Rok akad. 2004/2005

POLITECHNIKA WARSZAWSKA
Wydział Matematyki i Nauk Informacyjnych

P R A C A   M A G I S T E R S K A

Wawrzyniec Hyska

„Systemy agentowe w technologii RDF"

Opiekun pracy
dr Marcin Paprzycki

WARSZAWA 2006

Spis Treści

SPIS TREŚCI **2**

WSTĘP **4**

ROZDZIAŁ 1. PODSTAWY TEORETYCZNE **6**

ROZDZIELENIE TREŚCI I FORMY 7

Potrzeba rozwarstwienia dokumentów *7*

Standard RDF *10*

Język RDQL *18*

Ontologie *20*

SYSTEMY WIELOAGENTOWE 22

Próba definicji, historia systemów agentowych *22*

Podział i Funkcjonalność systemów agentowych *23*

Struktura Systemów Agentowych *25*

Zalety i wady podejścia agentowego *26*

ROZDZIAŁ 2. OPIS CZĘŚCI PRAKTYCZNEJ **29**

PROJEKT AGENTLAB **30**

FUNKCJONALNOŚĆ SYSTEMU 30

ARCHITEKTURA SYSTEMU 30

DZIAŁANIE SYSTEMU 35

JAK TO SIĘ MA DO TURYSTYKI – TRAVEL SUPPORT SYSTEM 37

PODSYSTEM ROZSZERZENIA ODPOWIEDZI – PROJEKT „FUMBO” 38

Praca Koordynatora *41*

Praca agenta-pracownika *44*

Możliwości rozwoju projektu Fumbo *48*

Konfiguracja systemu Fumbo *49*

WYKORZYSTYWANE MODUŁY, BIBLIOTEKI, NARZĘDZIA 50

Programowanie *50*

Środowisko Agentowe JADE *51*

Biblioteka Jena *55*

Moduł logujący Log4J *56*

Dane *57*
TESTY SYSTEMU 58

PODSUMOWANIE 64

BIBLIOGRAFIA: 66

Wstęp

Celem niniejszej pracy jest zaprojektowanie, zaimplementowanie i opis pewnego podsystemu większego systemu agentowego działającego na danych w formacie RDF. Całościowy system jest platformą przetwarzania i wyszukiwania danych, która napisana jest w postaci „programów” agentowych. Zadaniem systemu jest umiejętne wyszukiwanie informacji według jej znaczenia, wyciąganie wniosków z danych i dostarczenie użytkownikowi systemu trafnie wyselekcjonowanych informacji. Załączona implementacja dotyczy wyszukiwania danych na tematy związane z turystyką – na przykład danych o połączeniach, lokalach gastronomicznych, imprezach kulturalnych itp.

Zaimplementowany został zbiór programów agentowych, którzy współpracując, wspomagają tradycyjne wyszukiwanie informacji. Agenci ci potrafią rozszerzyć odpowiedź przygotowaną na podstawie danych zgromadzonych w centralnej bazie danych o informacje nie wynikające wprost z zadanego pytania. Dzięki analizie zadawanych zapytań oraz uzyskanych odpowiedzi, potrafią oni dopytać źródło danych w taki sposób, aby uzyskane informacje jak najbardziej odpowiadały intencjom pytającego. W ten sposób użytkownik może uzyskać dane na temat obiektów, o które nie pytał bezpośrednio (być może nie miał o nich nawet pojęcia), ale które są w pewien sposób związane z tematem, który go interesuje i w związku z tym mogą się okazać dla niego przydatne.

W pierwszym rozdziale pracy opisane są teoretyczne podstawy formatu RDF i systemów agentowych. Na początku opisany jest rozwój formatów przechowywania danych w epoce Internetu. Opisując standard RDF i konsekwencje jego stosowania, wykazane zostaną korzyści płynące ze stosowania ontologii. Dzięki RDF, dane stają się bowiem zrozumiałe dla programów komputerowych; mogą być analizowane i wykorzystywane jako pewne (tzn. takie, którym można ufać) źródła informacji. Format RDF pozwala analizować dane pod kątem ich znaczenia, pod kątem logiki, które one zawierają. W drugiej części pierwszego rozdziału zaprezentowane jest agentowe podejście do tworzenia systemów. Zdefiniowani są tam agenci jako jednostki programowane, opisane są struktury takich systemów, wyliczone są pozytywne i negatywne aspekty takiego podejścia.

Rozdział drugi opisuje konkretną implementację systemu. Całościowy system jest przedmiotem pracy wielu studentów na różnych uczelniach i nosi nazwę platformy *AgentLab*. Tematem niniejszej pracy jest część tej platformy – projekt *Fumbo*. Zarówno platforma

AgentLab w całości, jak i podprojekt Fumbo zostały opisane w drugim rozdziale. Znajduje się tutaj szczegółowy opis działania systemu, a także jego architektura – podział na moduły, podział odpowiedzialności wśród agentów. Zaprezentowany jest projekt systemu oraz pokazane są pewne charakterystyczne szczegóły implementacji. W drugiej części rozdziału opisane są technologie wykorzystane do stworzenia projektu Fumbo – narzędzia programistyczne, narzędzia testujące, biblioteki. Dokładnie opisane jest także wykorzystywane w pracy środowisko agentowe Jade, a także specyfika pisania programów agentowych.

W podsumowaniu znajdują się wnioski do jakich doszedł autor w wyniku projektowania i implementacji systemu.

Rozdział 1.

Podstawy teoretyczne

Rozdzielenie treści i formy

Potrzeba rozwarstwienia dokumentów.

Ponad dziesięć lat temu świat wszedł w epokę Internetu. W Sieci lawinowo rosła i rośnie liczba dokumentów i danych przechowywanych na niezliczonych serwerach i w archiwach. Niepomiernie zwiększyła się dostępność do wszelkiego rodzaju danych; w tej chwili praktycznie każdy ma możliwość dotarcia do interesujących go jawnych (choć nie tylko) informacji. Ukuło się nawet powiedzenie, że czego nie ma w Internecie, to nie istnieje. Szczególnie aktualne i trafne wydaje się ono w odniesieniu do biznesu. Jeżeli firma nie istnieje w Internecie, jeżeli nie jest tam łatwo odnajdywana, to w chwili obecnej nie ma większych szans na rozwój.

Wraz ze wzrostem liczby danych i informacji przechowywanych na podłączonych do sieci serwerach, rosła potrzeba szybkiego ich przeszukiwania. Szybko pojawiły się wyszukiwarki internetowe, bez których żaden internauta nie wyobraża sobie dziś odnajdywania informacji. Początkowo, główną cechą wyszukiwarek, na jaką zwracano uwagę, była liczba zaindeksowanych stron, czyli liczba źródeł, które użytkownik mógł przeszukiwać. Szybko jednak źródeł pojawiło się tak dużo, że znacznie poważniejszym problemem stał się nadmiar informacji. Nie sztuką jest już znaleźć strony, które w jakiś sposób dotyczą interesującego nas zagadnienia; sztuką jest takie przefiltrowanie wyszukanych informacji, żeby pytający dostał dokładną odpowiedź na zadawane pytanie. Sztuką jest zbudowanie wyszukiwarki, która rozumie pytania użytkownika i zwraca mu wyniki, które najbardziej go ciekawią (i/lub które koniecznie powinien otrzymać). Zatem trudniejsze wydaje się przefiltrowanie dostępnych informacji tak aby pozostały te naprawdę związane z tematem, a równocześnie wyrzucenie wszystkiego co jest niepożądane.

Działanie pierwszych wyszukiwarek internetowych bazowało głównie na zliczaniu słów występujących w dokumentach i zbieraniu metadanych z nagłówków. Oczywiście algorytmy oparte o tek proste techniki okazały się zbyt prymitywne. Nie zawsze strony, na których słowo-klucz występuje najczęściej, najtrafniej odpowiadają zapytaniu użytkownika. Jeżeli zaś chodzi o informacje z nagłówków, znajomość algorytmów przeszukiwania szybko wykorzystywali webmasterzy, którzy w nagłówkach swoich stron zaczęli umieszczać

wszystkie najczęściej wyszukiwane przez użytkowników wyrazy (powodując przekłamanie wyników wyszukiwania informacji).

Krokiem naprzód była zmiana algorytmów wyszukiwania wprowadzona w wyszukiwarce Google [1] . Zastosowano tutaj (między innymi) ocenianie stron ze względu na liczbę odnośników do nich prowadzących. Jeżeli do strony prowadzi wiele odnośników, najpewniej zawiera ona interesujące i sprawdzone informacje. Wciąż jednak wyszukiwanie i poszukiwanie treści „o czym" jest dany dokument polega na indeksowaniu słów kluczowych. Dzisiejsze strony internetowe i dokumenty dostępne w sieci są wciąż zwyczajnymi „beztreściowymi" tekstami. Komputery potrafią zliczyć występowanie słów w dokumencie, nie potrafią jednak analizować jego treści. Nie mogą zatem wejść na semantyczny poziom czytania i przeszukiwania tekstów. Na przykład, wyszukiwarka jest w stanie się „zorientować", że słowo „kuchnia" pojawia się w kilkanaście razy w tekście strony; wciąż nie będzie miała jednak pewności, czy jest to artykuł na tematy kulinarne czy o urządzaniu wnętrz.

Problem leży między innymi w formie przechowywania danych na serwerach w Sieci. Początek szybkiego rozwoju Internetu zbiegł się w czasie (na pewno nie bez powodu) z upowszechnieniem usługi WWW [2] i języka HTML [3]. Język HTML był pomyślany jako język „ubioru" dokumentu w wygląd. Pozwala on podzielić tekst na akapity, wprowadzić listy i tabele, wklejać obrazki i dołączać odnośniki. Są to jednak wszystko zabiegi na formie dokumentu, na tym jak te dane będą wyglądały dla końcowego użytkownika, oglądającego je poprzez przeglądarkę internetową. Człowiek czytający taki sformatowany tekst będzie go rozumiał; dla maszyny jest to jednak wciąż zbiór liter, na dodatek teraz zawierający także informacje nie dotyczące treści, ale np. marginesów, koloru tła, czcionek itp.

Krokiem naprzód było wprowadzenie w 1996 r. standardu kaskadowych arkuszy stylów CSS [4]. Pozwoliło ono oddzielić warstwę danych dokumentu od warstwy prezentacji. Od tej chwili dokument składa się jedynie z logicznych części (paragraf, nagłówek, stopka...). Styl CSS jest odpowiedzialny jak wyglądają w tym momencie nagłówki i jakich używamy czcionek. Dokument HTML może już zatem zawierać jedynie dane.

Jeżeli dodatkowo ten dokument jest poprawnym („well-formed") plikiem XML [5] (czyli jest to plik XHTML [6]), łatwiejszym staje się czytanie i analizowanie takiego pliku przez maszynę. W tym momencie analizatory mogą dzielić taki tekst na logiczne części, mogą poznać jego strukturę. Wciąż jednak nie wiadomo o czym tak naprawdę jest ten dokument i jaką niesie treść. Po analizie liczności wyrazów programy mogą zgadywać czego dotyczy, nie mogą jednak rozumieć tekstu. Są to wciąż ciągi liter, nie mające dla komputera sensu. W

szczególności, na tej podstawie systemy komputerowe nie mogą tworzyć baz wiedzy, na tej podstawie nie można wnioskować (za wyjątkiem wnioskowań przybliżonych, rozmytych czy innych form wnioskowania heurystycznego).

Tymczasem pożądane jest, aby komputery potrafiły jak najprecyzyjniej wnioskować na temat zgromadzonych w Internecie informacji (podobnie do tego jak to potrafi człowiek). Gdyby udało nam się przechowywać dane w taki sposób, aby były one zrozumiałe nie tylko dla człowieka, tworzyłyby się nowe perspektywy zastosowań systemów informatycznych. Zauważmy tutaj, że jak do tej pory, aplikacje potrafią nam analizować dane, które są dobrze ustrukturalizowane. Najlepiej, jeżeli takie dane są umieszczone w relacyjnej bazie danych, są pełne i niesprzeczne. Niestety to są często zbyt silne założenia, czy też wymagania do danych. Chcielibyśmy móc korzystać z danych dostępnych w Internecie. W Sieci jednak każdy podmiot (twórca) w odmienny sposób gromadzi i przechowuje dane. Nie wiadomo więc na przykład jakie są relacje między danymi od różnych dostawców. Nie wiadomo jak się te dane mają do siebie. Czy znaczenie np. konkretnych pól w tabelach jednej relacyjnej bazy danych jest takie samo jak podobnych pól w innej bazie? Istnieje wiec potrzeba stworzenia standardu formatu danych odpowiedniego do przechowywania treści dokumentów. Tak, aby takie dokumenty były zrozumiałe dla każdego, kto tego potrzebuje (i oczywiście jest uprawniony do ich odczytu, ale to jest już zupełnie inny problem). Jest potrzeba, aby dokumenty (dane, wiedza) były łatwo przyswajalne i rozumiane przez komputery. Żeby maszyny potrafiły znać treść danych pochodzących z wielu źródeł, żeby na tej podstawie potrafiły wyciągać wnioski.

Oddzielenie treści dokumentu od jego formy jest już możliwe od czasów wprowadzenia i popularyzacji formatu XML. Format XML zdobył świat przez swoją prostotę i uniwersalność. XML nie narzuca żadnych obostrzeń na dane przechowywane w dokumencie. Format ten jedynie nadaje pewne ramy, jak dokument ma wyglądać, aby był rozumiany przez obie strony komunikacji (piszącego i czytającego). Narzędzia takie jak DTD [7] i XML Schema [8] pozwalają uzgodnić strukturę wymienianych danych. Dzięki nim można łatwo stworzyć szablon poprawnego (według projektanta) dokumentu. Dzięki swoim zaletom, XML stał się platformą wymiany danych między systemami informatycznymi. Powstało wiele narzędzi do czytania, analizy i tworzenia plików XML, istnieją silniki baz danych oparte na plikach XML [9], istnieją języki zapytań do takich baz.

Mając do dyspozycji XML oraz style CSS (które można nakładać na „suche" dokumenty XML ustalając do pewnego stopnia ich wygląd na ekranie) jak i inne narzędzia (np. XSLT [10]) spełniono wiele postulatów dotyczących tego jakie powinny być dane przechowywane w Internecie. Wreszcie pojawiła się wyraźna granica między treścią a formą.

Dzięki temu osiągnięto sukcesy: te same dane mogły być źródłem dla różnych zastosowań, np. dla plików HTML, dla plików PDF, dla bazy danych. Dzięki temu można było również pójść krok dalej w usuwaniu szkodliwej redundancji danych. Jedna za złotych zasad programowania, ale także przechowywania danych i informatyki jako całości brzmi „Nie powtarzaj się” („**D**on’t **R**epeat **Y**ourself” – DRY [11]). Mając jedno źródło danych (np. właśnie plik XML), i generując pozostałe formy dokumentu (HTML, PDF...) jesteśmy zgodni z tą zasadą.

Nie został jednak rozwiązany problem rozumienia tekstu. Format XML wymaga jedynie, aby dane były „zamknięte” w odpowiednich tagach i aby te tagi miały spójną strukturę. Jednak fakt, że dane tworzą pewną spójną strukturę nie mówi nam jeszcze nic o ich znaczeniu. Na razie można było jedynie stwierdzić istnienie pewnego schematu w danych – np. stwierdzenie istnienia relacji między danymi. Wciąż jednak jaka to jest relacja – tzn. co ona ze sobą niesie, jakie ma konsekwencje, nie było wiadomo.

Na przeciw tym wyzwaniom wyszło znowu konsorcjum w3.org i w 1999 roku powstał standard RDF

Standard RDF

RDF [1], [12] to skrót od angielskiej nazwy Resource Descritpion Framework, co można tłumaczyć jako „Szkielet Opisu Zasobów”, „Ramowy Opis Zasobów” lub też „Podstawa do Opisu Zasobów”. Celem opracowania takiego standardu było stworzenie platformy wymiany danych jeszcze silniej zorientowanej na treść, niż zwykłe pliki XML. Jak można przeczytać na stronach http://w3.org [13], RDF jest językiem do reprezentacji informacji o zasobach w Internecie. Dla celów dokumentów dostępnych w Sieci (stron internetowych) głównym zadaniem jest opisanie metadanych. Wiele byśmy uzyskali mając informację o czym dokładnie jest dany dokument. Mając informacje o jego autorze, dacie stworzenia, jego streszczenie i czego dotyczy, wyszukiwarki mogłyby działać o wiele dokładniej. Poza opisem zasobów dostępnych w Internecie (dokumenty, pliki), możliwy (i również przydatny) jest także opis zasobów istniejących w świecie realnym, a mających jakieś odzwierciedlenie w Internecie. Przykładowo zasobem takim może być osoba (powiedzmy Wawrzyniec Hyska). Mając zdefiniowany taki zasób, możemy się nim posługiwać w wielu miejscach; pozwala nam to potem stwierdzać jednoznaczność takiego zasobu. Możemy zatem mieć pewność, że ten konkretny Wawrzyniec Hyska dokonał

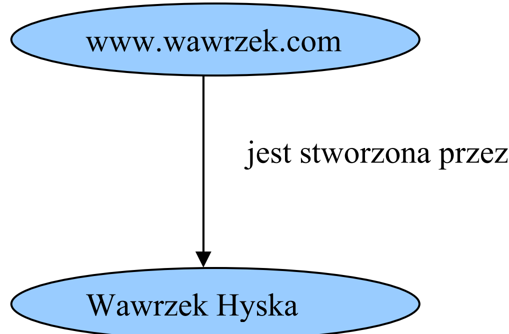
zakupów w księgarni internetowej, że jest autorem strony internetowej, że ma konto pocztowe i wykupioną polisę ubezpieczeniową.

Pliki RDF opisują zatem zasoby. Jak to się dzieje? Po pierwsze, pliki te są (zazwyczaj) dokumentami XML. Są to zatem pliki tekstowe, które spełniają wymagania XML – mają odpowiednią strukturę tagów. Jednak tak naprawdę to nie składnia decyduje o użyteczności plików RDF. Składnia jest warstwą niższą i jest tylko sposobem zapisu pliku; w zasadzie nie ma ona większego znaczenia dla istoty RDF.

Istotą standardu RDF jest struktura opisywanych zasobów. Zasoby są bowiem opisywane w formie grafu skierowanego. Zasoby są opisywane w relacjach z innymi zasobami. Dzięki temu wiemy jakie są rzeczywiste powiązania między nimi. W grafie tym wierzchołkami są zasoby, a krawędziami relacje między nimi. Weźmy najprostszy przykład. Chcemy opisać, że autorem strony www.wawrzek.com jest Wawrzek Hyska. Zatem w formie zdania wyglądać to będzie mniej więcej tak:

„strona www.wawrzek.com" została „stworzona" przez „Wawrzek Hyska"

A w formie grafu:



Wszystkie zdania (ang. *statements*) w plikach RDF wyglądają podobnie jak powyższy przykład i składają się z trzech elementów: podmiotu (ang. *subject*), orzeczenia (ang. *predicate*) i przedmiotu (ang. *object*). Plik RDF to tak naprawdę zbiór takich zdań – takich trójek. W powyższym zdaniu „strona www.wawrzek.com" jest podmiotem, „jest stworzona" jest orzeczeniem, a „Wawrzek Hyska" jest dopełnieniem. Wszystkie te trzy części są zasobami. Jeżeli później, w jakimś innym miejscu, będziemy mieli zdanie

„Wawrzek Hyska" „studiuje na" „Politechnika Warszawska"

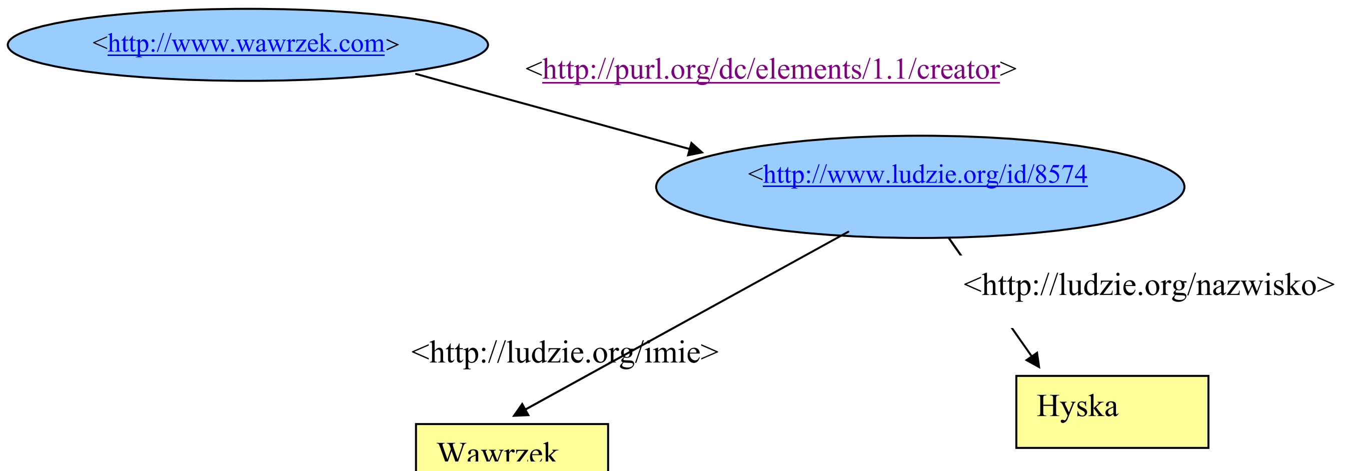
będziemy mogli połączyć te dwa fakty. Będzie można wysnuwać przesłanki, że strona www.wawrzek.com być może zawiera jakieś artykuły na tematy techniczne albo studenckie.

Aby jednak to wszystko działało, muszą być spełnione pewne dodatkowe warunki. Po pierwsze, musimy używać unikalnych identyfikatorów. Po drugie, zdania takie muszą być szeroko rozumiane. To znaczy orzeczenia nie mogą być dowolne, muszą one być jakoś zestandaryzowane. Jeżeli wszyscy będą używać tego samego słownika orzeczeń, dla wszystkich zwrot „studiuje na” będzie oznaczał to samo to i wyciągane wnioski powinny być słuszne.

Ponieważ RDF ma być używany przez komputery, a nie przez ludzi, trzeba stosować język inny niż polski czy angielski. Aby zasoby (podmioty, orzeczenia i przedmioty) były rozumiane przez maszyny, są one reprezentowane przez URI – Unified Resource Identifier (ang. Uniwersalny Identyfikator Zasobów, [14]). W Internecie używane są adresy URL (Uniform Resource Locator). Adresy (identyfikatory) URI wyglądają jak adresy stron internetowych (URL jest de facto URI). Jak było wspomniane, URI (a tak naprawdę referencje URI) opisują jednak nie tylko zasoby dostępne w Internecie. Dotyczą one także zasobów nie związanych bezpośrednio z Siecią (osoby, organizacje, przedmioty), a także abstrakcyjnych pojęć – np. właśnie orzeczeń. Dlatego w ścisłym RDFie pierwsze zdanie przytoczone jako przykład powyżej będzie wyglądało na przykład tak:

```
<http://www.wawrzek.com>          <http://purl.org/dc/elements/1.1/creator>
<http://www.ludzie.org/id/85740>
```

W ten sposób zostanie stwierdzone że zasób <http://www.wawrzek.com> jest w relacji <http://purl.org/dc/elements/1.1/creator> z zasobem <http://www.ludzie.org/id/85740>. W tym przypadku skądinąd będzie wiadomo, że zasób <http://www.ludzie.org/id/85740> oznacza osobę, której imię jest „Wawrzek”, a nazwisko „Hyska”. Graf reprezentujący powyższe zdanie będzie wyglądał następująco:



Jak widać na schemacie, zarówno przedmioty, obiekty, jak i orzeczenia reprezentowane są przez URI. Poniżej graf ten zapisany jest w postaci tekstowej:

```
<http://www.wawrzek.com><http://purl.org/dc/elements/1.1/creator><http://ww
w.ludzie.org/id/85740>
<http://www.ludzie.org/id/85740><http://ludzie.org/imie> „Wawrzek”
<http://www.ludzie.org/id/85740><http://ludzie.org/nazwisko> „Hyska”
```

Każdej krawędzi grafu odpowiada jedna trójka. Dlatego wierzchołki, z których wychodzi wiele krawędzi, pojawiają się w trójkach kilkakrotnie.

W grafie RDF wierzchołkami są zasoby reprezentowane przez URI. Gdybyśmy mieli jednak do czynienia wyłącznie z URI to niewiele byśmy wiedzieli. Znane byłyby relacje pomiędzy zasobami, ale nie znalibyśmy żadnych informacji o tych zasobach. Dlatego naturalne jest, że wierzchołkami w grafie RDF mogą być także literały, liczby, daty itp. Na powyższym rysunku literały są reprezentowane przez prostokąty.

Tak więc istotą schematu RDF jest reprezentowanie danych w postaci grafu. Chcąc przedstawić taki graf zapisuje się go w postaci zdań – trójek (podmiot, orzeczenie, przedmiot). Dla sensu tego standardu nie jest ważna składnia takiego zapisu. Jak wspomniano, najczęściej spotykaną składnią jest składnia plików XML. Istnieją jednak inne sposoby, no. formaty N3, czy też N-triples [15]. Przykłady różnych zapisów zdań RDF zostaną podane później.

Aby zapis trójek był wygodniejszy, wprowadzono skróty. Na początku dokumentu RDF można zdefiniować skróty dla poszczególnych prefiksów URI, a potem używać już tylko krótszych form. Dla przykładu definiujemy

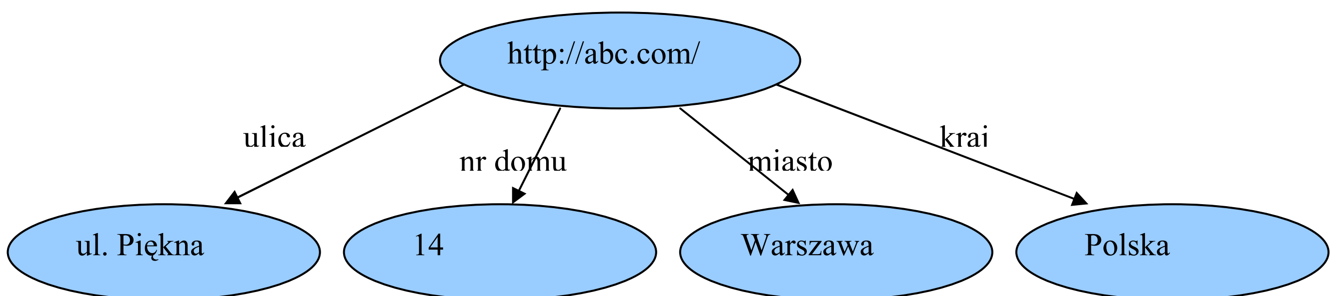
```
ludid = <http://ludzie.org/id/>
lud = <http://ludzie.org/>
```

i możemy potem używać zdań postaci

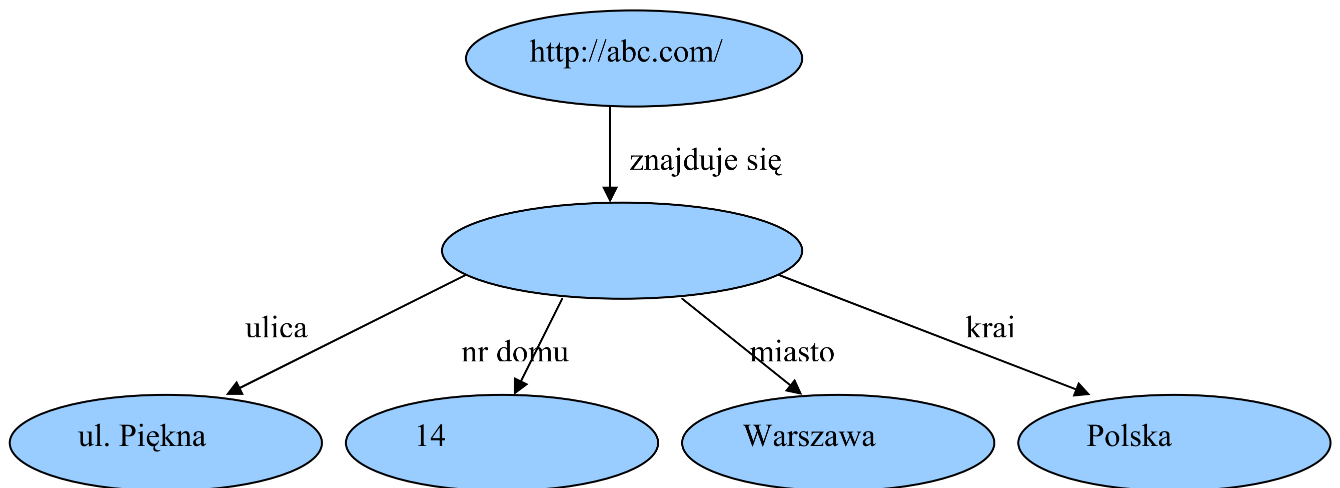
```
ludid:8574 lud:imie „Wawrzek"
ludid:8574 lud:nazwisko „Hyska"
```

Używanie identyfikatorów URI niesie bardzo wiele korzyści. Po pierwsze, takie identyfikatory są unikalne. Istnieje wiele osób nazywających się „Jan Nowak", dlatego identyfikowanie człowieka z tym ciągiem liter nie miałoby sensu. Poza tym graf RDF ma opisywać relacje między zasobami. Bo przecież strona www.wawrzek.com nie została stworzona przez ciąg liter „Wawrzek Hyska". Została ona stworzona przez pewną osobę (zasób). Cechami tej osoby są jej imię i nazwisko. Same literały nie są zasobami, które są w relacji z zasobem „www.wawrzek.com". Bardzo ważne jest rozróżnienie między zasobami a literałami i odpowiednie stosowanie identyfikatorów URI.

Inną ciekawą cechą standardu RDF jest dopuszczenie tzw. „blank nodes", czyli pustych węzłów. Za przykładem z w3.org, wyobraźmy sobie zasób – pewną firmę, znajdującą się pod pewnym adresem. Chcąc trzymać ten adres w sposób prawidłowy (łatwy w użyciu), nasz graf wyglądałby mniej więcej tak



Jeżeli mamy więcej danych o tej firmie, graf taki jest niewygodny. Z zasobem http://abc.com jest w relacji bardzo wiele zasobów. A przecież de facto dane o ulicy, numerze domu, mieście i kraju nie są związane bezpośrednio z tą firmą. Są to tylko dane adresowe siedziby tej firmy. Ponieważ jednak nie chcemy tworzyć sztucznego tworu, jakim byłby zasób „adres", tworzony jest właśnie pusty węzeł, który porządkuje nam strukturę grafu:



Dzięki zastosowaniu pustego węzła, nie musieliśmy tworzyć zasobu „adres firmy abc". Taki zasób byłby zapewne nie wykorzystywany nigdzie poza dokumentem i nie ma potrzeby np. troszczyć się o unikalne ID dla niego. Poniżej przedstawiony jest zapis takiego grafu w postaci trójek. Dla uproszczenia, zarówno w grafie jak i w trójkach zastosowałem uproszczone nazwy orzeczeń – nie pełne URI.

```
<http://abc.com>  <znajduje si•>    _:abcadres
_:abcadres  <ulica>     „ul. Pi•kna"
_:abcadres  <nr domu>   „14"
_:abcadres  <miasto>    „Warszawa"
_:abcadres  <kraj>      „Polska"
```

Jak widać, w celu oznaczenia wierzchołka jako pusty (blank) stosuje się prefiks **_:**.

Aby informacje zawarte w plikach RDF były zrozumiałe, musi istnieć jakaś platforma porozumienia. Jak było wspomniane, tylko w przypadku wspólnego, szeroko znanego zbioru orzeczeń jest szansa, że czytający zrozumie to co napisał twórca dokumentu. Istnieją w chwili obecnej słowniki orzeczeń dotyczące pewnych fragmentów rzeczywistości, jednym z przykładów jest tzw. Dublin Core [16]. Jest to zbiór orzeczeń do opisu danych dostępnych w Internecie (zasobów, stron, dokumentów). Dublin Core definiuje między innymi następujące właściwości metadanych:

- Tytuł (Title)
- Twórca (Creator)
- Temat (Subject)
- Opis (Description)
- Wydawca (Publisher)
- Ofiarodawca (Cotributor)
- Data (Date)
- Typ (Type)
- Format
- Idyntyfikator zasobu (Identifier)
- Źródło (Source)
- Język (Language)
- Związek z innymi zasobami (Relation)
- Zakres informacji (Covarage)
- Prawa autorskie (Rights)

Adres „http://purl.org/dc/elements/1.1/", który stosowano wcześniej, jest właśnie prefiksem do URI orzeczeń z Dublin Core. Dlatego często w dokumentach RDF możemy spotkać:

zdefiniowanie prefiksu „dc”:

```
xmlns:dc="http://purl.org/dc/elements/1.1/">
```

i używanie go w dokumencie:

```
<dc:title>Strona Osobista Wawrzka Hyski</dc:title>
```

Teraz wrócimy jeszcze do składni RDF. Najczęściej stosowaną składnią jest XML. Poniżej przykład krótkiego pliku RDF w formacie XML. Dla wygody po lewej stronie umieściłem numery linii.

```
1: <rdf:RDF
2: xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3: xmlns:dc="http://purl.org/dc/elements/1.1/">
4: <rdf:Description rdf:about="http://www.wawrzek.com">
5: <dc:title>Strona domowa Wawrzka Hyski</dc:title>
6: <dc:description>Opis moich zainteresowa•, krótkie artykuły o algorytmach
i strukturach danych, moje projekty uczelniane
7: </dc:description>
8: <dc:publisher>Wawrzek Hyska</dc:publisher>
9: <dc:date>2003-01-07</dc:date>
10: <dc:subject>
11: <rdf:Bag>
12: <rdf:li>Algorytmy i Struktury Danych</rdf:li>
13: <rdf:li>Projekty uczelniane<rdf:li>
14: <rdf:li>John Irving</rdf:li>
15: </rdf:Bag>
16: </dc:subject>
17: <dc:type>World Wide Web Home Page</dc:type>
18: <dc:format>text/html</dc:format>
19: <dc:language>pl</dc:language>
20: </rdf:Description>
21:</rdf:RDF>
```

W trzeciej linijce definiujemy prefiks do słownika Dublin Core jako “dc”. Następnie, w linijkach 5, 6, 8, 9 i 10 posługujemy się tym prefiksem do oznaczania ogólnie znanych metadanych. Warto zwrócić uwagę na linijkę 11, gdzie elementy zostają wyliczone za pomocą znacznika „Bag”. RDF wprowadza trzy typy wyliczeń: Bag (dosł. torba, zbiór), Seq (sekwencja czyli ciąg), Alt (ciąg alternatyw). Wyliczenie typu Bag jest wyliczeniem, które nie nadaje wyliczanym elementom żadnej dodatkowej struktury (np. ni narzuca żadnych związków między nimi).

Standard Dublin Core wprowadza nie tylko standardowe orzeczenia (predicates), ale również standardowe oznaczenia powszechnych zasobów. Dzięki takiemu słownikowi wiadomo powszechnie, że np. skrót „pl” odnosi się do języka polskiego, a „en” do angielskiego.

Zdania zapisane w postaci N3 [15] wyglądają podobnie do tego, jak trójki przestawione wcześniej. Poniższy przykład pokazuje te same dane co powyżej, ale w formie N3:

```
<#http://www.wawrzek.com>
      <#dc:title> "Strona domowa Wawrzka Hyski";
      <#dc:description>  "Opis  moich  zainteresowa•,  krótkie  artykuły  o
algorytmach i strukturach danych, moje projekty uczelniane ";
      <#dc:publisher> "Wawrzek Hyska";
      <#dc:date> "2003-01-07";
      <#dc:subject> "Algorytmy i Struktury Danych", "Projekty uczelniane",
"John Irving";
      <#dc:type> "World Wide Web Home Page";
      <#dc: format> "text/html";
      <#dc: language> "pl".
```

Jak widać, zapis ten jest dużo czytelniejszy dla człowieka. Zapis N3 pozwala też na znaczne skrócenie dokumentów. Wprowadzono w nim na przykład przecinek, który oddziela kolejne przedmioty których dotyczy zdanie. Można więc napisać, że strona zawiera informacje na trzy tematy, zamiast pisać podobne zdania trzy razy. Podobnie, średnik pozwala wprowadzać kolejne zdania (orzeczenia) dotyczące jednego podmiotu. Piszemy zatem na początku, że zdania będą dotyczyły strony www.wawrzek.com i już wiadomo, że wszystkie stwierdzenia (tytuł, data...,) jej będą dotyczyć.

Język RDQL

Kiedy dane są już przechowywane w formacie RDF, chcielibyśmy móc przeszukiwać takie dane według kryteriów budowanych z predykatów. W klasycznych, SQL-owych bazach danych możemy tworzyć kwerendy i zawężać odpowiedzi jedynie według najprostszych relacji. W klauzuli WHERE zapytań SQL-owych możemy podać relacje równości, nierówności, mniejszości/większości, LIKE, BETWEEN. Ograniczenia te wynikają z faktu, że w „klasycznej” bazie danych informacje są przechowywane jako nieznaczące teksty. System nie wie nic o ich treści i znaczeniach, co najwyżej potrafi porównać Stringi, liczby, wyrażenia regularne. Jeżeli jednak baza jest oparta o pliki RDF, mamy wiedzę także o bardziej skomplikowanych relacjach między danymi (zasobami). Korzystając z wcześniejszych przykładów, taką bazę można odpytać o zasoby utworzone (predykat dc:creator) przez osobę o nazwisku „Hyska”. Systemy oparte o RDF rozumieją więcej niż systemy SQL-owe. W kwerendach do plików RDF możemy używać dowolnych orzeczeń (czyli odnajdywać dowolne relacje), nie tylko tych najprostszych, znanych z SQL.

Językiem, w którym budujemy zapytania do baz danych opartych na plikach RDF jest RDQL [17], [18]. RDQL jest skrótem od RDF Data Query Language, czyli język zapytań do danych z plików RDF. Najprostsze zapytanie w tym języku wygląda tak:

```
SELECT ?x
WHERE (?x <http://www.w3.org/2001/vcard-rdf/3.0#FN> "Wawrzek Hyska")
```

Jest to pytanie o te zasoby, które mają własność <http://www.w3.org/2001/vcard-rdf/3.0#FN> wynoszącą „Wawrzek Hyska". Jak można się domyślić, własność ta odnosi się do słownika stworzonego przez organizację w3.org i oznacza pełne imię i nazwisko (full name). Podobnie jak w języku SQL, polecenie SELECT oznacza, że chcemy pobrać dane, natomiast klauzula WHERE pozwala nam je zawęzić. Jeżeli przed literałem jest znak zapytania (tak jak tutaj w „?x"), oznacza to, że deklarujemy zmienną. To zapytanie zwróci nam zasoby – czyli URI. Często chcemy jednak mieć dostęp do innych danych. Do tego może służyć zapytanie:

```
SELECT ?wiek
WHERE (?x <http://www.w3.org/2001/vcard-rdf/3.0#FN> "Wawrzek Hyska")
 (?wiek <http://www.w3.org/2001/vcard-rdf/3.0#AGE> ?x)
```

To pytanie zwróci nam wiek osoby o imieniu i nazwisku „Wawrzek Hyska". Jest bowiem polecenie wybrania takich danych **?x**, że imieniem i nazwiskiem **?x**-a jest „Wawrzek Hyska", a **?wiek** jest wiekiem **?x**-a. Czyli silnik bazodanowy odnajduje osoby o nazwisku „Wawrzek Hyska", a następnie dla takich osób odnajduje ile one mają lat. Ponieważ przy słowie SELECT jest tylko **?wiek**, zwrócone zostaną tylko liczby odpowiadające wiekowi.

Poza klauzulą WHERE, można narzucać inne warunki na zwracane dane. Na przykład

```
SELECT ?wiek
WHERE (?x <http://www.w3.org/2001/vcard-rdf/3.0#FN> "Wawrzek Hyska")
 (?wiek <http://www.w3.org/2001/vcard-rdf/3.0#AGE > ?x)
 AND ?wiek < 24
```

wydobyte dane można jeszcze przefiltrować przy pomocy polecenia AND:

Teraz system poza odnalezieniem tych samych osób, co w poprzednim zapytaniu, zawęzi jeszcze odpowiedź do osób młodszych niż 24 lata. Klauzula AND jest zatem podobna do znanej z SQL klauzuli HAVING.

W RDQL, podobnie jak w plikach RDF, aby nie używać długich nazw predykatów, stosuje się aliasy. Do stworzenia aliasu służy słowo kluczowe USING. Możemy zatem poprzednie zapytania zapisać w ten sposób:

```
SELECT ?wiek
WHERE (?x <w3#FN> "Wawrzek Hyska")
 (?wiek <w3#AGE > ?x)
 AND ?wiek < 24
 USING w3 FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

Wprowadzamy zatem oznaczenie, że w tym zapytaniu ciąg “w3” będzie oznaczał tyle co „http://www.w3.org/2001/vcard-rdf/3.0#”. O tym jak korzysta się z języka RDQL, o narzędziach i silnikach do zarządzania danymi w postaci RDF będzie napisane w drugim rozdziale.

Ontologie

W myśl tradycyjnej definicji pochodzącej z filozofii, Ontologia jest nauką o istnieniu, niejako pokrewną metafizyce. Z samej etymologii terminu można wnioskować, czym ta nauka się zajmuje. Z greckiego „*to on*” oznacza „coś co jest”, „cokolwiek” [19]

W ujęciu informatycznym pojęcie ontologii ma podobne znaczenie, być może tylko bardziej konkretne. W sieciach semantycznych ontologie używane są do wprowadzenia znaczenia i relacji. Ontologia opisuje zatem znaczenie bytów (a w konkretnych zastosowaniach po prostu znaczenie wyrazów). Stosowanie jej wprowadza ład w rozumowanie maszyn poprzez przejście od operowania na „suchych” tekstach do operowania na pojęciach, znaczeniach. Jak wcześniej wspomniano, dąży się do tego, aby komputery potrafiły przetwarzać teksty semantycznie, rozumieć ich treść i wnioskować. Ontologia wprowadza pojęcia, które zapewniają, że dany tekst jest jednakowo rozumiany przez obie strony komunikacji (piszącego i czytającego). W praktyce programistycznej, wprowadza ona schemat, który pozwala przekształcać struktury danych (czyli dane rozumiane przez maszynę) na tekst i z powrotem. Dzięki temu dane wraz z ich pełnym znaczeniem mogą być wymieniane między komunikującymi się bez straty „sensowności”.

Wprowadzenie takiego schematu odbywa się poprzez dwa działania: *kategoryzację* i *hierarchizację*.

Kategoryzacja jest to nazywanie bytów i przypisywanie bytom pewnych własności. Przykładem może być tu stworzenie pojęcia „człowiek”. Tworzy się pewne pojęcie, pewną klasę. Po stworzeniu klasy „człowiek” można opisywać, że pewne byty są, a pewne nie są „ludźmi”.

Hierarchizacja to zbudowanie struktury klas (kategorii). Każdą klasę można umieścić w hierarchicznej strukturze pojęć. Byt należący do danej klasy ma nie tylko własności

wypływające z niej bezpośrednio, ale także własności klas nadrzędnych, po których dana klasa dziedziczy.

Przykładowo, klasa „człowiek” jest podklasą klasy „zwierzę” i wszyscy ludzie posiadają cechy zwierząt (np. są organizmami żywymi). Jednocześnie, klasa „człowiek” jest podklasą klasy „obiekty cięższe od powietrza”, co oznacza, że żaden człowiek nie potrafi latać bez pomocy dodatkowej aparatury.

Do informatyki wprowadza się ontologię, aby uporządkować komunikację. Ontologie mają stanowić punkt odniesienia dla stron, mają być pewnego rodzaju metadanymi, dzięki którym komunikowanie się będzie jednoznaczne. Aby do tego doszło, strony komunikacji muszą oczywiście zgodzić się na tą samą ontologię. Aby przekaz był rozumiany przed odbierającego dokładnie w myśl zawartości założonej przez nadającego, struktura kategorii i ich hierarchia musi być identyczna. Jeśli przykładowo jedna strona za „człowieka” uznaje istotę dwunożną z wyłamanym kciukiem, a drugą strona za człowieka uznaje istotę posiadającą duszę, komunikacja nie będzie jednoznaczna.

Język RDF może być używany do reprezentowania ontologii. Zauważmy jednak, że w celu opisu ontologii stworzone zostały również inne języki np. OWL [20] (Ontology Web Language). OWL jest językiem mocniejszym od RDF, dzięki czemu maszyny czytające dane w OWL są w stanie wyciągać dalej idące wnioski niżby opierały się na danych w RDF. Problem z językiem OWL polega jednak również dokładnie na tym samym na czym polega jego siła. Tak jak język RDF jest językiem bardzo prostym, tak OWL został rozbudowany i sposób jego rozszerzenia (a zatem i „wzmocnienia”) nie pozostaje bez krytyków. W wyniku tejże krytyki, OWL nie osiągnął do tej pory popularności takiej jak znacznie prostszy RDF. Można mieć również wątpliwości czy nie stanie się tak, że OWL zostanie zastąpiony w przyszłości jakimś innym językiem. Biorąc to pod uwagę, w przypadku systemu wspomagania podróży podjęto decyzję zastosowania prostszego, ale i popularniejszego języka RDF.

Systemy Wieloagentowe

Próba definicji, historia systemów agentowych

Systemy wieloagentowe to systemy komputerowe oparte i zbudowane na bazie programów – agentów. Tworzenie takich systemów – programowanie agentowe - polega na stworzeniu zbioru programów agentowych i środowiska do ich działania. Dotychczas nie powstała jedna definicja agenta, na którą zgodziliby się wszyscy badacze tej dziedziny informatyki. Są oni jednak zgodni co do pewnych cech, jakie powinien mieć taki program:

1. Agent powinien być autonomiczny. Oznacza to, że powinien wykonywać zlecone mu zadania samodzielnie.
2. Agent działa w czyimś imieniu. Agent jest czyimś delegatem (użytkownika, programu, innego agenta) i wykonuje swoje zadania w imieniu zlecającego.
3. Agent powinien umieć się komunikować. Chodzi tu głównie o komunikację z innymi agentami (ale możliwa jest również komunikacja z użytkownikiem czy innymi programami takimi jak bazy danych). Dzięki takiej komunikacji działanie agenta może być skuteczniejsze.
4. Agent powinien mieć zdolności percepcji. Powinien zatem dostrzegać zmiany w środowisku, w jakim jest uruchomiony i powinien umieć na te zmiany reagować.
5. Przed agentem często stawia się także wymagania mobilności. Często potrzebne jest, aby program agentowy potrafił przemieścić się w inne miejsce (na inną maszynę, do innego środowiska) i tam kontynuować swoje zadania.

Systemy agentowe często kojarzone są z systemami sztucznej inteligencji. Dlatego nierzadko z agentami kojarzy się też takie cechy jak

- ogólnie pojęta „inteligencja”
- odporność na awarie
- możliwość samonaprawy
- zdolność uczenia się

- zdolność adaptacji do nowych warunków

Poniżej przedstawione są cztery napotkane w [21] definicje agenta:

Definicja 1. Cokolwiek, co może być uznane jako obserwujące otoczenie poprzez sensory i działające w ramach tegoż otoczenia poprzez efektory [22] (Russell i Norvig, 1995)

Definicja 2. Jednostki programowe podejmujące działania w imieniu użytkownika lub innych programów, w pewnym stopniu niezależnie lub autonomicznie, które działając stosują pewną wiedzę lub reprezentację celów lub potrzeb użytkownika [23] (IBM, 1997)

Definicja 3. Zamknięty system komputerowy znajdujący się w pewnym otoczeniu, posiadający umiejętność elastycznego działania w tymże otoczeniu, działania polegającego wypełnieniu celów dla jakich został stworzony [24] (Wooldridge, 1997)

Definicja 4. Autonomiczny system znajdujący się w dynamicznym otoczeniu działający niezależnie od narzucanych przezeń ograniczeń i wypełniający w jego ramach zbiór celów lub poleceń, dla których został stworzony [25] (Maes, 1998)

W 1996 roku założona została organizacja standaryzująca techniki programowania agentowego – FIPA (Foundation for Intelligent Physical Agents) [26]. Zajmuje się ona tworzeniem obowiązujących w programowanie agentowym standardów i formatów.

Podział i Funkcjonalność systemów agentowych

Aby lepiej sobie wyobrazić działania agentów i systemów agentowych, najlepiej prześledzić przykłady zastosowań tej metodologii. Systemy agentowe są z powodzeniem tworzone w środowiskach akademickich, powoli zaczynają się też pojawiać zastosowania komercyjne. Programy agentowe możemy podzielić ze względu na ich funkcjonalność:

1. Agenci personifikujący. Przykładem mogą być tu programy filtrujące wiadomości trafiające do użytkowników. Tacy agenci znają upodobania

swojego właściciela i w taki sposób dobierają wiadomości, aby były one dla niego jak najbardziej ciekawe. W oczywisty sposób ujawnia się tu potrzeba nauki agenta.

2. Agenci poczty elektronicznej. Program taki byłby odpowiedzialny za wstępną obróbkę trafiających do właściciela e-maili. Do obowiązków programu należałoby m.in. odsiewanie spamu, rozrzucanie poczty do odpowiednich katalogów, sprawdzanie na obecność wirusów, nadawanie priorytetów, łączenie wiadomości w ciągi (konwersacje), automatyczne odpowiadanie na proste maile. Tutaj także nieoceniona jest zdolność uczenia się, możliwości wymiany informacji z innymi agentami.
3. Agenci wspomagający użytkownika. Takie programy często wręcz sprawiają wrażenie kontaktu z osobą fizyczną. Ich zadaniem jest pomoc użytkownikowi w obsłudze aplikacji, urządzeń, serwisów internetowych
4. Agenci – asystenci. Szeroka grupa agentów wspomagających zarządzanie czasem i innymi aspektami życia codziennego. Przykładem może być tu zarządzanie kalendarzem spotkań (tutaj wybitnie potrzebna będzie komunikacja z bliźniaczymi agentami innych uczestników spotkania). Inne zastosowania to choćby sprawdzanie prognozy pogody i podpowiadanie ubrania czy sposobów podróżowania.
5. Agenci poszukujący informacji – programy, które w imieniu swojego właściciela szukają w zasobach internetowych pożądanej informacji. Agenci tacy korzystają z wyszukiwarek internetowych, z wiedzy innych agentów itp.
6. Agenci przeszukujący internet – w odróżnieniu od ostatnich, takie programy przeszukują Sieć w poszukiwaniu potencjalnie potrzebnej informacji. Nie jest to szukanie konkretnych danych, lecz wyszukiwanie wiadomości, które w mniemaniu agenta mogą być potrzebne jego właścicielowi.
7. Agenci zarządzający siecią – programy nadzorujące sieci komputerowe, wykrywające intruzów i awarie, reagujące na zagrożenia. Prowadzą również monitoring sieci i tworzą statystyki.
8. Agenci E-commerce. Z tą grupą programów wiązanych jest wiele nadziei biznesu. Można sobie wyobrazić programy wyręczające swoich właścicieli w podejmowaniu rutynowych decyzji biznesowych. Mogą być to agenci negocjujący ceny (oczywiście razem z agentami kontrahentów), wybierający najlepsze oferty, zarządzający zamówieniami i zakupami.

9. Agenci symulujący – za pomocą systemów agentowych można przeprowadzać symulacje złożonych systemów rozproszonych – choćby zachowań na rynkach papierów wartościowych.

Struktura Systemów Agentowych

Aby można było budować system wieloagentowy, potrzebne jest środowisko uruchomieniowe dla agentów (oczywiście możliwym jest wykorzystanie tylko „metafory agentowej” i stworzenie oprogramowania np. w języku C++, ale taka sytuacja nie interesuje nas w niniejszej pracy). Agenci są to programy komputerowe, które są uruchamiane i mogą się porozumiewać między sobą i oddziaływać z otoczeniem. Należy zatem stworzyć platformę, gdzie takie programy będą uruchamiane i która pozwoli im wykonywać swoje działania. Takie środowisko musi zapewnić:

1. System zarządzania agentami (Agent Management System - AMS) jest odpowiedzialny za tworzenie i zamykanie agentów, a także ich zawieszanie i wznawianie. Tutaj także prowadzona jest autentykacja agentów, migracja na i z innych maszyn, usługa nazywania (naming).
2. Kanał komunikacji między agentami (Agent Communication Channel – ACC) to usługa przesyłania wiadomość między agentami. Dzięki niej agenci mogą wymieniać informacje.
3. Usługę Katalogową (Directory Facilitator – DF) prowadzi katalog usług udostępnianych przez agentów.

Usługa nazywania (naming) jest konieczna z uwagi na potrzebę jednoznacznej identyfikacji agentów. Oczywiste jest, że jeśli agenci mają się porozumiewać, czyli wysyłać sobie komunikaty, każdy z nich musi być rozpoznawalny pod unikalnym identyfikatorem.

Jak wspomniano, komunikacja między agentami jest prowadzona poprzez wysyłanie komunikatów. FIPA stworzyła standard takich komunikatów zwany jako ACL (Agent Communication Language) [27]. Komunikaty takie są tekstem z jasno określoną strukturą. Do struktury wiadomości ACL należą takie dane jak:

- nadawca (AID – Agent’s ID, czyli identyfikator agenta)

- odbiorca (AID)
- adres zwrotny nadawcy (AID)
- typ wiadomości
- kontekst wiadomości (takie wiadomości jak „odpowiedź na...”)
- data i czas nadania
- ontologia (słownik)
- język
- treść (content)

Treść komunikatu jest tekstem. Aby jednak była on jednoznaczna, to właśnie tutaj korzysta się z ontologii. Jak zasugerowano powyżej, jeżeli nadawca i odbiorca mają tą samą ontologię, jest pewność, że komunikat zostanie zrozumiany prawidłowo. Jak widać, wiadomość niesie ze sobą nazwę ontologii, w jakiej została stworzona. Jak już przedstawiono, ontologie automatyzują tłumaczenie struktur danych na komunikaty – łańcuchy znakowe.

Ważnym polem w wiadomości ACL jest pole „typ” (Message Type). Określa ono cel wysłania wiadomości i ogólny jej wydźwięk. FIPA dopuszcza kilkanaście typów wiadomości, z których najbardziej popularne w tworzeniu systemów agentowych są między innymi: żądanie (REQUEST), informacje (INFORM), odrzucenie (REFUSE), czy też błąd (FAILURE). Jak zostanie pokazane w drugim rozdziale, przypisanie każdej wiadomości pewnego typu ma duży wpływ na projektowanie agentów. Tam też zostanie przedstawione dokładniej jedno środowisko uruchomieniowe agentów – JADE.

Zalety i wady podejścia agentowego

Programowanie agentowe, czy też tworzenie systemów agentowych, ma swoich zwolenników i przeciwników. Pomysły na rozwiązywanie skomplikowanych problemów poprzez wykorzystanie małych, autonomicznych programów nie są nowe. W dobie narastającej ilości zalewających nas informacji, takie podejście wydaje się być słuszne. Wydaje się, że samodzielne, uczące się programy będą w stanie zapanować nad lawinowo rosnącą ilością danych, że będą potrafiły te dane zanalizować. W wielu dziedzinach informatyki ewolucyjne podejście do rozwiązywania problemów przynosi dobre efekty. Podobnie może być z systemami agentowymi – agenci też potrafią się uczyć i ewoluować; co więcej, poprzez komunikację mogą wymieniać się doświadczeniami, co powinno zwiększyć

ich skuteczność. Dodatkową zaletą jest ich naturalna mobilność – agenci potrafią się przenosić na inne maszyny, co pozwoli zoptymalizować wykorzystanie mocy obliczeniowej i innych zasobów. Co więcej, istnieją możliwości implementowania agentów na maszyny mobilne (palmtopy, telefony komórkowe). Tacy agenci, mobilni, a więc mogący migrować między komputerami, serwerami w Internecie i urządzeniami mobilnymi otwierają nowe możliwości.

Do tej pory na programy agentowe patrzyliśmy z pozycji beneficjenta działalności agentów. Jednak na to zagadnienie można także spojrzeć od strony twórcy, czyli programisty. Można powiedzieć, że programowanie agentowe jest kolejnym krokiem na drodze porządkowania struktur dużych systemów informatycznych. W rozwoju inżynierii oprogramowania wielkim krokiem było pojawienie się programowania zorientowanego obiektowo (OOP), a potem także programowanie z komponentów (np. COM [28], ActiveX [29], Corba [30], Enterprise Java Beans [31]). Być może programowanie agentów jest dalszym krokiem w tym kierunku. Można przecież traktować agenta jako specjalistyczny moduł większej całości, jako część większej maszyny. Być może nie zawsze ma on wtedy wszystkie cechy agenta (być może nie jest potrzebna mobilność, umiejętność uczenia się, „inteligencja"), ale jako dobrze odseparowana całość, łatwo implementowalna jest dobrym rozwiązaniem. Tak jak wspomniane komponenty udostępniały swoje działania poprzez publiczne interfejsy, z agentem można się porozumiewać poprzez komunikaty ACL używając uzgodnionej ontologii. Wydaje się to bardziej uniwersalnym, odporniejszym na błędy i tańszym w utrzymaniu rozwiązaniem.

Tymczasem, mimo, że pomysły systemów agentowych pojawiły się już na początku lat 90., wciąż nie są one popularne. W zasadzie nie ma dużych, sprawdzonych, komercyjnych rozwiązań agentowych. Jest to jeden z argumentów krytyków podejścia agentowego. Zgłaszane wady technologii agentowych można podzielić na dwie grupy: wady z punktu widzenia teoretycznego i wady z punktu widzenia praktycznego. Krytycy teorii agentów zarzucają, że pomimo, że ta gałąź informatyki rozwija się już od ponad 20 lat, badacze nie mogą ustalić nawet podstawowej definicji agenta. Według nich, wskazuje to na słabość całej dziedziny technologii agentowych. Od strony praktycznej, krytykuje się przede wszystkim problemy z bezpieczeństwem. Poprzez swoją mobilność i autonomię, agent może bez trudu pełnić rolę intruza w systemie, może być koniem trojańskim. Z drugiej strony, agenci personifikujący, zawierający dane o swoich właścicielach mogą stać się łupem innych programów. Dane o właścicielu łatwo mogą się dostać w cudze ręce.

Wiele osób wskazuje na fakt, że podejście agentowe nie jest żadną rewolucją. Większość zadań, jakie stawia się przed agentami może z powodzeniem (i tak to się często dzieje) być wykonywane przez inne, „klasyczne" programy.

Rozdział 2.

Opis części praktycznej

Projekt AgentLab

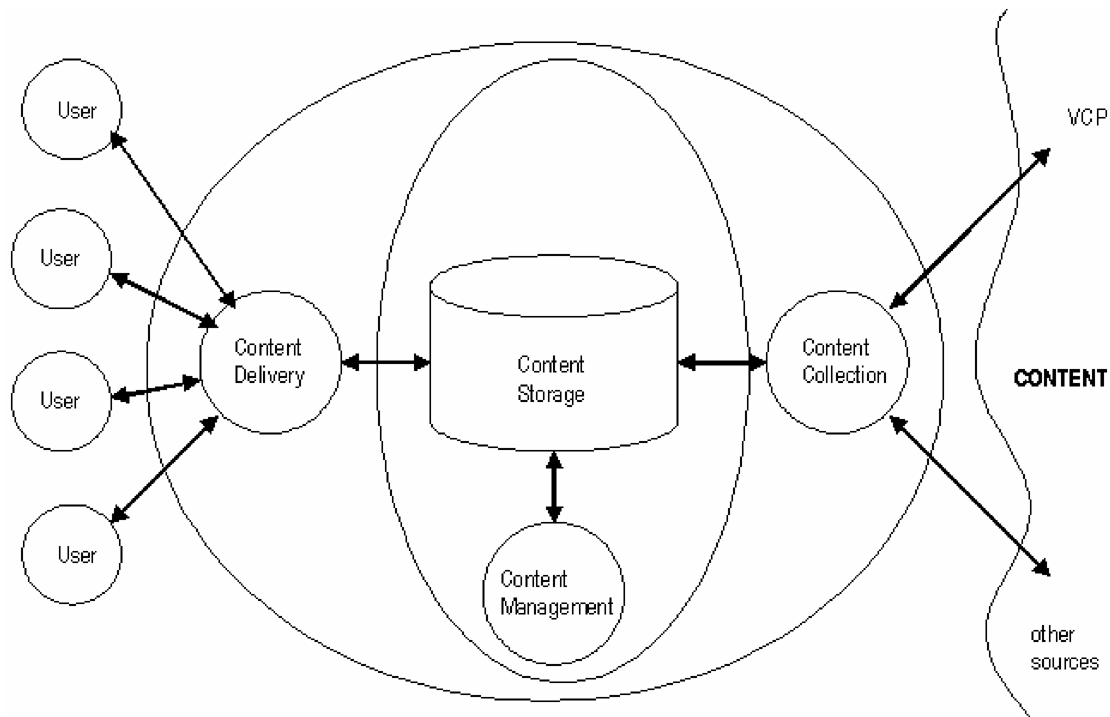
Oprogramowanie tworzące niniejszą pracę jest częścią większego projektu o nazwie AgentLab [32]. Projekt ten realizuje pewną platformę bazodanową opartą o technologie RDF i zbudowaną jako system wieloagentowy. Nad tym projektem w różnych jego fazach pracowało blisko 20 osób m.in. z Politechniki Warszawskiej, z Uniwersytetu Adama Mickiewicza w Poznaniu oraz Oklahoma State University. W niniejszym konkretnym zastosowaniu, platforma ta pełni rolę systemu wspomagającego podróże. Tym niemniej jest ona tworzona w sposób uniwersalny, tak aby mogła służyć jako podstawa do tworzenia systemów, wortali o różnej tematyce.

Funkcjonalność Systemu

Platforma AgentLab ma służyć jako system wspomagający podróżowanie. Poprzez zwrot „wspomaganie podróży" rozumie się gromadzenie, analiza i dostarczanie informacji związanych z turystyką, filtrowanie ich oraz odpowiednia prezentacja użytkownikowi. Aby jednak system nie był kolejną klasyczną bazą danych, jest on zaimplementowany jako zbiór agentów działających w sieci semantycznej. Dzięki formie w jakiej przechowywane są dane (RDF) aplikacja zna ich znaczenie i może zwracać wyniki bliższe temu, czego naprawdę szuka użytkownik. Ponieważ system jest zaimplementowany jako system wieloagentowy, posiada on dobrze wydzielone, niezależne komponenty (warstwy), jest skalowalny, niezawodny i łatwy do rozbudowy. Poniżej przedstawiony zostanie zarys architektury systemu.

Architektura Systemu

Ogólny schemat architektury platformy AgentLab wygląda następująco (schemat ten powtarza się także m.in. w pracach [33] i [34]:



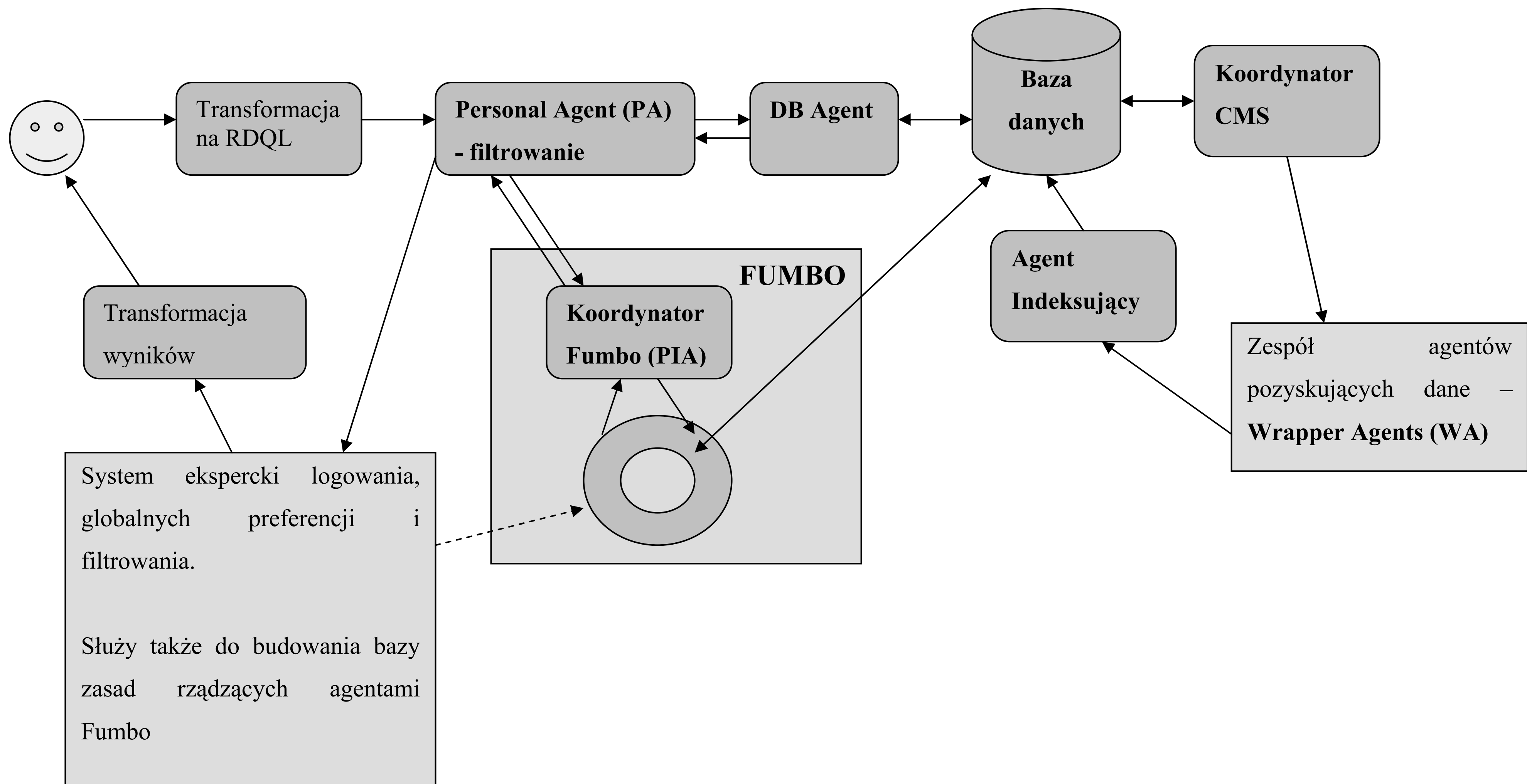
Platforma dzieli się na 3 podstawowe części: *Content Collection Subsystem, Content Management Subsystem* i *Content Delivery Subsystem*. Poniżej zostaną one pokrótce omówione.

1. Część *Content Collection Subsystem* (*CSS*) jest odpowiedzialna za gromadzenie informacji. Dane związane z turystyką (np. na temat połączeń, restauracji itp.) są zbierane z zewnętrznych stron internetowych. Aby dane te nadawały się do analizy, są one przekształcane do formatu RDF i ontologii obowiązujących w Platformie. CSS dysponuje agentami parsującymi strony HTML linii lotniczych, lotnisk, sieci kolejowych itd. (*Wrapper Agents – WA*). Prawidłowo odczytane i zinterpretowane dane trafiają do wewnętrznej bazy danych systemu, zaznaczonej na diagramie jako *Content Storage*.
2. *Content Management Subsystem* (*CMS*) to podsystem zajmujący się zarządzaniem dostępnymi danymi. Do przykładowych zadań tego podsystemu należy dbanie o regularne aktualizacje danych, rozwiązywanie konfliktów w przypadku sprzeczności czy też uzupełnianie istniejących danych ma podstawie nowych źródeł. Agenci tej części (*Data Management Agents – DMA*s) mają nadzór nad przechowywanymi danymi i delegują CSS do zbierania dodatkowych informacji.

3. *Content Delivery Subsystem* (CDS) jest to część najbliższa użytkownikowi systemu. Poprzez agentów CDS użytkownicy (na diagramie: Users) komunikują się z całą platformą. Kluczowymi agentami CDS są *Personal Agent* (PA), który odpowiada za filtrowanie i ocenianie odpowiedzi systemu pod kątem konkretnego użytkownika, a także podsystem *Fumbo*, który jest przedmiotem niniejszej pracy. PA jest bramą systemu do interakcji z użytkownikiem. Zapytania do bazy danych trafiają od PA i poprzez niego odsyłane są z powrotem do użytkowników. Fumbo (które dokładnie będzie opisane w dalsze części rozdziału) jest odpowiedzialne za rozszerzanie bazowego zbioru odpowiedzi, tak aby zwracane wyniki były szersze niż odpowiedź klasycznej bazy danych.

W systemie wprowadzony został podział źródeł danych na wiarygodne (*Verified Content Providers* - *VCP*) oraz inne (*Other Sources* – OS). Źródła VCP to źródła którym ufamy i na których opieramy działanie systemu. OS to pozostałe źródła internetowe. Być może nie są one zbyt wiarygodne, ale informacje tam się znajdujące mogą być również przydatne.

Na następnej stronie przedstawiony jest schemat przepływu zapytań i informacji w Platformie AgentLab.



Transformacja na RDQL
Personal Agent (PA) - filtrowanie
DB Agent
Baza danych
Koordynator CMS
FUMBO
Koordynator Fumbo (PIA)
Agent Indeksujący
Transformacja wyników
Zespół agentów pozyskujących dane – Wrapper Agents (WA)
System ekspercki logowania, globalnych preferencji i filtrowania.
Służy także do budowania bazy zasad rządzących agentami Fumbo

Działanie systemu

Na wejściu do systemu jest pytanie użytkownika zadane np. poprzez formularz HTML. Dostęp do internetu jest w obecnych czasach możliwy nie tylko z komputerów stacjonarnych; dostęp do Systemu jest zatem możliwy także poprzez protokół WAP czy z PalmTopów. Pytanie użytkownika trafia do Agenta Osobistego (PA) i jest tłumaczone na język RDQL. Jest to wykonywane na potrzeby bazy danych, która przechowuje dane w postaci RDF. Agent osobisty pełni dwie kluczowe funkcje w systemie:

1. Jest bramą i koordynatorem dla wszystkich zapytań trafiających do systemu od użytkownika
2. Filtruje i pozycjonuje przygotowane odpowiedzi na podstawie zapamiętanego profilu użytkownika

Po zarejestrowaniu zapytania w systemie i przetłumaczeniu go na RDQL, PA wysyła zapytanie do agenta bazodanowego (*DBAgent*).

Rolą agenta bazodanowego jest wydobycie informacji z bazy danych. W obecnej implementacji Systemu, bazą danych jest silnik MySQL. Należy podkreślić, że dane te są w formacie RDF, SQL służy tylko jako najniższa warstwa przechowywania informacji (zamiast, powiedzmy, plików). Agent bazodanowy otrzymuje zatem odpowiedź z bazy danych w postaci RDF. Odpowiedź ta wraz z pierwotnym zapytaniem trafia następnie do podsystemu odpowiedzialnego za potencjalne rozszerzenie tej wiedzy (będącego tematem tej pracy podsystemu *Fumbo*).

Podsystem Fumbo jest odpowiedzialny za stworzenie MRS (Maximal Result Set – Maksymalnego Zbioru Odpowiedzi). Zbiór taki zawiera wyniki z początkowego zbioru odpowiedzi, a także dane dodatkowe, otrzymane w wyniku zadawania kolejnych zapytań do bazy, które (przynajmniej według agentów Fumbo) mogą być użyteczne i które pasują w jakiś nietrywialny sposób do zadanego zapytania. Zamysłem jest przedstawienie użytkownikowi informacji, których on być może potrzebuje, a o które nie zapytał wprost. Często użytkownicy wyszukiwarek nie wiedzą dokładnie czego szukają. Zadaniem agentów rozszerzających zbiór odpowiedzi jest „domyślenie" się czego mógł poszukiwać użytkownik (i/lub czym jeszcze mógłby być zainteresowany) i podsuniecie mu tych informacji. Jako że ten kawałek Platformy jest tematem niniejszej pracy, będzie on w dalszych podrozdziałach przedstawiony dokładniej.

Utworzony Maksymalny Zbiór Odpowiedzi (MRS) trafia następnie ponownie do Agenta Osobistego (PA), wybierającego w tym momencie informacje potencjalnie najbardziej ciekawe dla użytkownika. Personal Agent jest w tym momencie odpowiedzialny za okrojenie Maksymalnego Zbioru Odpowiedzi tylko do tych informacji, które są potrzebne jego właścicielowi. Na podstawie danych zebranych wcześniej, agent osobisty wie, co lubi, a czego nie lubi jego właściciel. Agent osobisty usuwa więc z MRS odpowiedzi, które według niego nie zadowolą właściciela. Nadaje on też odpowiedziom priorytety – sortuje je według wystawionych przez siebie ocen. Na podstawie decyzji użytkownika, agent osobisty buduje sobie jego profil i zbiera informacje o jego preferencjach. Na tej podstawie, przy późniejszych zapytaniach, agent osobisty może w coraz większym stopniu trafnie podsuwać interesujące odpowiedzi.

Tak opracowana odpowiedź trafia do początkowego Usera, który zadał pytanie.

Jednak nie jest to koniec pracy. Jednym z kluczowych elementów frameworku jest analiza dalszych posunięć użytkownika. Agent ekspercki obserwuje zapytania, zwracane odpowiedzi i dalsze wybory użytkownika. Na tej podstawie budowana jest wiedza o globalnych preferencjach i zasadach wyboru opcji. Ważne jest, aby rozróżnić dwa potoki analizy działań użytkownika. Po pierwsze działają agenci osobiści, którzy budują profile konkretnych użytkowników systemu, oni zbierają dane tylko o akcjach swojego właściciela. Po drugie, działa agent, który analizuje i generalizuje informacje o globalnych, całościowych trendach. Dla przykładu, agent osobisty zapamięta, że jego właściciel nigdy nie decyduje się pływać promami, natomiast agent zbierający wiedzę ogólną zapamięta, że po kupieniu biletu na samolot do London-Stantstead, użytkownicy często zamawiają bilet kolejowy do centrum Londynu.

Poza tym istnieje podsystem logujący zdarzenia w systemie. Aby baza danych była jak najpełniejsza i aktualna, dane są na bieżąco synchronizowane z zewnętrznymi źródłami danych. Synchronizacja ta jest odpowiednio zarządzana, to znaczy jest przeprowadzana w odpowiednich cyklach, z uwzględnieniem sprzeczności danych.

Od strony gromadzenia i zarządzania informacjami (CCS i CMS) operacje wykonywane przez agentów wyglądają następująco:

CSS to grupa agentów okresowo sprawdzających publiczne źródła danych i dokonujących aktualizacji bazy. Te źródła danych są niezależne od platformy; agenci muszą radzić sobie z wyciąganiem danych ze stron HTML. Dla każdego źródła danych powoływany jest nowy agent (zwany Wrapper Agent – WA), którego zadaniem jest ściągnięcie informacji. Agent taki zna strukturę strony internetowej i potrafi ją sparsować w celu wyłowienia

interesujących go danych. Na dzień obecny, dla każdego zewnętrznego źródła danych powoływany jest osobny Wrapper Agent, znający strukturę stron HTML źródła. W przyszłości, kiedy dane w Internecie będą umieszczane w postaci RDF (a więc dużo bardziej ustrukturalizowanej niż „pływający” HTML), wyciąganie danych z „obcych źródeł” będzie dużo prostsze. Wtedy jedynym zadaniem takiego agenta będzie transformacja danych do wewnętrznie używanej ontologii.

Nad grupą agentów poszukujących danych czuwa podsystem CMS. Jego zadaniem jest koordynowanie Wrapperów, rozwiązywanie konfliktów, ocena źródeł danych itp.

Powyżej zostały omówione podstawowe elementu Platformy. Oprócz wymienionych agentów, istnieją inne, jak choćby agent Proxy, który tłumaczy zapytania użytkownika i zwracane odpowiedzi na odpowiedni język. Użytkownik może bowiem komunikować się z systemem nie tylko za pomocą WWW, może być to też np. WAP dostępny w telefonach komórkowych. Istnieją też agenci służący i koordynujący pracę, agenci pozwalający podglądać działanie całego systemu, agenci testujący i inni.

Jak to się ma do turystyki – Travel Support System

Zaimplementowana przez zespół wersja platformy ma służyć planowaniu podróży. Baza danych zawiera zatem informacje geograficzne, o środkach komunikacji, o restauracjach, wydarzenia kulturalnych i inne. System ma wspierać planowanie podróży w jak największym stopniu. Począwszy od zaplanowania środków lokomocji, jakimi dotrzemy do celu, poprzez terminy, przewoźników, przesiadki, gastronomię, zaplanowanie czasu wolnego aż do podróży powrotnej. W odróżnieniu od klasycznych wyszukiwarek, system agentowy powinien wspierać użytkownika, niejako wyczuwać jego zamiary, podpowiadać rozsądne oferty, wyciągać wnioski z zachowań klientów. Dlatego, jeżeli ktoś pyta o podróż z Warszawy do Bostonu, system powinien dać coś więcej niż tylko listę lotów relacji Warszawa - Boston. System powinien przykładowo podpowiedzieć, ze taniej będzie polecieć do Nowego Jorku, a tam przesiąść się w inną linię, że taniej jest polecieć dzień wcześniej, albo że lecąc dzień później ominiemy tłok związany ze Świętem Dziękczynienia. Chodzi o to, żeby agenci potrafili niejako postawić się w sytuacji użytkownika, żeby sami starali się jak najwnikliwiej zbadać dostępne możliwości.

Dla celów tego systemu został stworzony specjalny moduł – agenci planujący podróż. Opierają się oni na bazie danych o środkach komunikacji (samoloty, pociągi, statki, autobusy...), położeniu geograficznym lotnisk, portów, stacji, przystanków, rozkładach jazdy, czasach przejść między stacjami.

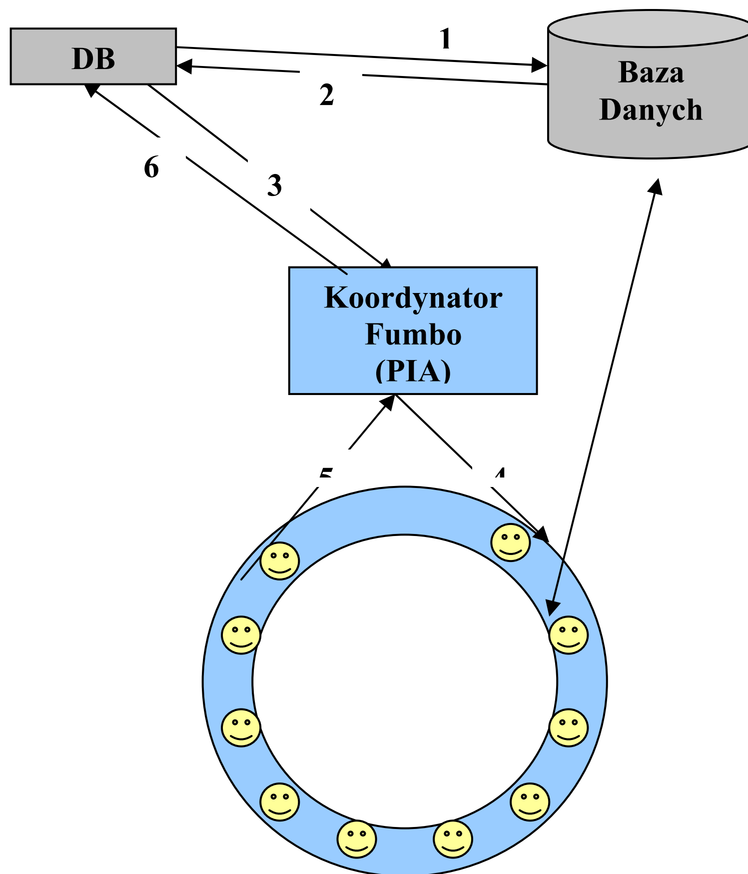
W tej implementacji agenci osobiści gromadzą takie informacje, jak np. fakt, że ten konkretny użytkownik nie lubi jeździć autokarami, że nigdy nie kupuje biletów business-class, że nie jada mięsa i nie lubi musicali. Dzięki temu, odpowiedzi zwracane pytającym w o wiele większym stopniu odpowiadają ich preferencjom.

Agenci rozszerzający zbiór odpowiedzi mogą kierować się takimi zasadami, że ludzie pytający o restauracje chińskie, zadowolą się także kuchnią koreańską, albo że szukający kurortów na Seszelach, zainteresują się również Mauritiusem. Takie zasady, podstawy takiego wnioskowania są budowane na bazie reakcji użytkownika. Jeżeli po pytaniu o wycieczki do Egiptu klient zdecydował się jednak na Tunezję, należy wysunąć hipotezę, że te dwa cele są w jakiś sposób podobne.

Zbieranie danych odbywa się poprzez przeglądanie stron internetowych linii lotniczych, lotnisk, linii kolejowych, przewoźników autobusowych. I tutaj znów agent zarządzający musi decydować, że np. w przypadku terminów bardzo bliskich należy raczej zaufać danym z lotniska, a nie ze stron linii lotniczych (np. ze względu na odwołane loty z powodów atmosferycznych).

Podsystem Rozszerzenia Odpowiedzi – Projekt „Fumbo”

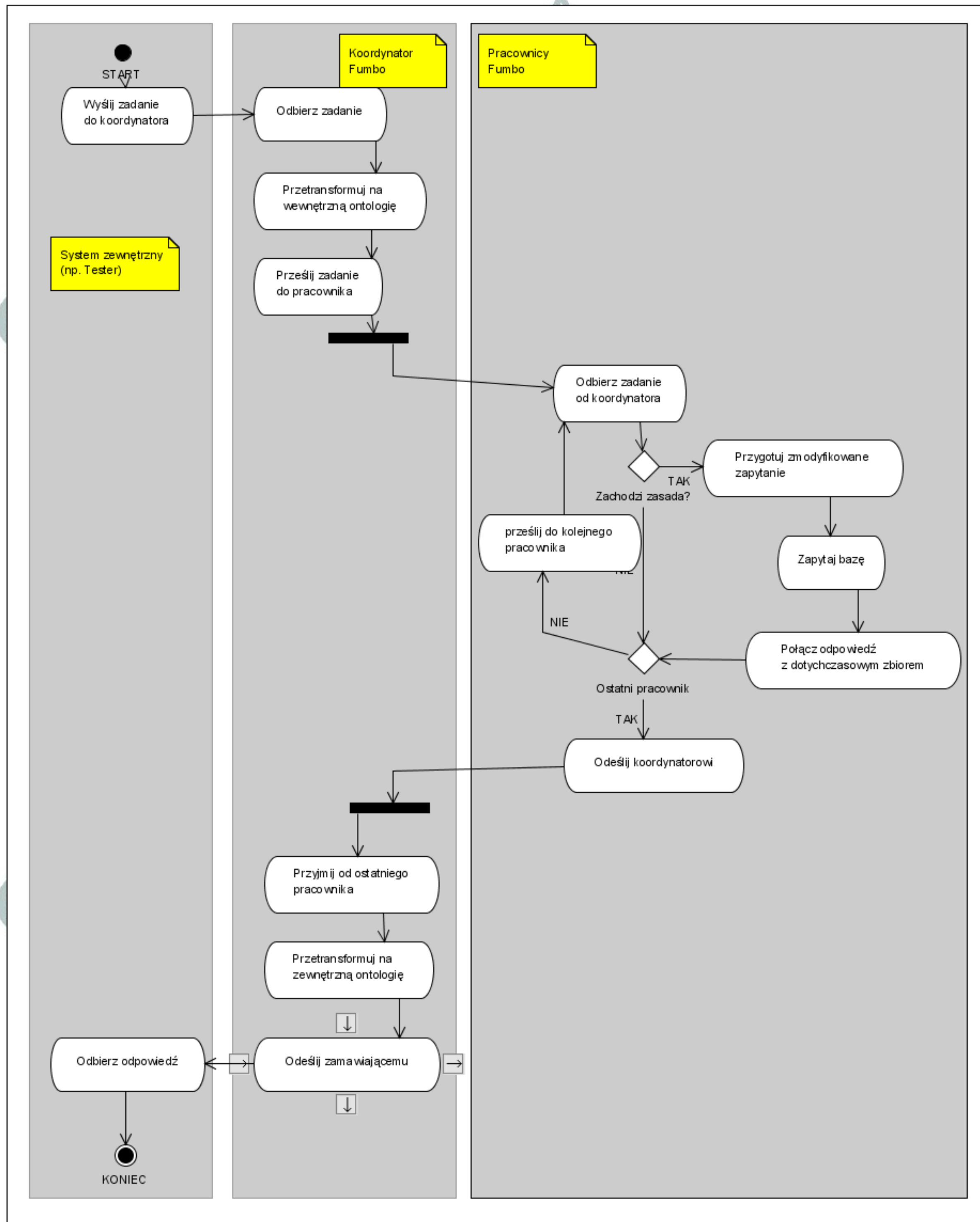
Moją częścią projektu AgentLab jest zespół agentów poszerzających wyjściowy zbiór odpowiedzi i stworzenie Maksymalnego Zbioru Odpowiedzi (MRS). Nazwa tego podsystemu wzięła się z języka Suahili [35], gdzie „Fumbo” oznacza aluzję, ukryte znaczenie, metaforę. Nazwa ta oddaje funkcję podsystemu; jego zadaniem jest odczytanie rzeczywistych (a więc nie tylko ujawnionych bezpośrednio) potrzeb klienta i dostarczenie mu zadowalających odpowiedzi. Mimo, że klient może pytać o kawiarnie, może mieć również na myśli lodziarnie i herbaciarnie; trzeba mu takie możliwości przedstawić. Poniżej zaprezentowany jest wstępny schemat działania mojej grupy agentów.



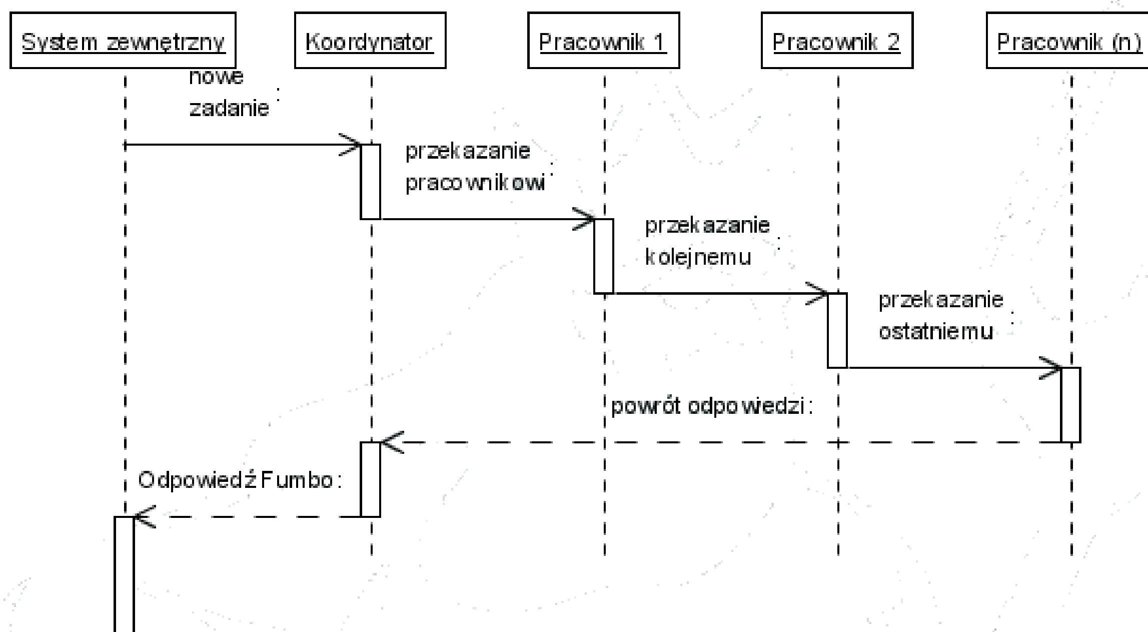
Na rysunku na niebiesko zaznaczone są elementy tworzące projekt Fumbo. Zgodnie ze strzałkami na schemacie, zapytanie trafia najpierw do DBAgenta, następnie jest wysyłane do bazy danych, a potem, wraz z odpowiedzią wysyłane jest do Koordynatora Fumbo. Koordynator (występuje też pod nazwą *PIA - Personalization Infrastructure Agent)* rejestruje powierzone mu zadanie i organizuje pracę innych agentów. PIA posiada listę pracowników – agentów odpowiedzialnych za rozszerzanie odpowiedzi (*Fumbo Workers*). Po odebraniu nowego zadania od agenta bazodanowego, koordynator organizuje rozdział pracy. Możliwych jest kilka topologii pracy agentów - pracowników. W projekcie Fumbo zaprojektowano to jako ring – sekwencyjne przetworzenie zapytania przez wszystkich agentów. Z powierzonego mu zadania koordynator buduje obiekt zadania. Na taki obiekt składa się pytanie zadane przez użytkownika i dotychczas zbudowany zbiór odpowiedzi. Ponadto obiekt zadania zawiera kolejność przekazywania pracy kolejnym agentom i unikalny identyfikator zadania. Zgodnie z treścią zadania i zarządzonym przez koordynatora porządkiem, zadanie jest przekazywane kolejnym agentom. Agenci – pracownicy realizują swoje zadania (dodawanie nowych pozycji i budowanie MRS) na podstawie własnych reguł. Każdy agent – pracownik ma swoją jedną regułę, którą się kieruje podczas tworzenia odpowiedzi. Reguła taka określa w jaki sposób i

co (jeżeli w ogóle) należy dopisać do „rozwiązania” zadania. O regułach będzie mowa w dalszej części rozdziału.

Poniżej przedstawiony jest diagram aktywności podsystemu Fumbo. Diagram jest podzielony na 3 pionowe części (ang. *Swimlines*), które wskazują przez którą część podsystemu czynności są wykonywane.



Poniżej jest także diagram sekwencji dla Fumbo. Jest na nim pokazany schemat przepływu jednego zapytania. Warto zauważyć, że komunikacja między agentami jest asynchroniczna.



Praca Koordynatora

Do obowiązków koordynatora (PIA) należy organizacja pracy agentów i kontakt z resztą systemu.

Po pierwsze, koordynator powołuje do życia i zamyka podległe mu programy agentowe. Sterowanie istnieniem agentów – pracowników polega na zmianach zawartości odpowiedniego katalogu. Koordynator okresowo sprawdza zawartość ustalonego katalogu. W tym katalogu istnieją pliki z regułami dla agentów – pracowników. Jeżeli od ostatniego sprawdzenia w katalogu pojawił się nowy plik, nowy agent jest uruchamiany. Jeżeli jakiś plik został skasowany, odpowiadającego agenta się „uśmierca". Oczywiście koordynator trzyma aktualną listę działających agentów. Administracja całym podsystemem odbywa się za pomocą sterowania koordynatorem. Koordynator opiera swoją konfigurację na danych zawartych w odpowiednio umieszczonym pliku konfiguracyjnym. Administracja logiką działania podsystemu odbywa się poprzez włączanie / wyłączanie agentów – pracowników,

co, jak było wspomniane, jest wykonywane poprzez dodawanie / usuwanie plików reguł. Przez cały okres działania systemu koordynator jest w pełni odpowiedzialny za pracowników. Żaden agent – pracownik (Worker) nie może istnieć niezależnie czy samodzielnie. Nawet w momencie śmierci pracownika (np. kiedy agent pracownik nie może pracować z powodu błędu), pracownik jest zabijany przez koordynatora. W momencie wykrycia błędu, z którym pracownik nie może sobie poradzić, wysyła on do koordynatora żądanie uśmiercenia, które jest po chwili wykonywane. W ten sposób koordynator ma zawsze aktualną listę dostępnych pracowników.

Drugi zadaniem koordynatora jest interakcja z resztą systemu. Koordynator pełni rolę interfejsu, bramy na świat. Poprzez koordynatora do Fumbo trafiają nowe zapytania i poprzez niego zwracane są odpowiedzi. Tak jak było to przedstawione na schemacie, nikt poza koordynatorem nie ma dostępu do agentów – pracowników. Dzięki temu, na wysokim poziomie całych programów agentowych rozdzielony jest interfejs od implementacji projektu. Gdyby z jakichś przyczyn zaszła potrzeba zmian w działaniu pracowników, dla innych części systemu byłaby ta zmiana przeźroczysta. Wystarczy, że pozostałaby nie zmienna ontologia do komunikacji z koordynatorem. Wprowadza to możliwości zmiany implementacji (np. wykorzystanie innych topologii szeregowania pracowników) a także bezpieczniejszego testowania. Jeżeli bowiem napiszemy agenta-testera, (który, notabene, został napisany), po zmianie implementacji pracowników wystarczy sprawdzić, czy system działa równie dobrze jak przed zmianami. Ponieważ jedynym interfejsem używanym w Fumbo do komunikacji ze światem zewnętrznym jest ontologia koordynatora (tzw. External Ontology), przeniesienie projektu do innego frameworku byłoby stosunkowo proste. Wystarczyłoby bowiem zaimplementować nową ontologię zewnętrzną, wewnętrzna komunikacja pozostałaby bez zmian.

Praca koordynatora nad jednym zapytaniem wygląda następująco:

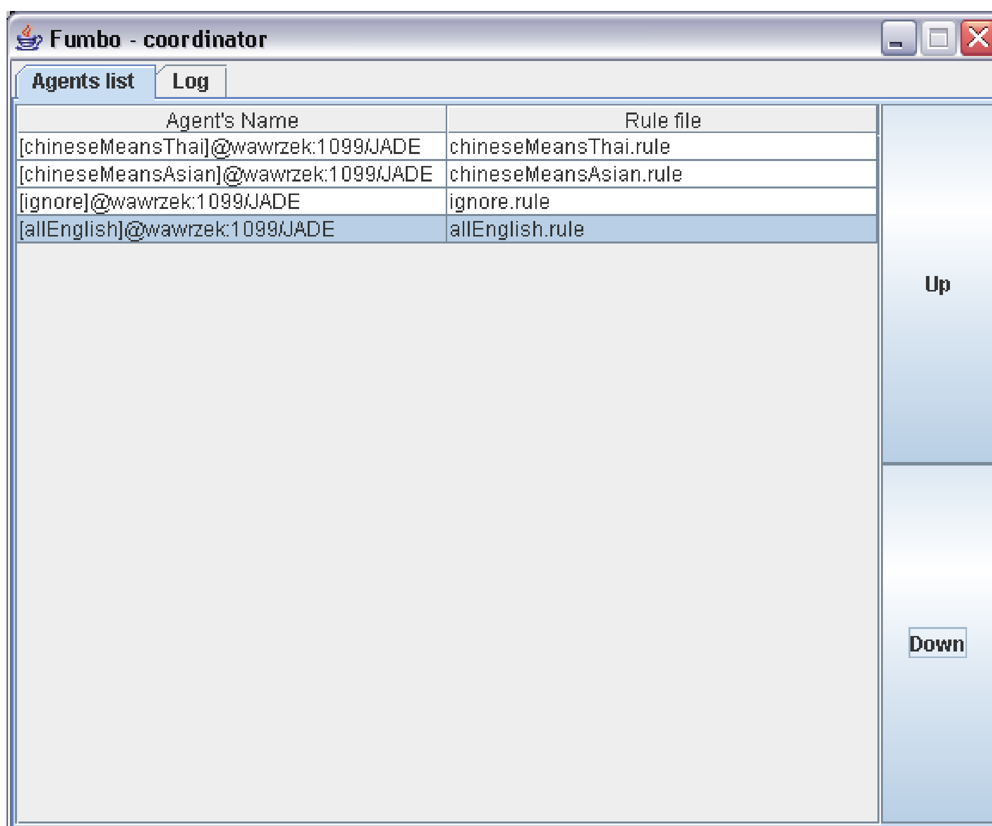
1. Po otrzymaniu zapytania, koordynator rejestruje je na swojej liście nadając mu unikalny identyfikator (w rzeczywistości kolejny numer)
2. Koordynator ustala kolejność pracy pracowników
3. Tworzy obiekt zadania
4. Obiekt zadania jest serializowany przy pomocy wewnętrznej ontologii podsystemu Fumbo (tak, aby był prawidłowo rozumiany przez pracowników)
5. Obiekt zadania jest wysyłany do pierwszego pracownika na liście

6. Po pewnym czasie (kiedy pracownicy wykonają już swoje obowiązki), koordynator otrzymuje odpowiedź.
7. Otrzymana odpowiedź jest wiązana z pytaniem (oczywiście po identyfikatorze)
8. Odpowiedź jest wysyłana do agenta (spoza Fumbo), od którego nadeszło zapytanie.

Warto podkreślić, że koordynator działa asynchronicznie. Agent ten nasłuchuje na pytania z zewnątrz oraz na odpowiedzi i komunikaty administracyjne od pracowników. Po otrzymaniu zapytania, skonstruowaniu zadania i wysłaniu go do pracowników, koordynator niejako zapomina o nim. Koordynator może zbierać odpowiedzi w dowolnej kolejności, niekoniecznie takiej, w jakiej pytania zostały wysyłane.

Dodatkowo, koordynator może posiadać interfejs graficzny (GUI) w postaci okna z informacjami na temat pracy podsystemu. Jest wtedy pokazywana lista aktualnie pracujących agentów, a także log z prowadzonych przez nich działań. Dla celów wydajnościowych, można z GUI zrezygnować i śledzić pracę czytając log. Na głównym panelu GUI koordynatora umieszczono dwa przyciski umożliwiające zmianę kolejności pracowników w ringu. Taka zmiana może znacznie wpłynąć na rozwiązanie.

Poniżej przedstawiony jest zrzut ekranu z GUI koordynatora:

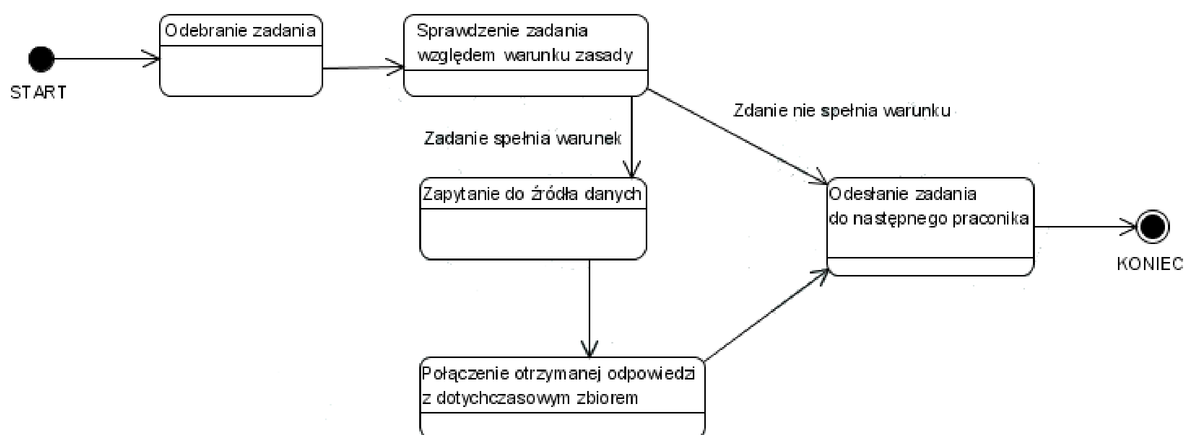


Praca agenta-pracownika

Zadaniem agent „pracownika” jest uzupełnienie odpowiedzi o, według niego, potrzebne informacje. Uzupełnienie to opiera się na trzech źródłach informacji:

1. Zadanym przez użytkownika pytaniu
2. Zbudowanym do tej pory zbiorze odpowiedzi
3. Zasadzie (Rule) kierującej agentem

Operacje przeprowadzanie przez jednego agenta – pracownika są zgodne z poniższym diagramem:



Każdy agent-pracownik (Worker) działa w oparciu o pewną zasadę. Jak było wspomniane powyżej, z każdym agentem-pracownikiem jest związany plik z zasadą. Plik taki jest podstawą działania agenta. Życie pracownika jest do tego stopnia związane z plikiem zasady (Rule File), że poprzez tworzenie / kasowanie plików powołuje się do życia i uśmierca agentów.

Plik zasady jest to plik tekstowy w którym zapisane są wytyczne co do pracy agenta. Taka zasada jest zawsze zbudowana według poniższego schematu:

```
IF <warunek> THEN <polecenie>
```

Jak można się domyślić, agent wykona `<polecenie>`, jeśli spełniony będzie `<warunek>`. Za pomocą takich zasad określa się funkcjonalność systemu.

Parsowane są trzy postaci wyrażenia <warunek>

1. TRUE (warunek spełniony zawsze, reguła zawsze obowiązuje)
2. FALSE (reguła nie obowiązuje nigdy – zaimplementowane dla symetrii)
3. COUNT

Przy pomocy warunku COUNT określa się warunek na liczność już uzyskanego zbioru odpowiedzi. Łatwo można sobie wyobrazić komendy do wykonania, jeżeli nasz zbiór jest za mały. Przykładowo, jeżeli uzyskano mniej niż 3 pozycje spełniające kryteria, próbujemy dodać nowe dane na temat pozycji bliskich założeniom.

Wyrażenie `<polecenie>` może być napisane w jednej z trzech postaci:

1. `RDQL`(*kwerenda*)
2. `MEANS`(*znaczenie1*, *znaczenie2*)
3. `IGNORE`(*część*, *nazwa*)

W pierwszym przypadku (`RDQL`) podajemy explicite zapytanie w języku RDQL, które agent powinien zadać bazie danych. Wtedy do zbioru odpowiedzi są dopisywane pozycje spełniające zadane w komendzie (w zapytaniu) kryteria. Ponieważ jednak trudno w ten sposób tworzyć bardziej ogólne i uniwersalne reguły, wprowadzono dwie inne możliwości tworzenia zapytań.

Schemat `MEANS`(,) oznacza, że wprowadzamy pewien synonim. Jako argumenty polecenia `MEANS` podaje się dwa wyrazy; wprowadzane jest drugie znaczenie pewnego słowa. Agent-pracownik odpytuje więc bazę danych pierwotnym zapytaniem (tym samym, co odpytywał ją agent bazodanowy), lecz z zastosowaniem drugiego znaczenia. Ta funkcjonalność odpowiada przykładowi z kuchniami orientalnymi. Jeżeli pierwotne pytanie dotyczył kuchni chińskiej, możemy mieć komendę `MEANS`(chińska, koreańska). Wtedy agent wyśle do bazy danych to samo zapytanie, ale ze zmienioną „kuchnią”.

W rzeczywistości, jak widać, w komendzie MEANS nie trzeba stosować wyłącznie synonimów, mogą być to rozszerzenia jakichś pojęć np. MEANS(toskańska, włoska), albo po prostu jakieś powiązania np. MEANS(japońska, sushi).

Trzecią możliwą komendą jest komenda IGNORE. Komenda ta dotyczy obostrzeń w klauzuli WHERE zapytania. Agent wysyła do bazy danych to samo zapytanie, które było wysyłane pierwotnie, jednakże z usuniętym jednym kryterium. Jako argumenty komendy podaje się

1. Część zdania – jedną ze stałych {SUBJECT, PREDICATE, OBJECT}
2. String – URI lub wartość

Agent zada pierwotne pytanie, ale z klauzuli WHERE usunie kryterium dotyczące wskazanej części zdania, o ile pokrywa się ona z podaną wartością. Dla przykładu, można zadać to samo pytanie o restauracje, ale ignorować dostępność miejsc dla palących, albo też zapytać o bilety do Nowego Jorku już bez względu na linię lotniczą. Komenda ta ma duże zastosowanie razem z warunkiem COUNT; można tym zestawem budować zady typu „jeżeli odpowiedzi jest mniej niż 10, powtórz pytanie bez precyzowania przedziałów cenowych”.

W celu przybliżenia działania agentów – pracowników, przedstawione są poniżej przykładowe zasady, które mogą decydować o ich pracy:

```
IF COUNT<10 THEN
RDQL(
      SELECT ?x
      WHERE (?x <http://www.w3.org/2001/vcard-rdf/3.0#FN> "ABC")
 )
```

Zasada ta mówi, żeby jeżeli odpowiedz jest mniej niż 10, należy dołączyć do nich wszystkie obiekty, których własnością <http://www.w3.org/2001/vcard-rdf/3.0#FN> jest „ABC”

Drugi przykład:

```
IF TRUE THEN MEANS( CHINESE, SECHUAN )
```

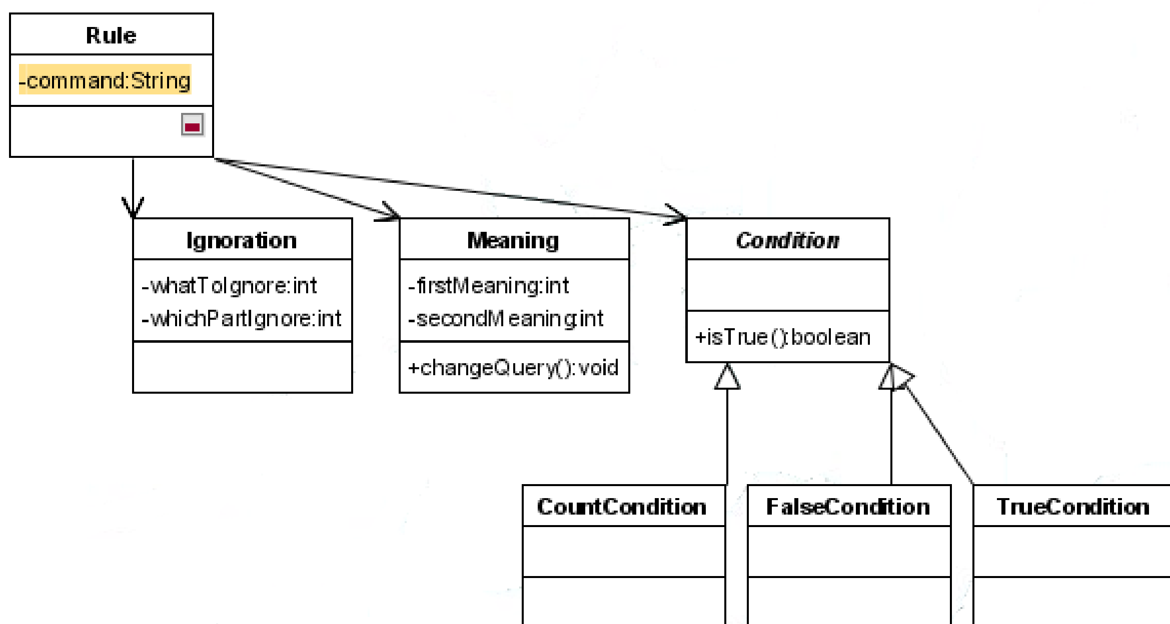
Ta zasada mówi, że „chińska” oznacza także „seczuańska”. Czyli, jeżeli ktoś szuka restauracji chińskich, należy podpowiedzieć mu także te seczuańskie.

Trzeci Przykład:

```
IF COUNT<10 THEN IGNORE(PREDICATE, http://www.w3.org/2001/vcard-rdf/3.0#FN)
```

Ta zasada mówi, że jeżeli mamy mniej niż 10 odpowiedzi, należy powtórzyć pytanie, ale rezygnując z warunku, w którym orzeczenie wynosi <http://www.w3.org/2001/vcard-rdf/3.0#FN>, co w praktyce oznacza, żeby wyszukać w bazie jeszcze raz, rezygnując z zawężania odpowiedzi po nazwie.

Działanie agenta-pracownika jest zrealizowane przy pomocy klas przedstawionych na poniższym diagramie:



Każdy pracownik posiada zasadę nim kierującą. Zasada zawiera obiekty Condition, Command, Ignoration, Meaning, czyli, odpowiednio:

- Warunek sprawdzający, czy zasada ma zastosowanie
- Komenda RDQL do wykonania
    - lub polecenie Ignorowania części zapytania
    - lub nowe znaczenie pewnego Stringa.

Jak już wspomniano, włączanie/wyłączanie agentów odbywa się poprzez umieszczanie i kasowanie plików z zasadami w odpowiednim katalogu. Ważną własnością

frameworku jest fakt, że takie pliki z zasadami nie muszą być tworzone ręcznie przez administratora. System ekspercki, który obserwuje wszystkie poczynania użytkowników, analizuje je i na ich podstawie stara się wytwarzać reguły, co do których powinna stosować się ta rozbudowana wyszukiwarka. Jeżeli wielu użytkowników po pytaniu o pociągi do Zakopanego decyduje się jednak na pociąg do Krakowa, a stamtąd autobus, system mógłby podpowiadać takie rozwiązanie od razu. Śledzenie takich poczynań i tworzenie nowych zasad rządzących agentami jest ważnym elementem całego systemu.

Podsumowując, agenci – pracownicy kierują się zasadami. Każda zasada składa się z warunku i komendy. Zaimplementowane warunki to TRUE, FALSE i COUNT. Zaimplementowane komendy do RDQL, MEANS i IGNORE. Wydaje się, że taka kombinacja umożliwia pisanie uniwersalnych (to znaczy nie koniecznie związanych z turystyką), lecz także szczegółowych zasad kierowania agentami.

Wprowadzenie metajęzyka w programowaniu agentami ma wiele zalet. W pisaniu logiki systemu można odejść od szczegółów implementacji i przejść bliżej w stronę dziedziny problemu. Opracowują logikę, można myśleć nie o tym jak agent ma działać, lecz co agent ma zrobić. Co więcej, programowanie agentów nie wymaga rekompilacji kodu, ani restartu systemu. Przy dużych projektach internetowych, kiedy musi być zapewniona wysoka dostępność systemu ma to bardzo duże znaczenie.

Możliwości rozwoju projektu Fumbo

Jak wspomniano wcześniej, zadanie jest przetwarzane przez agentów-pracowników sekwencyjnie. Można także stosować inne topologie. Jednym z pomysłów jest topologia gwiazdy. Koordynator mógłby tworzyć w wyznaczonym na serwerze miejscu tymczasowy plik RDF, który byłby wyjściowym zbiorem odpowiedzi. Każdy z agentów-pracowników otrzymywałby poprzez komunikat ACL zadanie do wykonania, wraz z adresem tymczasowego zbioru odpowiedzi. Agenci mogliby wtedy wytwarzać swoje odpowiedzi i wysyłać je do koordynatora, który aktualizowałby ostateczną odpowiedź. Takie rozwiązanie miałoby wady i zalety. Zaletą byłoby nieprzesyłanie tymczasowych rozwiązań wszystkim agentom, pracownicy mogliby też pracować równolegle. Niestety, i tak jeden agent (zapewne koordynator) musiałby zanalizować wszystkie odpowiedzi pracowników, aby wytworzyć jedną spójną, nie zawierającą powtórzeń odpowiedź. Co do równoległości działań, topologa

ringu nie wyklucza jej; nic nie stoi na przeszkodzie, aby podczas gdy piąty agent zajmuje się pierwszym zadaniem, pierwszy agent już zajmował się drugim. Koordynator nie musi dostawać odpowiedzi w tej samej kolejności, w jakich wysyłał zapytania. Czyli równoległość pracy de facto istnieje.

Topologia ringu niesie ze sobą także inne konsekwencje. Należy zauważyć, że kolejność ustawienia agentów może mieć znaczenie dla treści odpowiedzi. Jeżeli są agenci, którzy kierują się zasadami opartym na warunku `COUNT`, w zależności od kolejności wykonania, zadanie może mieć różne rozwiązania. Dla przykładu, pierwszy agent może na tyle rozszerzyć odpowiedzi, że żaden inny agent nic już nowego nie wniesie. A być może bez tego pierwszego agenta odpowiedzi byłyby trafniejsze. Te własności łatwo prześledzić sterując kolejnością pracowników poprzez GUI koordynatora (co będzie pokazane w późniejszym rozdziale)

Rozwiązaniem tego problemu byłoby zapewne wprowadzenie ocen pracy agentów. Wtedy byliby oni sortowani w ringu według oceny – według trafności zasad, jakimi się kierują. Takie oceny mogłyby być również generowane przez czuwający nad wszystkim system ekspercki.

Konfiguracja systemu Fumbo

Podsystem Fumbo konfiguruje się za pomocą wpisów w pliku config/fumbo.cfg. Poniżej opisane są przykładowe dopuszczalne wpisy i ich oznaczenia:

- logfilename = "agents.log" – nazwa pliku do którego logowane są zdarzenia w systemie
- rulesdirectory = "rules" – nazwa katalogu z plikami reguł do tworzenia agentów – pracowników
- dataurl = "file:data\ex1.rdf" – adres URL źródła danych
- watchLog = 0 – czy koordynator ma wyświetlać log systemu
- workersPeriod = 1000 – okres odświeżania listy agentów – pracowników (w milisekundach)

Do konfiguracji modułu logującego Log4J (o którym będzie mowa w następnym rozdziale) służy osobny plik, config/log4j.cfg

Wykorzystywane moduły, biblioteki, narzędzia

Do pisania projektu Fumbo, a także do tworzenia całego frameworku AgentLab, wykorzystywanych jest wiele narzędzi zewnętrznych. Celem zespołu było wytworzenie oprogramowania rzeczywiście działającego. Chcieliśmy wykorzystując istniejące technologie pójść krok dalej i stworzyć pewną nową, większą całość. O ile nam wiadomo, na świecie nie istnieje rozbudowany i równie złożony agentowy system obsługi podróży. Korzystając z dostępnych narzędzi, taki system udało nam się stworzyć.

Programowanie

Cały projekt jest napisany w języku Java [36]. Język ten, ze względu na swoje mocne korzenie „obiektowe", a także mnogość dostępnych narzędzi, nadaje się do tego projektu idealnie.

Do tworzenia wersji używana jest biblioteka ant [37]. Pozwala ona zautomatyzować proces przygotowania wersji – kompilacji, pakowania archiwów, kopiowania plików konfiguracyjnych, przygotowania skryptów uruchomieniowych. Aby korzystać z biblioteki ant, tworzy się jeden plik konfiguracyjny opisujące zadania i procesy zachodzące w procesie tworzenia wersji (tak jak pliki make). Po stworzeniu takiego skryptu, utworzenie wersji wymaga jednego polecenia.

Do testowania klas i modułów zastosowano rozwiązanie JUnit [38]. Jest to narzędzie wspomagające testowanie oprogramowania napisanego w Javie. Aby kontrolować poprawność kodu, pisze się tak zwane Test-Case'y (Przypadki Testów) – funkcje testujące nasz program. Jeżeli na początku procesu wytwarzania oprogramowania napisze się dobre testy, proces ten można znaczenie przyśpieszyć, a program uczynić bardziej niezawodnym. JUnit pozwala zautomatyzować testowanie – uruchamiać automatycznie kolejne testy i oglądać jedynie ich wyniki. Ponieważ platforma AgentLab była pisana przez wiele osób oddzielnie, odpowiednie procedury testujące były bardzo ważne. Chodziło o to, aby odrębne

części systemu można było zbadać i ocenić niezależnie od całości. Mając dobrze przetestowane elementy, podsystemy, można było myśleć o łączeniu ich w całość.

Aby programowanie było szybkie i efektywne, należało do tego użyć dobrego środowiska programistycznego. Poza narzędziami ant i JUnit, należało mieć platformę programistyczną wspierającą język Java. Tutaj znakomitym narzędziem był program Eclipse [39]. Nie dość, że zawiera on dobry edytor, kompilator, to wspiera on stosowanie przeze mnie rozwiązania (ant i JUnit).

Środowisko Agentowe JADE

Jak było wspomniane w rozdziale dotyczącym systemów agentowych, niezbędne jest posiadanie środowiska uruchomieniowego agentów. W tym projekcie wybór padł na środowisko JADE [40]. Jest to środowisko w całości zaimplementowane w języku Java i spełniające wszystkie kryteria organizacji FIPA. Spełnia ono wszystkie wspomniane wymagania stawiane środowiskom agentowym:

- pozwala swobodnie tworzyć, zabijać i zmieniać stany agentów
- implementuje usługę przesyłania komunikatów ACL
- implementuje usługę katalogową
- daje możliwość migracji agentów na inne maszyny
- daje wygodny interfejs użytkownika do celów testowych i debugowych
- daje kilka przykładowych agentów wspierających debugowanie, np.
  - Sniffer (podgląd wszystkich wymienianych komunikatów)
  - Introspector Agent – podgląd stanu agenta – jego zachowań, działań
- Jest dobrze udokumentowane

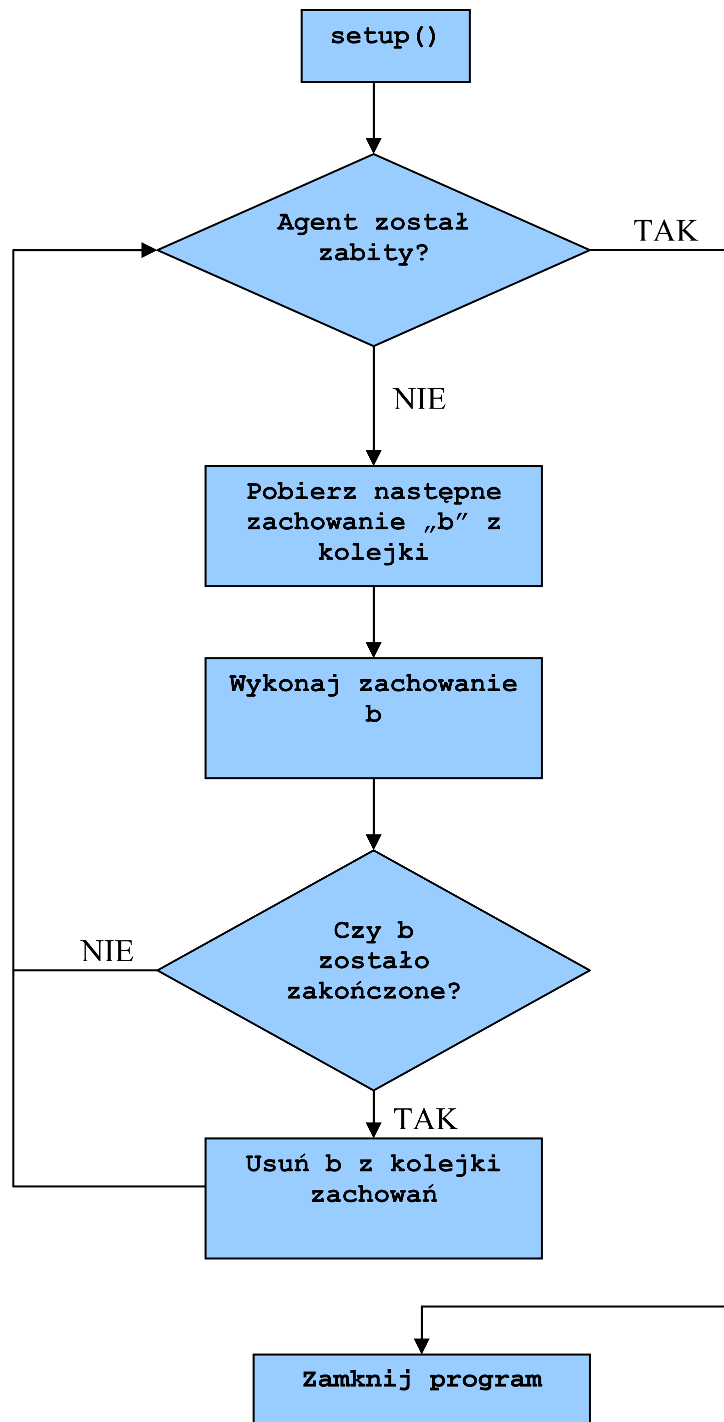
W środowisku JADE, pisanie agenta rozpoczyna się od utworzenia klasy rozszerzającej klasę Agent. Aby takiego agenta uruchomić (który oczywiście jeszcze nic nie będzie robił), należy uruchomić środowisko Jade z odpowiednimi parametrami. Aby agentowi dać pewną funkcjonalność, implementujemy pewne ważne metody klasy Agent – np. metodę setup(), która będzie wywołana na starcie agenta.

To, co odróżnia pisanie agenta od pisania innych aplikacji, to filozofia pracy programu agentowego. W ciągu całego swojego życia, agent oczekuje na pewne bodźce i na nie reaguje.

Aby to zaprogramować, używa się tzw. zachowań (behaviours). Należy napisać kod tworzący takie zachowanie i podłączyć na jeden z kilku sposobów do cyklu życia agenta. Bardzo ważne jest, aby wiedzieć, że agent w środowisku JADE jest całkowicie sekwencyjny. Nie ma możliwości tworzenia własnych wątków. Dzięki temu nie ma potrzeby dbania o bardzo wiele spraw, unika się potencjalnych błędów synchronizacji i dostępu do zasobów, a przede wszystkim, programy agentowe są znacznie szybsze. Istnieją możliwości przełączania się między „równoległymi" zachowaniami agenta, ale leży to w gestii programisty. W JADE istnieje zasada jednego wątku na agenta. Przy dużej liczbie agentów jest to bardzo dobra zasada; poza tym przełączanie między zachowaniami agenta jest znacznie szybsze niż przełączanie maszyny między wątkami Javy.

Zatem działanie agenta polega na wykonywaniu zleconych mu zachowań. Zachowania takie mogą być zlecone na początku życia agenta (w metodzie setup()), lub w dowolnym innym momencie, np. w odpowiedzi na jakieś zdarzenie. Zlecenie zachowania polega na wywołaniu jednej funkcji, która dopisuje je do aktualnej kolejki zachowań. Każdy agent ma swoją kolejkę zachowań. Według niej wykonywane są wszystkie działania programu.

Cykl działania programu agentowego wygląda zatem następująco:



Istnieje kilka typów zachowań:

1. Zachowania jednorazowe (OneShotBehaviour)
2. Zachowania cykliczne (CyclicBehaviuor)
3. Zachowania okresowe (TickerBehaviour)

4. Zachowania własne, niestandardowe

Zachowania jednorazowe programuje się dla akcji mających wystąpić raz w danym kontekście. Przypominają one zwyczajne funkcje. Wywołanie zachowania (a de facto dodanie go do kolejki) ma podobny skutek, co wywołanie funkcji czy metody. Za pomocą zachowań jednorazowych implementuje się zadania do wykonania dla agenta. Przykładowo, dla agentów – pracowników w projekcie Fumbo, zachowaniem jednorazowym jest rozszerzenie zbioru odpowiedzi w myśl pewnej zasady.

Zachowania cykliczne to zachowania, które wywoływane są na okrągło, raz za razem – podobnie jak pętle. Zachowań cyklicznych często używa się jako pętli oczekiwania na pewne komunikaty. Agent czeka w pętli na jakieś zlecenie; kiedy je dostanie, do kolejki zachowań dorzuca zachowanie rozwiązujące to zlecenie.

Zachowania okresowe to zachowania, które zostaną przez środowisko wywołane raz na określony czas. W projekcie Fumbo agent – koordynator używa zachowań okresowych do sprawdzania zawartości katalogu z regułami. Jeżeli pojawił się tam nowy plik – tworzony jest nowy agent. Jeżeli plik zniknął – agent jest uśmiercany.

Zachowania własne mogą być bardziej wyspecjalizowane działanie i ich wywoływanie można określić samemu.

Podsumowując, pisanie agenta w środowisku JADE polega na:

1. Zdefiniowaniu i zarejestrowaniu zachowań w metodzie setup().
2. Reagowaniu na komunikaty za pomocą zachowań cyklicznych.
3. Realizacji własnych zadań poprzez uruchamianie nowych zachowań.

Platforma JADE to jednak nie tylko środowisko uruchomieniowe agentów. JADE daje również rozbudowane biblioteki do budowania agentów. Jak wspomniano, pisanie agenta polega na pisaniu klas dziedziczących np. po klasie Agent czy klasach zachowań. Dodatkowo, JADE daje możliwości łatwej komunikacji między agentami. Wysłanie czy odebranie komunikatu odbywa się poprzez wywołanie jednej funkcji. Podobnie, JADE wspiera używanie ontologii do budowy treści komunikatów. Mając zbudowane klasy ontologii, za pomocą jednego polecenia tworzymy tekst komunikatu na podstawie złożonych obiektów.

Biblioteka Jena

Biblioteka Jena [41] jest platformą do budowy sieci semantycznych. Wspiera ona programowanie aplikacji z wykorzystaniem technologii RDF, RDFS [42], OWL. Projekt ten narodził się w laboratoriach firmy HP, jako HP Labs Semantic Web Programme [43] (http://www.hpl.hp.com/semweb/), w tej chwili jest rozwijany jako projekt Open Source.

Główne pola działania biblioteki Jena to:

1. API do działania z danymi w postaci RDF
2. Pisanie i czytanie z plików RDF w formie XML, N3 i N-triples
3. API do działania z danymi w języku OWL
4. Trzymanie danych zarówno w pamięci, jak i w formie trwałej
5. Obsługa języka RDQL

Jena pozwala przetwarzać dane w formacie RDF na wysokim poziomie. Za pomocą pojedynczych poleceń narzędzie to potrafi wczytać dane z pliku XML, a także do niego zapisać. Kiedy takie dane są już w pamięci (dostępne poprzez klasę Model), programista operuje na nich jak na grafie. W momencie wczytania danych, nie istnieje już kwestia składni RDF; informacje są dla nas dostępne w postaci zdań - trójek (Podmiot, Orzeczenie, Przedmiot). Można tworzyć własne zdania i dodawać je do modelu, także wykorzystując puste węzły (blank nodes). Jena pozwala skupić się programiście na modelowaniu danych i relacji między nimi i nie martwić się o zapis tego modelu.

Biblioteka ta pozwala przechowywać dane zarówno w pamięci podręcznej, jak i trwałej – np. jako pliki lub w relacyjnej bazie danych. Oczywiście można także takie dane wygodnie przetwarzać – istnieją funkcje do wyławiania informacji na temat zadanego Podmiotu, związanych z jakimś przedmiotem, lub będących z innymi w pewnej relacji. Oprócz stosowania bezpośredniego dostępu do danych (z poziomu kodu javy), Jena pozwala dostać się do nich poprzez język RDQL. Zapytania RDQL można wywoływać z kodu, lub uruchamiając niewielki program z linii komend. W projekcie Fumbo szeroko stosowane jest to pierwsze rozwiązanie. Agent-pracownik dostaje do wykonania zadanie i zazwyczaj sprowadza się to do wykonania zapytania RDQL. Zapytanie to jest budowane na podstawie pierwotnego zapytania użytkownika i odpowiednio zmieniane w myśl zasady rządzącej

agentem. Po uruchomieniu zapytania, agent dostaje zbiór odpowiedzi dotyczących danego Podmiotu. Następnie zadawane jest jeszcze pytanie o wszystkie cechy tego podmiotu, tak, aby przekazać do użytkownika jak najwięcej danych.

Oprócz działań na pojedynczych zdaniach, Jena wspiera także działania na całych modelach (grafach, zbiorach danych). Przykładowo, Jena pozwala łączyć dwa modele. W projekcie Fumbo jest to także wykorzystywane. Agent – pracownik, po zadaniu bazie danych swojego zapytania, łączy (union) otrzymane odpowiedzi ze zbiorem odpowiedzi „uzbieranych" już wcześniej. Biblioteka Jena dba o to, aby dane te nie powtarzały się w modelu. Oznacza to, że w grafie istnieją co najwyżej pojedyncze krawędzie między węzłami (czyli zdania Podmiot - Orzeczenie - Przedmiot nie powtarzają się).

Moduł logujący Log4J

Aby skutecznie śledzić pracę systemu, należy zapewnić prawidłowe logowanie zdarzeń. Standardem w projektach pisanych w języku Java stał się moduł Log4J [44]. Jest on dla programisty bardzo prosty w obsłudze i konfiguracji, a jednocześnie dostarcza silne narzędzia logujące.

Podstawowym zadaniem logowania jest wypisywanie na określone wyjście komunikatów pochodzących od aplikacji. Log4J pozwala szeregować komunikaty i podzielić je na 5 podstawowych typów:

1. Fatal Error – błąd krytyczny – Aplikacja nie może już w żaden sposób pracować
2. Error – błąd aplikacji
3. Warning – Ostrzeżenie – Aplikacja może pracować, ale istnieje pewne niebezpieczeństwo
4. Info – komunikat informacyjny
5. Debug – komunikat dla programisty na temat działania programu

Wypisując każdy komunikat, programista określa jego poziom. Dzięki temu, można na poziomie tworzenia programu śledzić wszystkie najdrobniejsze komunikaty, a podczas działania produkcyjnego wysyłać np. tylko błędy.

Log4J jest modułem wysoce konfigurowalnym. W pliku konfiguracyjnym można określić poziom logowania, a także jego wyjścia. Można wypisywać komunikaty np. do pliku,

na konsolę, do bazy danych, czy też poprzez wysłanie maila. Różne konfiguracje można stosować do różnych poziomów logów. Przykładowo można określić, że informacje na poziomie Debug lądują w pliku, ostrzeżenia są zapisywane w bazie danych, a błędy są oprócz tego natychmiast wysyłane mailem.

Dane

Aby skutecznie testować oprogramowanie, potrzebne były duże zasoby danych w postaci RDF. Ponieważ cały system ma mieć zastosowanie w turystyce, dobrze, aby te dane były związane z tą gałęzią gospodarki. Dodatkowo, oczywiście ważne jest, aby te dane były w formacie, którego dotyczy ta praca, czyli jako RDF.

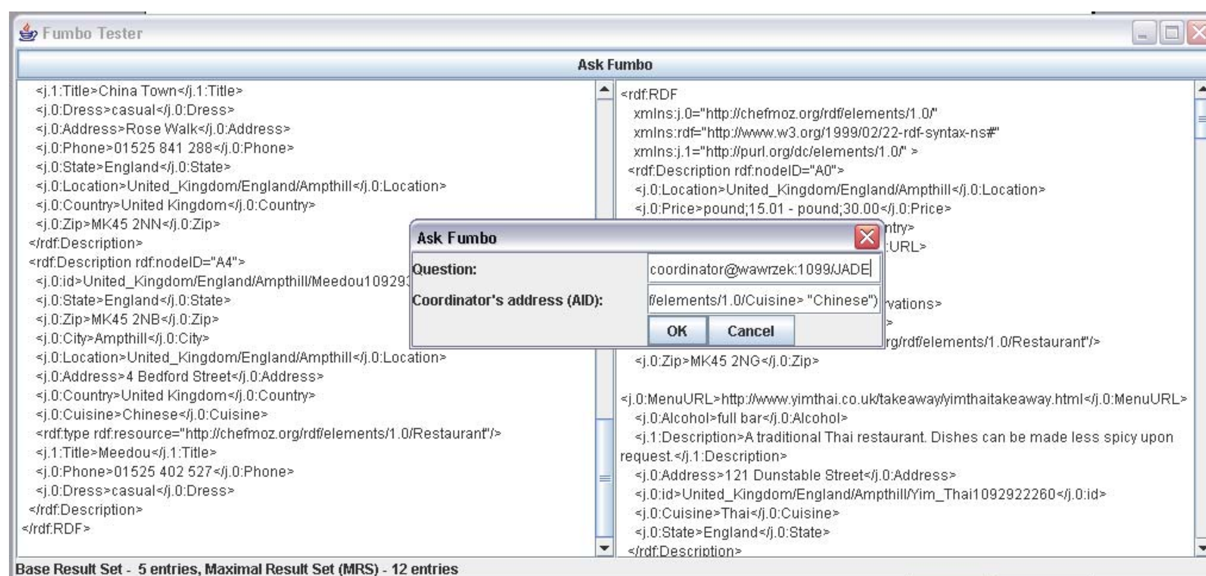
Dane takie znaleźliśmy na stronach projektu ChefMoz [45] Jest to projekt, którego celem jest zbieranie informacji o wszystkich światowych restauracjach, barach, pubach, kawiarniach. Opiera się on na pracy wielu ochotników z całego świata; każdy może do systemu wprowadzać swoje recenzje lokali i je oceniać. Dzięki temu baza danych liczy ok. 240 000 obiektów. Plik XML zawierający dane jest do ściągnięcia za darmo. Niestety zawiera on dużo błędów – nie jest poprawnie sformatowany, zawiera znaki niedozwolone, nie jest w kodowaniu UTF-8. Jeżeli weźmiemy pod uwagę jego rozmiar (ok. 200 MB) nie jest rzeczą prostą przekonwertować te dane do poprawnej formy.

We wczesnych stadiach implementacji aplikacji, do celów testowania korzystałem też z innych źródeł danych – nie związanych z turystyką. Jak było wspomniane, zamysłem przy tworzeniu frameworku AgentLab było stworzenie platformy dosyć ogólnej, łatwo dostosowywalnej do innych dziedzin życia niż turystyka. W rzeczywistość projekt Fumbo nie zakłada, że działa w systemie wspomagania podróży i może operować na dowolnych danych w formacie RDF. Przykładowo, posiłkowałem się plikiem zawierającym opisy ok. 10.000 enzymów. Bardzo dobry spis adresów do baz danych w formacie RDF znajduje się na [46].

Jak było napisane na początku pracy, do upowszechnienia standardu RDF i semantycznego przetwarzania danych konieczny jest rozwój źródeł danych. Potrzeba, aby dane zaczęły być przechowywane w sposób zrozumiały także dla maszyn, aby były opisane za pomocą uzgodnionych oznaczeń (ontologie). W ten sposób będą one w dużo większym stopniu przetwarzane i szerzej dostępne. Jak można zobaczyć pod podanym powyżej adresem, takie źródła danych już powstają.

Testy systemu

W celu testowania i prezentacji podsystemu Fumbo został napisany specjalny agent testujący. Symuluje on działanie pozostałych komunikujących się z Fumbo części Systemu. Agent ten posiada graficzny interfejs użytkownika (GUI), dlatego nadaje się do testowania oprogramowania przez człowieka. Jak wspomniano, do testów automatycznych przeznaczone są testy jednostkowe wykonywane przy pomocy biblioteki JUnit. Po uruchomieniu Testera, użytkownik dostaje takie okienko:



Na górze ekranu jest przycisk służący do zapytania Fumbo. Należy wpisać adres agenta-koordynatora (PIA), a także zapytanie RDQL. Po naciśnięciu przycisku OK, w lewej części okna dostajemy odpowiedź bezpośrednią, to znaczy taką, jaką zwraca baza danych RDF na zadane pytanie. Po chwili w prawej części ekranu pojawia się odpowiedź, którą odsyła nam koordynator. Jest to Maksymalny Zbiór Odpowiedzi (MRS).

Poniżej jest przedstawiona realizacja przykładowego zapytania do Fumbo.

Klient (u nas agent-tester) zadaje następujące pytanie do Fumbo:

```
      SELECT  ?x  WHERE  (?x    <http://chefmoz.org/rdf/elements/1.0/Cuisine>
"Chinese")
```

Pytanie jest zatem o wszystkie obiekty [gastronomiczne] z kuchnią chińską. Do dyspozycji koordynatora jest trzech agentów:

- ChineseMeansThai
- ChineseMeansAsian
- AllEnglish

Agenci ChineseMeansThai i ChineseMeansAsian kierują się zasadą `MEANING`. Zasada pierwszego z nich wygląda tak:

```
IF TRUE THEN MEANS (Chinese, Thai)
```

a zasada drugiego – tak:

```
IF TRUE THEN MEANS(Chinese, Asian)
```

Zatem nasze bazowe zapytanie jest przez nich zmieniane odpowiednio na

```
SELECT ?x WHERE (?x  <http://chefmoz.org/rdf/elements/1.0/Cuisine> "Thai")
i
SELECT ?x WHERE (?x  <http://chefmoz.org/rdf/elements/1.0/Cuisine> "Asian")
```

Trzecim agentem pracownikiem jest agent AllEnglish, którego praca jest oparta na zasadzie RDQL i warunku `COUNT`:

```
IF COUNT<=7 THEN RDQL(
SELECT ?x WHERE (?x  <http://chefmoz.org/rdf/elements/1.0/Country>  "United
Kingdom"))
```

Czyli wyciąga on z bazy informację o wszystkich restauracjach z Wielkiej Brytanii i działa tylko wtedy, jeśli dotychczasowy zbiór rozwiązań liczy nie więcej niż 7 pozycji. Klient

(Tester) ma już pewien bazowy zbiór odpowiedzi, który dla danych testowych odpowiedzi liczy 5 pozycji i wygląda następująco:

```
<rdf:RDF
    xmlns:j.0="http://chefmoz.org/rdf/elements/1.0/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.1="http://purl.org/dc/elements/1.0/" >

  <rdf:Description rdf:nodeID="A0">
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.1:Title>China Town</j.1:Title>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A1">
    <rdf:type rdf:resource="http://chefmoz.org/rdf/elements/1.0/Restaurant"/>
    <j.1:Title>The China City</j.1:Title>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.0:Cuisine>Vietnamese</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
    <j.0:Cuisine>Indian / Pakistani</j.0:Cuisine>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A2">
    <j.1:Title>Dew Drop Inn</j.1:Title>
    <j.0:Country>United Kingdom</j.0:Country>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A3">
    <j.1:Title>Meedou</j.1:Title>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

<rdf:Description rdf:nodeID="A4">
    <j.0:Cuisine>Thai</j.0:Cuisine>
    <j.1:Title>Taste Of Asia</j.1:Title>
    <j.0:Country>United Kingdom</j.0:Country>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
  </rdf:Description>
</rdf:RDF>
```

Zadanie składające się z pytania i bazowego zbioru odpowiedzi jest wysyłane do koordynatora. Koordynator rejestruje je i przesyła do agentów – pracowników. Jeżeli na liście koordynatora agenci – pracownicy są ułożeni w kolejności {ChineseMeansThai, ChineseMeansAsian, AllEnglish), odpowiedź Fumbo liczy 11 pozycji i wygląda następująco: (elementy wypisane na szaro były już w odpowiedzi bazowej, czarne są dodane przez Fumbo):

```
<rdf:RDF
   xmlns:j.0="http://chefmoz.org/rdf/elements/1.0/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:j.1="http://purl.org/dc/elements/1.0/" >

 <rdf:Description rdf:nodeID="A0">
   <j.0:Cuisine>Chinese</j.0:Cuisine>
   <j.1:Title>Dew Drop Inn</j.1:Title>
   <j.0:Country>United Kingdom</j.0:Country>
 </rdf:Description>
```

```
  <rdf:Description rdf:nodeID="A1">
    <j.0:Cuisine>Thai</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
    <j.1:Title>Mai Thai</j.1:Title>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A2">
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.1:Title>Taste Of Asia</j.1:Title>
    <j.0:Cuisine>Thai</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A3">
    <j.0:Country>United Kingdom</j.0:Country>
    <j.0:Cuisine>Indian / Pakistani</j.0:Cuisine>
    <j.1:Title>Jaflong</j.1:Title>
    <j.0:Cuisine>Asian</j.0:Cuisine>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A4">
    <j.1:Title>Sukothai</j.1:Title>
    <j.0:Cuisine>Thai</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A5">
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.1:Title>Meedou</j.1:Title>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A6">
    <j.0:Cuisine>Indian / Pakistani</j.0:Cuisine>
    <j.0:Cuisine>Vietnamese</j.0:Cuisine>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.1:Title>The China City</j.1:Title>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A7">
    <j.1:Title>Yim Thai</j.1:Title>
    <j.0:Cuisine>Thai</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A8">
    <j.0:Country>United Kingdom</j.0:Country>
    <j.1:Title>Zarin</j.1:Title>
    <j.0:Cuisine>Indian / Pakistani</j.0:Cuisine>
    <j.0:Cuisine>Asian</j.0:Cuisine>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A9">
    <j.0:Cuisine>Indonesian</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
    <j.1:Title>Java Restaurant</j.1:Title>
    <j.0:Cuisine>Asian</j.0:Cuisine>
  </rdf:Description>

  <rdf:Description rdf:nodeID="A10">
    <j.1:Title>China Town</j.1:Title>
    <j.0:Cuisine>Chinese</j.0:Cuisine>
    <j.0:Country>United Kingdom</j.0:Country>
  </rdf:Description>
</rdf:RDF>
```

Zatem przybyło 6 pozycji potencjalnie ciekawych dla użytkownika. Fumbo zwraca wszystkie dostępne informacje na temat obiektów; dla uproszczenia zapisu zrezygnowano powyżej z wyświetlania danych tutaj nieistotnych (np. położenia, parkingu, cen itp.)

Poniżej zaprezentowany jest fragment logu Fumbo, który pokazuje kolejne działania agentów.:

```
agents.coordinator :: new external task received.
agents.coordinator :: Task no. 1 sent to workers , |MRS|=5
agents.worker      :: [chineseMeansThai] received a task no. 1
agents.worker      :: [chineseMeansThai] forwarded a task no. 1 to [chineseMeansAsian],|MRS|=9
agents.worker      :: [chineseMeansAsian] received a task no. 1
agents.worker      :: [chineseMeansAsian] forwarded a task no. 1 to [ignore], |MRS|=11
agents.worker      :: [ignore] received a task no. 1
agents.worker      :: [ignore] forwarded a task no. 1 to [allEnglish], |MRS|=11
agents.worker      :: [allEnglish] received a task no. 1
agents.worker      :: [allEnglish] forwarded a task no. 1 to coordinator, |MRS|=11
agents.coordinator :: received an answer to task no. 1
agents.coordinator :: coordinator sent answer to task no. 1 back to the user
```

Jeżeli zmienimy kolejność agentów-pracowników na następującą:

1. ChineseMeansThai
2. ChineseMeansAsian
3. AllEnglish

to Fumbo zwróci odpowiedź zupełnie inną. Ponieważ agent „AllEnglish” kieruje się warunkiem `IF COUNT <=7` i jest na początku listy, do MRS zostaną dodane wszystkie restauracje w Wielkiej Brytanii. Nie będę wypisywał zwróconego zbioru odpowiedzi, dla danych testowych liczy on 90 pozycji. Wynika to z faktu, że w pierwszym przykładzie agent AllEnglish nie dodawał swoich odpowiedzi, ponieważ nie był spełniony jego warunek. W drugim przykładzie, bazowy zbiór rozwiązań zawierał tylko 5 pozycji, a więc agent AllEnglish dopytywał bazę.

Jak widać, jeżeli pracownicy kierują się zasadami opartymi na warunku `COUNT`, na MRS bardzo duży wpływ ma ich kolejność. Najważniejszą przesłanką przy ustalaniu kolejności pracowników powinna być ocena ich szczegółowości. Agenci, którzy zadają pytania ogólniejsze (które zwrócą zapewne większe zbiory rozwiązań) powinni być umieszczani raczej na końcu łańcucha. Na początku do głosu powinni dochodzić agenci w delikatny sposób zmieniający zapytanie, np. wprowadzający bliskie synonimy (zasada `MEANS`) lub ignorujące wyjątkowo ostre kryteria (zasada `IGNORE`).

Podsumowanie

Celem pracy było zbudowanie i opisanie systemu agentowego działającego w oparciu o format RDF. Do zilustrowania problemu oraz funkcjonalności, zdecydowano się na implementację konkretnego zastosowania systemu i wybrano zagadnienie wspomagania podróży. Platforma działa jako rozbudowana, semantyczna wyszukiwarka obiektów i połączeń turystycznych. Dzięki podejściu agentowemu oraz dzięki analizie danych na poziome semantycznym, daje ona wyniki bliższe intencjom użytkownika. Opierając się na danych w formacie RDF, skonstruowano oprogramowanie oceniające informacje według ich znaczenia, filtrujące je i przedstawiające użytkownikowi w jak najlepszej postaci. Dzięki używaniu RDF, system potrafi przetwarzać dane rozumiejąc je, może wyciągać wnioski, uczyć się na zachowaniach użytkownika. Wykorzystanie ontologii pozwoliło wejść na nowy poziom analizy informacji – jest to już nie filtrowanie za pomocą słów kluczowych, lecz za pomocą znaczenia.

System jest systemem agentowym, co oznacza, że cała funkcjonalność jest podzielona między niezależne, autonomiczne i wysoce samodzielne jednostki programowe – agentów. Projekt całości został podzielony na drobne części odpowiadające małym fragmentom funkcjonalności; za każdy taki odcinek odpowiadał oddzielny agent. Programy agentowe wykonują swoje zadania samodzielnie i niezależnie, ale oczywiście ze sobą się komunikują i są odpowiednio koordynowane. Zastosowanie takiego podejścia pozwoliło zaimplementować skomplikowany system realizując w praktyce sprawdzoną zasadę „Dziel i zwyciężaj" (ang. *Divide & Conquer*). Co więcej, dzięki takiemu podejściu, system jest lepiej skalowalny i powinien się lepiej zachowywać w warunkach dużego obciążenia.

Moją częścią całego systemu jest podprojekt „Fumbo". Zadaniem moich agentów jest rozszerzanie pierwotnej odpowiedzi dostarczonej przez silnik wyszukiwarki. Agenci kierują się predefiniowanymi, parsowanymi zasadami, na podstawie których „dopytują" bazę danych o dodatkowe informacje, takie o które nie pytał użytkownik wprost. Przy dobrze zbudowanych zasadach, pozwala to przekazać użytkownikowi więcej cennych informacji, niż się on spodziewał. Często pytający nie do końca nawet wie, o co powinien pytać; Dobrze skonfigurowani agenci projektu „Fumbo" potrafią niejako odgadnąć intencje pytającego i podsunąć mu takie informacje, których nie zwróci zwyczajna wyszukiwarka. Przykładowo, agenci kierują się zasadami synonimów (zadają dodatkowe zapytania używając słów

bliskoznacznych), lub zasadami obcinania warunków (w przypadku małej liczby odpowiedzi, agent zadaje pytanie powtórnie, ale z łagodniejszym warunkiem).

Praca agentów w projekcie „Fumbo" jest koordynowana przez agenta – koordynatora, który czuwa nad opracowywaniem każdej odpowiedzi, nad życiem i działaniem agentów – pracowników.

System agentowy został napisany i jest uruchamialny w środowisku Jade. Programy agentowe są napisane w języku Java. Do wygodnego i wydajnego zarządzania danymi w postaci RDF, a także do korzystania z dobrodziejstw ontologii, wykorzystywana jest zaawansowana biblioteka Jena. Całość została przetestowana przy pomocy narzędzia JUnit oraz na danych dostępnych w Internecie.

W pracy pokazano, że agentowe podejście do tworzenia systemów jest ciekawą i nowoczesną alternatywą dla klasycznej budowy aplikacji. Rozbicie funkcjonalności pomiędzy niezależne programy agentowe pozwala osiągnąć skalowalność i niezawodność. Systemy takie mogą w łatwy sposób same się uczyć, dostosowywać do nowych wymagań i nowych warunków. W przypadku implementacji agentów mobilnych jest także możliwość korzystania z systemu przy pomocy szerokiej gamy urządzeń: komputerów, palmtopów, telefonów komórkowych. Nadanie programom agentowym dużej autonomii i samodzielności pozwala zwiększyć równoległość ich pracy, wydajność oraz niezawodność systemu. Ponieważ ilość informacji dostępnych w Internecie rośnie w tempie wykładniczym, istnieje zapotrzebowanie na narzędzia, które pozwolą nam wyszukiwać i przetwarzać je coraz szybciej i coraz bardziej precyzyjnie. Na dzisiejszym globalnym rynku, tylko ciągły rozwój takich technik pozwoli nam zapanować nad morzem danych i wyprzedzić konkurencję.

Bibliografia:

1. Google, http://google.com
2. World Wide Web Consortium, www.w3.org
3. HTML, http://www.w3.org/MarkUp/
4. CSS, http://www.w3.org/Style/CSS/
5. XML, http://www.w3.org/XML/
6. XHTML, http://www.w3.org/MarkUp/
7. DTD, http://www.w3schools.com/dtd/default.asp
8. XML Schema, http://www.w3.org/XML/Schema
9. XPATH, http://www.w3.org/TR/xpath
10. XSLT, http://www.w3.org/Style/XSL/
11. Andrew Hunt, David Thomas - "Pragmatyczny Programista. Od czeladnika do mistrza", Wydawnictwa Naukowo-Techniczne, Maj 2002
12. RDF Primer, http://www.w3.org/TR/rdf-primer/
13. World Wide Web Consortium, http://www.w3.org
14. URIhttp://www.w3.org/2004/11/uri-iri-pressrelease
15. N-triples, http://www.w3.org/2001/sw/RDFCore/ntriples/
16. Dublin Core, http://dublincore.org/
17. RDQL, http://www.w3.org/Submission/RDQL/
18. RDQL tutorial, http://jena.sourceforge.net/tutorial/RDQL/
19. Ontologia, http://pl.wikipedia.org/wiki/Ontologia
20. OWL, http://www.w3.org/TR/owl-features/
21. Marcin Paprzycki – „Agenci programowi jako metodologia tworzenia oprogramowania", dostępne pod http://www.e-informatyka.pl/article/show/422
22. Russel, Norvig 1995
23. IBM, 1887
24. Wooldridge, 1997
25. Maes, 1998
26. FIPA, http://www.fipa.org/
27. ACL, http://www.fipa.org/repository/aclspecs.html
28. COM, http://www.microsoft.com/com/default.mspx

29. ActiveX, http://www.microsoft.com/com/default.mspx
30. Corba, http://www.omg.org/
31. Enterprise Java Beans, http://java.sun.com/products/ejb/
32. AgentLab, agentlab.swps.edu.pl
33. Maciej Gawiniecki – „Modelowanie użytkownika na podstawie interakcji z systemem opartym o technologie WWW"
34. Szymon Pisarek - „Ontologicznie zorientowane przeszukiwanie Internetu"
35. English-Swahili Dictionary, http://www.yale.edu/swahili/
36. Java, http://www.java.sun.com
37. Ant, http://ant.apache.org/
38. JUnit, http://www.junit.org/
39. Eclipse, http://www.eclipse.org/
40. Jade, http://jade.tilab.com/
41. Jena, http://jena.sourceforge.net
42. RDFS, http://www.w3.org/TR/rdf-schema/
43. HP Labs, http://www.hpl.hp.com/
44. Log4J, http://logging.apache.org/log4j/docs/
45. ChefMoz, http://chefmoz.org/
46. Dane RDF, http://rdfdata.org/data.html
47. Martin L. Griss – „My Agent Will Call Your Agent…But Will It Respond? - http://martin.griss.com/pubs/tr-hpl-1999-159.pdf

Oświadczenie o samodzielnym wykonaniu pracy

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przez mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami. Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni. Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

....

Data Wawrzyniec Hyska