesowe i cała logikę biznesową, która modyfikuje i zarządza danymi. Model zawiadamia warstwę Widok o zmianach oraz umożliwia wykonywanie zapytań w celu uzyskania specyficznych danych.
- Widok - Warstwa umożliwia prezentację danych uzyskanych z warstwy Model. Aktualizuje prezentacje zawiadomiony przez warstwę Model oraz przekazuje dane dotyczące interakcji użytkownika warstwie Kontroler

[7] ang. request handler

[8] Klasa ta reprezentuję pojęcie zachowania agenta.



Rysunek 5.5. Wzorzec projektowy MVC

- Kontroler - Warstwa ta definiuje zachowanie się aplikacji. Wysyła żądania klienta i wybiera odpowiedni typ prezentacji (poprzez interakcję z warstwą Widok). Interpretuje interakcje użytkownika na akcje do wykonania przez warstwę Model. Na podstawie tych interakcji i rezultatu działania warstwy Model wybiera kolejny widok. W przypadku aplikacji sieciowej, żądania klienta to najczęściej zapytania HTTP GET lub HTTP POST. Aplikacja ma najczęściej jedną warstwę Kontroler dla każdego zbioru pokrewnych funkcjonalności. Niektóre aplikacje używają kilku komponentów typu Kontroler dla różnych typów klientów ze względu na różnice w interakcji tych typów.

ROZDZIAŁ 6

Szczegóły implementacji

Następujący rozdział jest opisem metod i technik, jakimi się posłużyłem przy implementacji najbardziej problematycznych i interesujących dziedzin klienckiej strony Systemu Wspomagania Podróży. Jednym z założeń programowania agentowego jest wszechobecność agentów. Teraźniejszy stan rozwoju technologii agentowej nie spełnia jednak tego oczekiwania, albowiem wciąż istnieją środowiska niedostępne dla agentów (jak telefony komórkowe). Problematyka mojego zadania wyrosła głównie ze starcia się dwóch światów: agentowego i nie-agentowego.

6.1. Agent-Kontroler a protokół interakcji

Należało ustalić, w jaki sposób zaimplementować komponenty systemu, czyli PrA (ang. Proxy Agent), TA (ang. Transformer Agent) i PA (ang. Personal Agent). Rozważania z rozdziału 5.4.1 sugerują użycie wzorca projektowego `Kontroler` zaadoptowanego do podejścia agentowego.

Agent-Kontroler, chcąc udostępnić swoje usługi w środowisku agentowym, musi posiadać mechanizm pozwalający na przyjmowanie zleceń od „klientów". W przypadku, gdy klient jest agentem, zlecenie i odpowiedź na to zlecenie będą wiadomościami ACL. Naturalnym wydało mi się, więc zastosowanie implementacji protokołu interakcji FIPA-request (diagram protokołu - rysunek 6.1) będącej częścią platformy JADE.

Protokół ten pozwala agentowi na zlecenie innemu agentowi wykonania pewnej akcji. Partner[1] przetwarza zlecenie i decyduje o jego przyjęciu (akt komunikacji FIPA-agree) lub odrzuceniu (akt komunikacji FIPA-refuse). W przypadku przyjęcia zlecenia, Partner musi udzielić jednej z odpowiedzi:

- porażka - poprzez akt komunikacji FIPA-failure
- zawiadomienie o wykonaniu zadania - poprzez akt komunikacji FIPA-inform-done
- przekazanie wyników działania - poprzez akt komunikacji FIPA-inform-result

Implementacja protokołu rozszerza klasę zachowania agenta (`Behaviour`). Agent-Kontroler udostępnia swoje usługi poprzez dodanie zachowania, które jest pochodną klasy `AchieveREResponder` (będącej implementacją Partnera[1]). Agent, chcąc skorzystać z usług Agenta-Kontrolera, zmuszony jest przestrzegać protokołu interakcji poprzez dodanie zachowania[2]rozpoczynającego konwersację (pochodna klasy `AchieveREInitiator`).

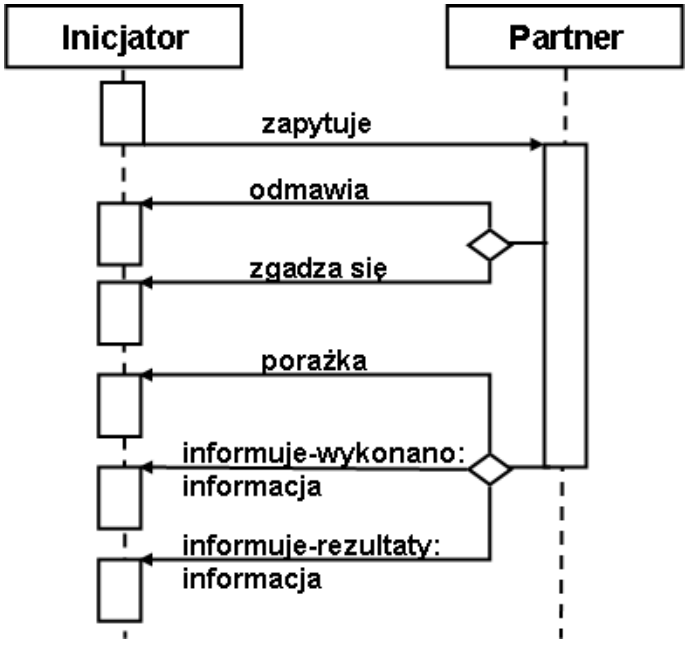
Klasy `AchieveREResponder` można użyć na dwa sposoby:

- Przeciążenie metod wywoływanych przy nadejściu zlecenia od klienta.
- Zarejestrowanie zachowania obsługującego zdarzenie nadejścia takiego zlecenia.

[1] ang. responder

[1] patrz rozdział 4.2.4

[2] Dodanie zachowania oznacza dodanie do zbioru zachowań agenta instancji klasy dziedziczącej z klasy `Behaviour`, która implementuje to zachowanie.



Rysunek 6.1. Diagram protokołu interakcji FIPA-reqest

opracowano na podstawie [?]

Rozwiązanie pierwsze jest łatwe w implementacji i wystarczające w większości przypadków. Rozwiązanie drugie wymaga nieco większego wkładu programisty jednakże posiada dwie zalety poprawiające jakość powstałego kodu. Po pierwsze, w przypadku zajścia potrzeby zmienienia sposobu obsługi określonego zdarzenia, wystarczy podmienić klasę implementującą obsługę tego zdarzenia bez potrzeby ingerencji w kod klasy zachowania. Po drugie, platforma JADE dostarcza zachowanie wspomagające `HandlerSelector`, które można zarejestrować do obsługi zdarzenia nadejścia zlecenia klienta. Działanie tego zachowania polega na wyborze właściwego, wcześniej zarejestrowanego zachowania obsługującego to zdarzenie, w oparciu o dane zawarte w nadchodzącej wiadomości ACL. Widoczna jest, zatem analogia z naturą `Kontrolera`, który dysponuje zbiorem delegatów (`RequestHandlers`), którzy w jego imieniu wykonują powierzone mu zadanie w zależności od nadchodzącego zapytania. Użycie rozwiązania drugiego pozwala na elastyczną, bezingerencyjną rozbudowę systemu o nowe typy zleceń (i w konsekwencji nowych delegatów).

6.1.1. Obsługa wyjątków

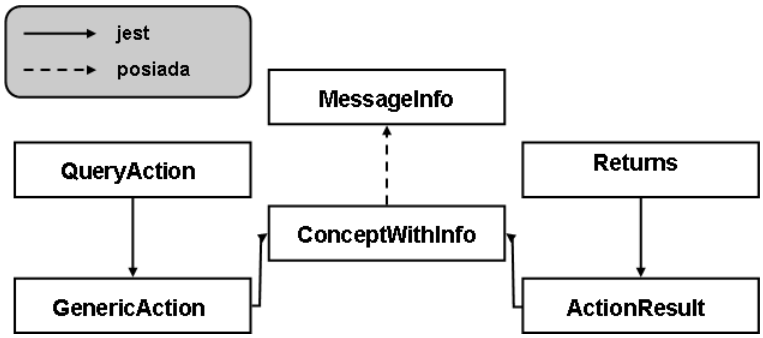
Implementacja Inicjatora dialogu w klasie `AchieveREInitializer` pozwala na wygodną obsługę odpowiedzi Partnera, w tym, na obsługę aktów komunikacji informujących o odmowie wykonania zadania (FIPA-refuse) lub o porażce wykonania zadania (FIPA-failure). Istnieje możliwość przeciążenia odpowiedniej metody obsługującej zdarzenie lub rejestracji zachowania obsługującego to zdarzenie. Pozwala to na wygodną i elastyczną obsługę sytuacji wyjątkowych, występujących podczas wykonywania operacji przez innego agenta.

6.2. Wiadomości w systemie

Elementem systemu mającym bezpośrednią styczność ze światem agentowym i nie-agentowym jest oczywiście warstwa pośrednicząca pomiędzy protokołem klienckim a Agentem Personalnym i resztą systemu. W miejscu tym zachodzi zamiana wiadomości pochodzącej z zewnątrz systemu na wiadomość systemową[3], tak aby PA mógł ją zrozumieć. Fundamentalnym okazało się zatem pojęcie wiadomości w systemie.

Zalecanym sposobem ustanowienia kontekstu rozmowy pomiędzy agentami jest ustalenie wspólnej ontologii[4]. Tym sposobem, agenci są w stanie wymieniać zrozumiałe informacje jak na przykład zdania (prawdziwe lub fałszywe stwierdzenia) lub polecenia wykonania pewnej akcji.

Ze względu na powyższe, stworzyłem ontologię *System-Ontology* (rysunek 6.2). Wiadomości



Rysunek 6.2. Diagram UML klas budujących ontologię *System-Ontology*

wewnątrz systemu muszą być zaopatrzone w informacje związane z sesją klienta (numer sesji, typ protokoły klienckiego, nazwa zleconej operacji, nazwa użytkownika). Funkcję nośnika tej informacji pełni koncept (klasa) `MessageInfo`. Klasa `ConceptWithInfo` jest konceptem posiadającym referencję do klasy `MessageInfo`. Klasa ta jest nadrzędna w stosunku do klas:

- `GenericAction` - klasa odpowiadająca konceptowi ogólnej akcji z bliżej nieokreślonymi danymi wejściowymi
- `QueryAction` - klasa odpowiadająca konceptowi akcji zapytującej, której danymi wejściowymi są parametry zapytania w postaci klucz-wartość
- `ActionResult` - klasa odpowiadająca konceptowi rezultatu wykonania akcji

Referencję do klasy `ActionResult` posiada klasa `Returns`, która odpowiada pojęciu predykatu stwierdzającego wartość wyniku akcji (`ActionResult`).

W kontekście zagadnienia komponentów typu Agent-Kontroler, używających protokołów interakcji FIPA-request, instancje klas `GenericAction` oraz `QueryAction` służą jako nośnik wiadomości zlecających pewne zadanie innemu agentowi (akt komunikacji typu FIPA-request) zaś instancja klasy `ActionResult` służy jako odpowiedź agenta w kończącym protokół interakcji akcie komunikacji FIPA-inform lub FIPA-failure (w przypadku porażki wykonania zadania).

[3] wiadomość ACL
[4] patrz rozdział 4.3

6.3. Komponenty systemu

6.3.1. Agent-Proxy

Agent-Proxy to ogólna nazwa dla komponentu oddzielającego i pośredniczącego pomiędzy specyficznym protokółem klienckim a PA. Z klasy tego agenta, dziedziczą wszelkie moduły nasłuchujące znajdującego się po stronie serwera. Rola tego komponentu to między innymi:

- Udostępnienie kanału komunikacji dla pewnego protokołu (HTML, WML, e-mail, etc.).
- Nadanie identyfikatora sesji przychodzącym zapytaniom klientów. (Powinna go cechować unikatowość w obrębie całego systemu.)

Implementacja pierwszej funkcji pozostawiona jest deweloperom i pozwala na zupełną dowolność w doborze metod. Zadanie to wymaga nierzadko użycia finezyjnych technik by pokonać brak wsparcia technologicznego dla komunikacji typu agent i nie-agent (rozdział 2). W przypadku implementacji modułów nasłuchujących dla protokołów HTTP/HTML oraz HTTP/WML stworzyłem bardzo lekki, współbieżny serwer HTTP. Serwer ten zaczyna działanie wraz z działaniem Agenta-Proxy (PrA). Serwer ten przyjmuje w konstruktorze instancję pewnej klasy wewnętrznej PrA, która udostępnia, poprzez interfejs, funkcjonalność potrzebną do przekazania zlecenia klienta i odebrania odpowiedzi systemu. Instancja tej klasy jest przekazywana wątkom serwera. W przypadku nadejścia żądania HTTP, uruchamiany jest nowy wątek serwera, z nagłówka HTTP ekstrahowane są odpowiednie parametry, tworzona jest instancja `ExternalRequest` (wiadomość z zewnątrz systemu) i przekazywana jest do odpowiedniej metody instancji klasy wewnętrznej Agenta-Proxy. Za pomocą tej instancji tworzona jest instancja konceptu akcji `QueryAction`[5], która służy jako treść wiadomości przesyłanej do wnętrza systemu. Wątek w następnej kolejności czeka (nie zamykając połączenia) do momentu nadejścia odpowiedzi systemu. Przy pomocy identyfikatora sesji Agent-Proxy przekazuje odpowiedź odpowiedniemu wątkowi, który po dostarczeniu jej klientowi kończy swój żywot.

Ze względu na unikatowość imion agentów w platformie, kolejne identyfikatory powstają poprzez konkatenacje imienia Agenta-Proxy i wygenerowanej przez agenta liczby.

6.3.2. Personal Agent

Agent Personalny (ang. Personal Agent) jest centralnym elementem systemu. Z konceptualnego punktu widzenia, to on jest reprezentantem użytkownika, który ma do swojej dyspozycji zbiór komponentów - agentów funkcjonalnych. Z punktu widzenia architektury może być on jednakże postrzegany jako kolejny komponent o nieco szerszych obowiązkach. Agent ten przyjmuje zlecenie od modułu nasłuchującego, wykonuje je przy pomocy dostępnych środków i zwraca wynik. Z tego względu, agent ten to także Agent-Kontroler.

Funkcje wykonywane przez Agenta Personalnego są rozległe. Na jedno zlecenie klienta może składać się konsultacja z kilkoma agentami funkcjonalnymi. Agent ten powinien wydobyć spersonalizowaną informację, dostępną poprzez komponenty systemu, oraz przedstawić ją w formie czytelnej dla protokołu użytkownika. Powyższe stwierdzenie sugeruje dwie klasy dzielące funkcjonalność agenta personalnego:

- Pozyskiwanie danych z systemu.
- Transformowanie tych danych do formy czytelnej dla protokołu użytkownika.

[5]Koncept pochodzi z ontologii *System-Ontology* opisanej wcześniej w tym rozdziale.

Do implementacji PA użyłem więc wzorca projektowego Model-Widok-Kontroler (rozdział 5.4.2). W warstwie Modelu znajdują się wszelkie komponenty odzyskujące dane (*DB Agent*), zaś w warstwie Widoku znajduje się agent TA (Transforming Agent), którego zadaniem jest obróbka rezultatów działań warstwy Modelu.

Każdy agent, z wyjątkiem PrA, w budowanym przeze mnie podsystemie to Agent-Kontroler. Oznacza to, że PA jest jednocześnie Partnerem (w dialogu z PrA) i Inicjatorem w interakcjach z DBA (Database Agent) oraz z TA. Agent ten, do obsługi nadchodzącego żądania, rejestruje instancję zaimplementowanego przeze mnie zachowania `MVC`[6], dziedziczącego z `SequentionalBehaviour` platformy JADE. Zachowanie `SequentionalBehaviour` pozwala na rejestrację podzachowań[7], które są wykonywane sekwencyjnie. Zachowanie `MVC`, w pierwszej kolejności rejestruje zachowanie inicjujące dialog z DBA, w drugiej zaś, rejestruje zachowanie inicjujące dialog z TA. Odpowiedź DBA przekazywana jest jako wiadomość inicjująca dialog PA z TA. Odpowiedź TA stanowi odpowiedź PA na wcześniej rozpoczęty dialog pomiędzy PrA i PA.

6.3.3. Agent „Bazodanowy"

Agent ten to kolejny agent funkcjonalny, udostępniający swoje usługi poprzez dodanie odpowiedniego zachowania Partnera. Klient tego komponentu, musi postępować zgodnie z protokołem interakcji, na przykład poprzez dodanie zachowania inicjującego dialog. Zapytanie klienta powinno zawierać parametry zapytania w postaci klucz-wartość. DBA, za pomocą bibliotek pomocniczych, na podstawie tych parametrów generuje zapytanie w języku RDQL [**?**] i zapytuje bazę danych przy użyciu platformy JENA. Otrzymany zbiór trójek RDF/XML opakowywany jest wiadomością ACL i zwracany jako odpowiedź[8] do Inicjatora konwersacji.

6.3.4. Agent Transformator

Agent Transformator (ang. Transforming Agent) należy do warstwy Widoku i wdraża zachowanie implementujące Partnera protokołu interakcji FIPA-request. Agent ten, wykonując zadanie, bierze pod uwagę nazwę wykonywanej operacji (zlecenia) oraz typ klienckiego protokołu (HTML, WML, etc.). Na podstawie tych parametrów wyznacza odpowiedniego delegata, którego zadaniem jest wykonanie właściwej pracy. Agent ten to typowy przykład agenta funkcjonalnego, którego zadanie rzadko wymagać będzie interakcji z pozostałymi komponentami agentowymi.

6.3.5. Transformacja RDF/XML

Agent Transformator to typowy agent funkcjonalny. Jego interakcje z innymi agentami zależą od specyfiki typu zachowania wybranego do obsługi zadania zlecanego przez Agenta Personalnego. Cechą wspólną wszystkich zleceń jest typ danych przeznaczonych do konwersji, którym jest RDF/XML.

Ogólny opis metody

Rozwiązanie, które pozwoliło mi na rozwiązanie problemu transformacji danych w formacie RDF/XML na formę czytelną dla protokołu klienta składa się z odpowiednio skonfiguro-

[6] MVC (Model-View-Controller)

[7] wolne tłumaczenie słowa „subbehaviour", które pojawia się w dokumentacji JADE i oznacza zachowanie wykonujące się w ramach innego zachowania

[8] akt komunikacji typu FIPA-inform

wanego serwera Raccoon oraz ze zbioru dokumentów RxSLT transformujących model RDF. TA po otrzymaniu zlecenia (zbiór „trójek" w języku RDF), ekstrahuje dane RDF/XML i zapisuje je w pliku o arbitralnej nazwie (dla ustalenia uwagi nazwę ten plik „dane.rdf") w przestrzeni dyskowej poszukiwań serwera Raccoon. Następnie, z zapytania przesłanego przez PA, ekstrahowane są parametry: nazwa operacji (dla ustalenia uwagi niech będzie to „get-data-operation") oraz typ protokołu klienta (dla ustalenia uwagi niech będzie to „html"). Parametry te używane są przez TA do stworzenia następującego adresu URL:

```
,,url serwera Raccoon''/dane.rdf?operation-name=get-data-operation&protocol=html
```

Serwer Raccoon, w oparciu o metadane zapytania „operation-name" oraz „protocol", wybiera odpowiedni dokument zawierający transformacje w języku RxSLT i stosuje go do pliku „dane.rdf", zwracając przetransformowaną zawartość RDF/XML. Wyniki działania Raccoon są skolekcjonowane przez TA i wysłane do zleceniodawcy (czyli PA).

Przykład transformacji

Założę, że TA otrzymał zlecenie od PA (parametry jak wyżej) zawierającą następującą treść:

```
<j.0:Restaurant rdf:about="http://www.agentlab.net/db/restaurant-db
   #Poland_DS_Brzeznik_Pod_Roza__Restauracja1011069562">
  <j.1:country>Poland</j.1:country>
  <j.0:title>Pod Roza, Restauracja</j.0:title>
  <j.1:city>Brzeznik</j.1:city>
  <j.1:streetAddress>Brzeznik, (p-ta Boleslawiec)</j.1
     :streetAddress>
  <j.1:phone>+48 (75) 732 65 00 </j.1:phone>
  <j.1:zip>59-700</j.1:zip>
  <j.0:locationPath>Poland/DS/Brzeznik</j.0:locationPath>
  <j.1:state>DS</j.1:state>
</j.0:Restaurant>
```

Instancja tej restauracji posiada między innymi własności jak:

- Kraj: Polska
- Nazwa: Restauracja Pod Różą
- Miasto: Brzeźnik

Praca wykonywana przez wyznaczony do transformacji dokument RxSLT polega na wyszukaniu w transformowanym dokumencie wszystkich zasobów zidentyfikowanych (poprzez URI) jako restauracje, a następnie odpowiednie sformatowanie własności tego zasobu w postaci: „predykat" - „obiekt". W efekcie otrzymujemy (fragment w przypadku języka HTML):

```
<tr>
  <td style="background-color:orange">
                <p style="font-size:20">
                        Pod Roza, Restauracja
                </p>
  </td>
</tr>
<tr>
  <td><B>city</B></td>
```

```
  <td>Brzeznik</td>
</tr>
<tr>
        <td><B>country</B></td>
        <td>Poland</td>
</tr>
<tr>
  <td><B>phone</B></td>
  <td>+48 (75) 732 65 00 </td>
</tr>
<tr>
  <td><B>state</B></td>
  <td>DS</td>
</tr>
<tr>
  <td><B>streetAddress</B></td>
  <td>Brzeznik, (p-ta Boleslawiec)</td>
</tr>
<tr>
  <td><B>zip</B></td>
  <td>59-700</td>
</tr>
<tr>
  <td><B>locationPath</B></td>
  <td>Poland/DS/Brzeznik</td>
</tr>
<tr>
  <td><B>type</B></td>
  <td>Restaurant</td>
</tr>
```

6.3.6. Metoda generowania formularza zapytującego

W poniższym rozdziale, terminem „cecha" będę określał własność restauracji, zaś terminem „własność" będę określał własność cechy restauracji. W omawianym Systemie Wspomagania Podróży zachodzi potrzeba każdorazowego generowania formularza zapytującego na podstawie ontologii systemu opisującej klasę restauracji[9]. Klasa restauracji, składa się z pewnej liczby opisów cech (`rdf:Property`) restauracji. Opisy cech restauracji o nazwie „cuisine" (rodzaj kuchni) oraz „city" (miasto) ma następującą postać:

```
<rdf:Property rdf:about="http://www.agentlab.net/schemas/restaurant
   #cuisine">
  <comment>The type of food a restaurant serves. We repeat this
     field up to three times.</comment>
  <domain rdf:resource="http://www.agentlab.net/schemas/restaurant#
     Restaurant"/>
  <label>cuisine</label>
  <range rdf:resource="http://www.agentlab.net/schemas/restaurant#
     CuisineCode"/>
</rdf:Property>

<rdf:Property rdf:about="http://www.agentlab.net/schemas/location#
   city">
```

[9]patrz rozdział 1.2.6

```
    <domain rdf:resource="http://www.agentlab.net/schemas/location#
       Location"/>
    <label>city</label>
    <range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
```

Widoczne są następujące własności:

- *comment* - komentarz wyjaśniający funkcję własności
- *domain* - przynależność do klasy restauracji
- *label* - etykieta własności
- *range* - określa dopuszczalne wartości dla tej własności

Podczas procesu generowania formularza wziąłem pod uwagę wszystkie cechy restauracji oraz ich własności: *comment*, *label* oraz *range*. Najbardziej problematyczna okazała się własność *range*, gdyż określa ona dopuszczalne wartości i dlatego determinuje rodzaj formatki użytej w formularzu.

W danej ontologii restauracji użyto dwóch typów opisu wartości poprzez własność *range*:

- wyspecyfikowanie typu oczekiwanych danych (przykład, cecha „*city*")
- wymienienie wprost wszystkich dostępnych wartości dla tej własności (przykład, cecha „*cuisine*")

W pierwszym przypadku używa się typów określonych w specyfikacji XML Scheme [**?**] do których należą między innymi:

- `http://www.w3.org/2001/XMLSchema#int`- liczba całkowita
- `http://www.w3.org/2001/XMLSchema#String`- ciąg znaków
- `http://www.w3.org/2001/XMLSchema#nonNegativeInteger`- nieujemna liczba całkowita
- `http://www.w3.org/2001/XMLSchema#float`- liczba rzeczywista

Do reprezentacji własności o tak określonym typie wartości zastosowałem pole tekstowe. Nazwę formatki określa URI cechy. Walidacja danych wprowadzonych do formularza odbywa się poprzez wywołania odpowiednich funkcji JavaScript. Cecha „city" generuje następującą formatkę w języku HTML:

```
<input name="http://www.agentlab.net/schemas/location#city"
        type="text">
```

W drugim przypadku, konkretne wartości opisywane są przez instancje pewnej określonej klasy. Przykład cechy „*cuisine*" ilustruje mechanizm przypisywania tego typu wartości. Wartość własności „*range*" to URI:

```
http://www.agentlab.net/schemas/restaurant#CuisineCode,
```

które identyfikuje pewną klasę „*CuisineCode*":

```
<Class rdf:about="http://www.agentlab.net/schemas/restaurant#
   CuisineCode">
        <Comment>cuisine</Comment>
</Class>
```

Instancje tej klasy to dopuszczalne wartości dla tej własności:

```
<CuisineCode xmlns="http://www.agentlab.net/schemas/restaurant#"
   rdf:about="http://www.agentlab.net/schemas/restaurant#
   AfghanCuisine">
  <rdfs:label>Afghan</rdfs:label>
</CuisineCode>

<CuisineCode xmlns="http://www.agentlab.net/schemas/restaurant#"
  rdf:about="http://www.agentlab.net/schemas/restaurant#
     AfricanCuisine">
  <rdfs:label>African</rdfs:label>
</CuisineCode>
```

Do reprezentacji tego typu cech użyłem list wyboru. Cecha „cuisine" generuje następującą formatkę w języku HTML:

```
<select name="http://www.agentlab.net/schemas/restaurant#cuisine">
        <option selected="selected">-select-</option>
        <option value="AfghanCuisine">Afghan</option>
        <option value="AfricanCuisine">African</option>
        <option value="AmericanCuisine">American</option>
        ...
</select>
```

Nazwę formatki stanowi URI cechy, zaś wartości atrybutów „*value*" poszczególnych możliwości wyboru stanowią nazwy lokalne identyfikatorów URI instancji klasy „*CuisineCode*".

6.4. Agent na urządzeniu przenośnym

W podrozdziale tym, wbrew założeniom budowanego systemu, rozważam przypadek, gdy instalacja dodatkowego oprogramowania na urządzeniu przenośnym klienta jest możliwa.

Pomimo trudności z integracją środowisk występujących na urządzeniach przenośnych i tych na serwerach[10] udało mi się wykorzystać istniejący dorobek projektu JADE-LEAP do zaprojektowania i zaimplementowania agenta zdolnego do egzystencji na telefonie komórkowym[11], obsługującym J2ME. Agent ten posiada wystarczające zdolności operacyjne do komunikowania się z istniejącą infrastrukturą agentową systemu i do bezpośredniego komunikowania się z użytkownikiem.

W początkowej fazie projektu, owym agentem rezydującym na telefonie był PA, zbliżając to rozwiązanie do ideału, w którym zachodzi bezpośrednia interakcja klient - Agent Personalny. Rozwiązanie to posiadało jednakże dwie zasadnicze wady. Po pierwsze, przyszły rozwój systemu ma w planie intensywną rozbudowę funkcjonalności Agenta Personalnego z użyciem technik wymagających zasobów o dużej mocy (wielowątkowość, refleksja, etc.), których obecnie nie wspiera technologia J2ME. Po drugie, Agent Personalny, w istniejącej strukturze pełni kluczową rolę w zarządzaniu obiegiem informacji podczas wykonywania zlecenia użytkownika. Faktem jest także, że agent ten rozproszył swą funkcjonalność pomiędzy kilku agentów pomocniczych (PIAs, DBA, TA). Zatem użycie tego kluczowego komponentu w tak niestabilnym środowisku, jakim jest telefon komórkowy (nagłe zerwanie połączenia sieciowego, słaba bateria zasilająca) nie może być brane pod uwagę.

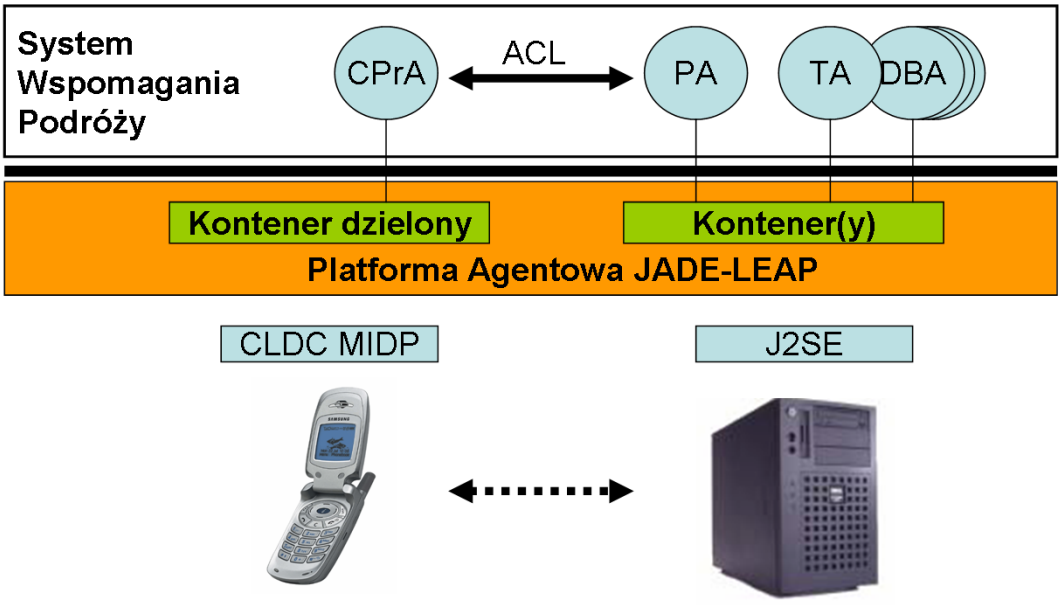
[10] patrz rozdział 2

[11] w warunkach symulacji na komputerze stacjonarnym

Ze względu na powyższe, postanowiłem umieścić na telefonie agenta zbliżonego swoją funkcjonalnością do PrA - Agenta Proxy Komórkowego[12] (*Cell Proxy Agent* - CPrA). Agent ten odpowiada za:

- przyjmowanie i wysyłanie wiadomości do Agenta Personalnego
- generowanie interfejsu użytkownika na podstawie danych otrzymanych od PA (formularzy, odpowiedzi od systemu)
- obsługę procesu interakcji użytkownika z wygenerowanym interfejsem

Architekturę rozwiązania obrazuje rysunek 6.3.



Rysunek 6.3. Schemat architektury rozwiązania uwzględniającego agenta na telefonie komórkowym

Należy zauważyć, że agent na telefonie komórkowym i Agent Personalny komunikują się ze bez użycia oprogramowania nie-agentowego. Agenta tego można zatem uznać za klienta bardziej „wyszukanego" o którym wspomniałem w rozdziale 1.2.

6.4.1. Dynamiczne generowanie interfejsu użytkownika

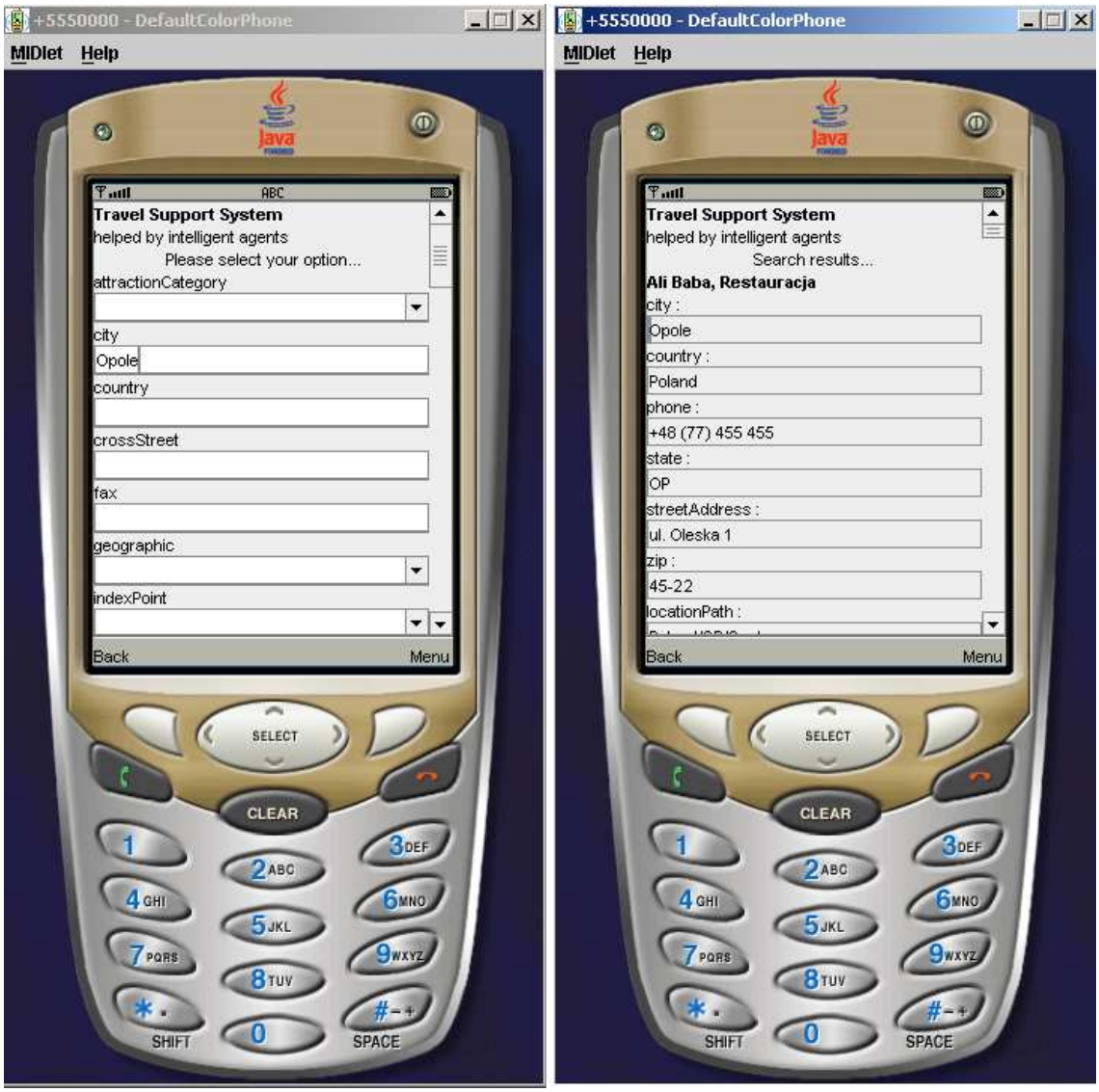
CPrA pełni rolę Agenta-Proxy i przeglądarki sieciowej, ponieważ oprócz obsługi komunikacji z PA musi on być w stanie generować interfejs użytkownika (ekran logowania, formularz zapytujący, etc.). CPrA jest inicjatorem aktywności (jak przeglądarka sieciowa). Wysyła prośbę o wykonanie pewnej akcji by następnie otrzymać odpowiedź na podstawie której możliwe będzie wysłanie prośby o kolejną akcję. Taka sekwencja to na przykład:

1. Uruchomienie agenta CPrA i związane z tym wysłanie prośby o wykonanie akcji zwracającej ekran powitalny.
2. Ekran powitalny umożliwia zalogowanie się do systemu po wprowadzeniu nazwy użytkownika i hasła - możliwa akcja związana z tym ekranem (po pomyślnym zalogowaniu) zwraca ekran, zawierający selekcję usług dostarczanych przez System.

[12] patrz rozdział 6.3.1

3. Użytkownik - wybiera przykładowo: „znajdź restaurację”, co powoduje wysłanie prośby o wykonanie akcji zwracającej formularz zapytujący

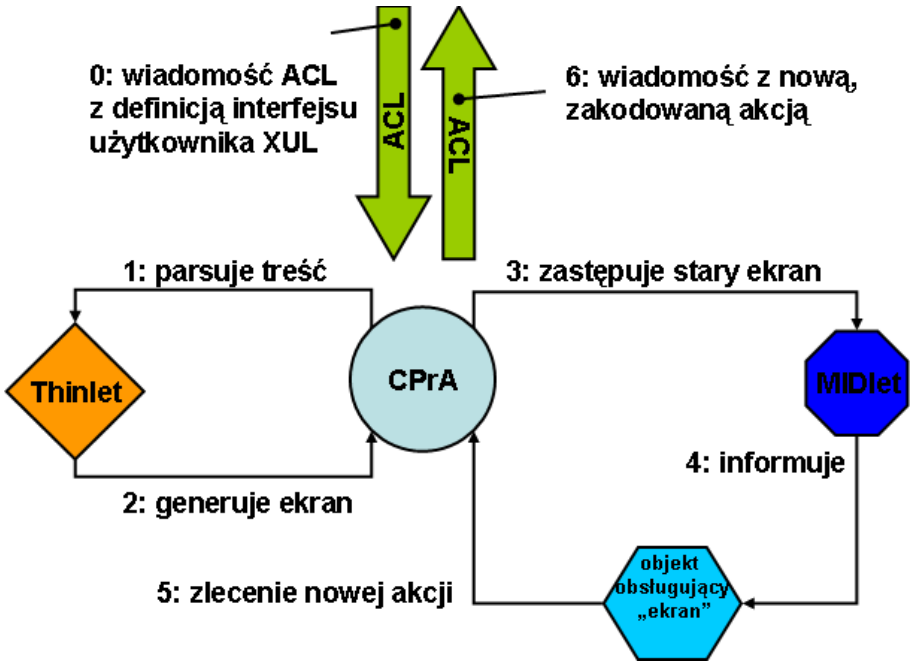
4. ... etc.



Rysunek 6.4. Zrzuty ekranu obrazujące efekty działania agenta CPrA. Lewe okno przedstawia wygenerowany formularz. Prawe okno zawiera przykładowe wyniki wyszukiwania restauracji. Symulacja środowiska telefonu komórkowego została przeprowadzona przy użyciu narzędzia J2ME Wireless Toolkit 2.2 (http://java.sun.com/products/sjwtoolkit/)

Moje rozwiązanie zagadnienia dynamicznego generowania interfejsu na telefonie komórkowym, polega na użyciu technologii Thinlet[13] (implementacja dla J2ME), która umożliwia dynamicznie generowanie interfejsu użytkownika na podstawie jego definicji w języku XUL[14]. Język XUL traktowany jest w systemie jako jeden z możliwych do użycia przez urządze-

[13] patrz rozdział 4.5.1
[14] patrz rozdział 4.5



Rysunek 6.5. Sekwencja operacji wykonywanych w CPrA po otrzymaniu wiadomości od Systemu

nie klienta język znacznikowy. Oznacza to, że TA posiada funkcjonalność pozwalającą na przekształcenie treści RDF/XML na XUL.

Każdy agent na platformie JADE-MIDP posiada referencję do instancji MIDlet, która z kolei dostarcza funkcjonalność do manipulacji zawartości ekranu telefonu komórkowego. CPrA wymienia wiadomości ACL z systemem. Wyekstrahowaną treść wiadomości parsuje odpowiednia biblioteka Thinlet, by na jej podstawie wygenerować interfejs (za pomocą kontrolek dostarczanych przez MIDP). Instancja interfejsu zastępuje stary ekran MIDlet.

Obsługa wygenerowanego interfejsu użytkownika

W systemie każda wiadomość zaopatrzona jest w informację, mówiącą jakiego rodzaju akcji dotyczy (prośba o formularz, zapytanie o restauracje, logowanie do systemu). Użyte przeze mnie rozwiązanie zakłada istnienie skończonej, ustalonej ilości rodzajów akcji CPrA do systemu. Dla każdego rodzaju akcji należy zdefiniować wyspecjalizowaną klasę zdolną do obsługi danego ekranu. Dla przykładu formularz zapytujący o restauracje posiada przycisk „Query”, który wysyła zawartość formularza do PA. Zanim to nastąpi, instancja klasy obsługującej ten rodzaj ekranu, musi wyekstrahować wartości z formularza i na ich podstawie stworzyć wiadomość ACL z treścią zapytania stanowiącą selekcję użytkownika (rysunek 6.5 ilustruje sekwencję).

Zakończenie

Agent to jednostka programowa, która wyrosła z domeny sztucznej inteligencji i już w pierwotnym założeniu miała ona wspierać użytkownika w interakcji z komputerem, siecią Internet lub też inną, bardziej konkretną aplikacją. Trudno więc przecenić wagę kwestii sprawnej komunikacji pomiędzy klientem a agentem. Niestety, wbrew oczekiwaniom Pattie Maes [**?**] czy Christine Guilfoyle i Ellie Warner [**?**], technologia agentowa nie wzniosła się na pułap dorównujący swym teoretycznym aspiracjom i oczekiwaniom. Nie istnieje bowiem dobrze rozwinięta technologia oparta o oprogramowanie agentowe. Agenci nie wspierają nas przy przeszukiwaniu zasobów Internetu, nie zarządzają naszymi transakcjami finansowymi, nie pomagają w podróży.

Niniejsza praca rozważa takie sposoby komunikacji użytkownika z agentem programowym, które na dzień dzisiejszy umożliwiają ich wdrożenie w konkretny, działający system. Odrzuciłem w ten sposób alternatywę, prowadzącą do zakończenia obfitującego w kolejne teoretyczne wywody, które w „niedalekiej" przyszłości mogłyby przynieść pożytek. Postanowiłem skorzystać z dorobku technologii agentowej i zaimplementować część klienckiej strony Systemu Wspomagania Podróży [**?**] w razie potrzeby wspierając się szerokim wachlarzem najnowszych, nie-agentowych technologii (Java, J2ME, JADE, LEAP, XUL, RDF, Raccoon).

System Wspomagania Podróży składa się z kilku komponentów - każdy zawiera kolektyw współdziałających ze sobą agentów. Źródłem „wiedzy" systemu jest repozytorium danych oparte o technologię Semantycznego Internetu (RDF). Użytkownik kontaktuje się ze swoim Agentem Personalnym, który dostarcza mu spersonalizowane informacje. Klienci rozważanego systemu to podróżni, często wyposażeni w urządzenia przenośne typu telefon komórkowy czy PDA.

W czasie prac nad projektem wyłoniły sie dwie podstawowe kwestie. Pierwsza, to ograniczenia dystrybucji agentów na urządzenia przenośne. Druga, to metoda dostarczania wiadomości od użytkownika do agenta (dane wejściowe) i od agenta do użytkownika (rezultaty działania).

Ograniczona pamięć operacyjna jak i moc obliczenia telefonów komórkowych nie pozwoliły na funkcjonowanie wielowątkowowego Agenta Personalnego. Zmusiło mnie to do przeniesienia go na stronę serwera, co z kolei utrudniło wymianę wiadomości z klientem. Agent obecny w komputerze użytkownika (lub telefonie komórkowym) mógłby generować pewien graficzny interfejs użytkownika, za pomocą którego mogłyby zachodzić wzajemne interakcje. Obejściem tego problemu okazało się użycie protokołu HTTP do przesyłania danych i wybranego języka znacznikowego (HTML, WML, XML, etc.) do prezentacji danych. W rozwiązaniu tym konieczna była obecność elementu pośredniczącego (Agent Proxy), który tłumaczyłby wiadomości w „języku" HTTP na język ACL[15] i odwrotnie. Należało też rozwiązać problem pogodzenia synchronicznej natury protokołu HTTP (zapytanie-odpowiedź) i asynchronicznej natury działania systemu wieloagentowego (wymiana wiadomości).

Kolejnym krokiem okazało się opracowanie metody prezentacji użytkownikowi wyników działania systemu. Repozytorium systemu składa się z danych opisanych w języku RDF według modelu danych opisanego w języku RDF Scheme. Użyte przeze mnie rozwiązanie objęło użycie serwera Raccoon do transformacji semantycznych danych w języku RDF na postać czytelną dla przeglądarki klienta. Wymagało to ode mnie sporządzenia dokumentów

[15] wiadomości zrozumiałe przez agentów zgodnych ze specyfikacją FIPA

w języku RxSLT[16] przekształcających dane w formacie RDF/XML na jeden z języków znacznikowych (HTML, WML, XUL, etc.). Ponadto, przy użyciu technologii XUL/Thinlet i J2ME udało mi się umieścić prostego agenta na telefonie komórkowym (z obsługą języka Java), który komunikuje się bezpośrednio z Agentem Personalnym i z użytkownikiem.

Rozpoczęta przez mnie implementacja części klienckiej strony Systemu Wspomagania Podróży musiała uwzględniać przyszły rozwój systemu. Z tego względu, posłużyłem się dobrze znanymi technikami programowania takimi jak wzorce projektowe [**?**] czy metodologia tworzenia systemów wieloagentowych Promethues [**?**]. W efekcie, powstał elastyczny szkielet aplikacji o otwartej architekturze, umożliwiający jego sprawną rozbudowę.

Przyszły rozwój systemu może przykładowo obejmować stworzenie dodatkowego agenta funkcjonalnego, jak Agent Personalizujący. Dzięki użyciu wzorców projektowych oraz protokołów interakcji międzyagentowych FIPA, poszczególne komponenty funkcjonalne Podsystemu Dostarczania Informacji są wyspecjalizowane i dobrze odseparowane (nie są „świadome" operacji wykonywanych w innych komponentach). W konsekwencji, dodanie kolejnego agenta wymagać będzie jedynie minimalnego nakładu programisty, ograniczającego się do zdefiniowania sposobu interakcji z istniejącymi komponentami agentowymi (poprzez implementacje klas określających zachowanie agentów). Co więcej, rozwiązanie to nie wymaga zmian w implementacjach już istniejących agentów.

Kolejnym elementem rozbudowującym system będzie z pewnością dodanie mechanizmu obsługi sesji. Sesją, w przypadku serwisów sieciowych, nazywa się trwałe (w sensie logicznym) połączenie z klientem. W przypadku protokołu HTTP klient jest fizycznie rozłączany po każdym zapytaniu do serwera. Dlatego też informacje o „stanie" klienta przekazywane są (po stronie serwera) pomiędzy kolejnymi zapytaniami. Technika ta jest powszechnie stosowana, gdyż umożliwia ona rejestrację momentu rozpoczęcia świadczenia usługi, a także procesu jej trwania oraz momentu zakończenia (ważne przy procesie rejestracji gestów użytkownika przez Agenta Personalnego). Dzięki użyciu ontologii jako kontekstu interakcji międzyagentowych mechanizm obsługi sesji może być zrealizowany poprzez proste rozszerzenie klas budujących tę ontologię o pole będące identyfikatorem aktualnej sesji. Tym sposobem, identyfikator sesji przekazywany jest w każdej przesyłanej wiadomości ACL umożliwiając agentom identyfikację użytkownika.

Rozszerzenie systemu może objąć dodanie nowej funkcjonalności do istniejących już agentów, na przykład poprzez dodanie obsługi nowego typu zapytań ze strony użytkownika. Ze względu na użycie `Agentów-Kontrolerów`[17] zadanie to sprowadzi się do implementacji „zachowań" w poszczególnych agentach (w odrębnych klasach) i zarejestrowania ich instancji jako delegatów do obsługi tych zadań.

Multimodalny interfejs użytkownika, z definicji, musi dostarczyć wielorakich sposobów interakcji. W pracy tej udało mi się wdrożyć interakcje z system agentowym poprzez formularz HTML (komputer biurowy), WML (telefon komórkowy z WAP) oraz XUL (telefon komórkowy z obsługą języka Java). Użycie innych kanałów komunikacji w implementowanym przeze mnie systemie wymaga jedynie znajomości sposobu przetłumaczenia zapytania na język ACL[18] oraz zdefiniowania dokumentu RxSLT transformującego treść odpowiedzi na czytelną dla urządzenia klienta postać.

6.5. Źródła

[16] dialekt XSLT

[17] adaptacja obiektowego wzorca projektowego `Kontroler`

[18] język wiadomości używany przez agentów zgodnych ze specyfikacją FIPA

Spis rysunków

Skorowidz