**Raport**

Instytutu Automatyki i Informatyki Stosowanej

Politechniki Warszawskiej

# Oprogramowanie wspierające prace badawcze nad mechanizmami przydziału zasobów w zadaniach wieloagentowych dużej skali

Piotr Talipski, Mariusz Kamola

**E-mail:** P.Talipski@stud.elka.pw.edu.pl, M.Kamola@ia.pw.edu.pl

Raport nr: 10-04

Warszawa, marzec 2010



Praca naukowa finansowana ze środków na naukę w latach 2008-2010 jako projekt badawczy N N514 416934

# Spis treści

# Wstęp

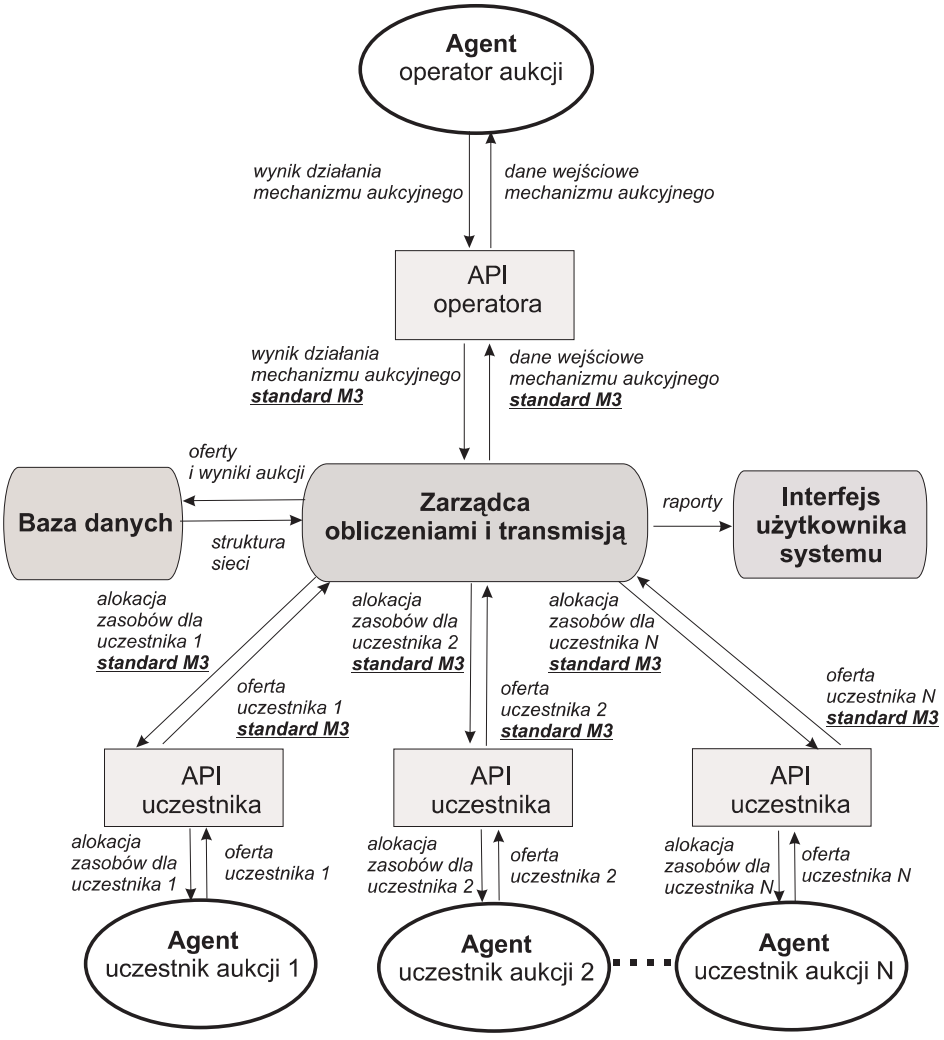
W IAiIS został opracowany model danych, $M^3$ [1], służący opisowi procesu składania ofert kupna i sprzedaży usług na rynku wielotowarowym. Został on następnie uszczegółowiony dla rynku usług przesyłowych w sieci, np. sieci teletransmisyjnej z protokołem IP. Następnie zostały opracowane mechanizmy przydziału zasobów dla różnych sytuacji na rynku, wykorzystujące model $M^3$. Jednak okazuje się, że powstanie modelu $M^3$ oraz szeregu mechanizmów nie wystarcza. Potrzebne jest narzędzie umożliwiające analizę powstałych modeli i algorytmów na podstawie danych testowych. Dopiero wielokrotne wykonanie testów i porównanie ich wyników dostarczy dostatecznych informacji na temat zachowania konkretnego modelu czy algorytmu. Dostępne narzędzia do rozwiązywania zadań optymalizacji nie ułatwiają automatyzacji testów i wymagają specyficznego formatu danych (np. model AMPL [11]). Dodatkowym utrudnieniem jest rozwój technologii przetwarzania danych i wzrost znaczenia komunikacji w środowisku rozproszonym.

Dlatego swoje uzasadnienie ma powstanie platformy, tj. oprogramowania do organizacji eksperymentów i aplikacji, wspomagającej ich tworzenie, uruchamianie i analizę. Pożądanym rezultatem jest takie zaprojektowanie architektury systemu, aby był on łatwo skalowalny, a modyfikacje nie nastręczały większych trudności. Jednak zadanie stworzenia takiej platformy jest w dużym stopniu skomplikowane i wymaga obszernej wiedzy w dziedzinie nowoczesnych metod projektowania, ale przede wszystkim dogłębnej znajomości zagadnień optymalizacji. Dlatego sensownym rozwiązaniem wydaje się podział ról i odpowiedzialności na część związaną z opracowaniem architektury rozwiązania i część poświęconej samej optymalizacji. W ramach pierwszej z nich powstała niniejsza praca. Oprócz posiadania pewnych założeń koncepcyjnych, wybór konkretnych technologii i znalezienie sposobu na ich złączenie było głównym celem przyświecającym pracom rozwojowym.

Umożliwienie organizacji i analizy eksperymentów w przyjaznym dla użytkownika interfejsie stanowi punkt wejściowy do opracowania algorytmów i strategii dla ewentualnej platformy giełdowej, gdzie zawierane byłyby rzeczywiste transakcje na rzeczywiste usługi. Docelowo platforma mogłaby stanowić punkt, w którym spotykają się operatorzy oferujący swoje usługi i klienci, którzy poprzez akceptację reguł gry rynkowej, zawierają kontrakty na konkretne usługi. Stworzenie i skomercjalizowanie takiego sytemu pozwoli na stymulację rozwoju sieci poprzez mechanizmy wolnorynkowe i regulację (przy założeniu istnienia regulatora).

Przedstawione tutaj oprogramowanie stanowi trzon takiej platformy badawczej mechanizmów

aukcyjnych. Stworzenie jego architektury podlegało szeregowi ograniczeń, a określenie funkcjonalności – szeregowi postulatów docelowych użytkowników. Ogólna architektura opracowanej platformy została przedstawiona na Rys. 1.Struktura tego dokumentu jest następująca. Rozdział 1 przedstawia szczegółowy opis modelu $M^3$. Rozdział 2 stanowi studium wykonalności projektu; jest dyskusją nt. zasadności jego wykonania i stanowi wstępną analizę dostępnych technologii. Rozdział 3 prezentuje architekturę platformy, poczynając od ogólnej jej struktury, a następnie omawiając organizację bazy danych, komunikację pomiędzy platformą a modułami użytkownika, warstwę prezentacji i samej logiki aplikacji. Podsumowanie oraz wizja możliwości zastosowania platformy w obliczeniach równoległych dla zadań złożonych obliczeniowo są zawarte w rozdziale 4.



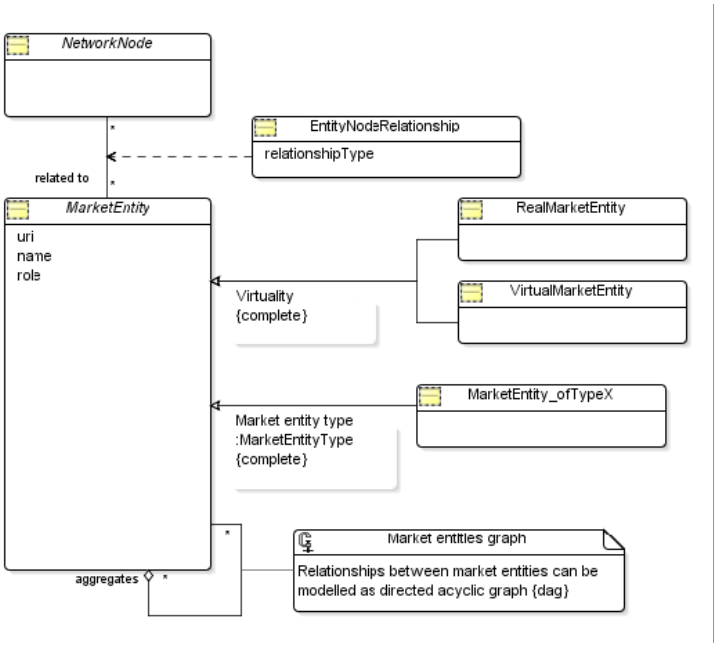

**Rysunek 1. Ogólna architektura środowiska badawczego mechanizmów aukcyjnych**

# 1. Wykorzystanie modelu M$^3$ (Multicommodity Market Data Model)

Punktem wyjściowym do opracowania platformy organizacji eksperymentów jest model M$^3$. Projekt ten dostarcza elastycznych i uniwersalnych modeli danych rynku i komunikacji. Umożliwia to integrację danych rynkowych z różnych, rozproszonych źródeł i systemów oraz zapewnia normalizację wszystkich procedur rynkowych, zarówno z punktu widzenia uczestników rynku, jak i operatora (regulatora). M$^3$ w szczególności przeznaczony jest dla rynków o strukturze wielotowarowej, gdzie wolność w handlu jest ograniczona infrastrukturą. Ponadto otwartość i uniwersalność tego modelu daje duże możliwości przy integracji zarówno w systemów czasu rzeczywistego, jak również w projektach badawczych, gdzie przedmiotem badań są nowe modele rynkowe i algorytmy. Dodatkowo, model M$^3$ może być używany do zbierania danych porównawczych (*ang. benchmark*) opracowanych eksperymentów [1].

W celu wykorzystania modelu M$^3$, w oprogramowaniu, oprócz modelu koncepcyjnego w UML, powstał zestaw schematów XSD (ang. *XML Schema Document*). Zawiera on konkretne definicje poszczególnych bytów modelu, ich parametrów, rozszerzeń i zależności pomiędzy nimi. Podstawowe struktury wykorzystywane w platformie są następujące:

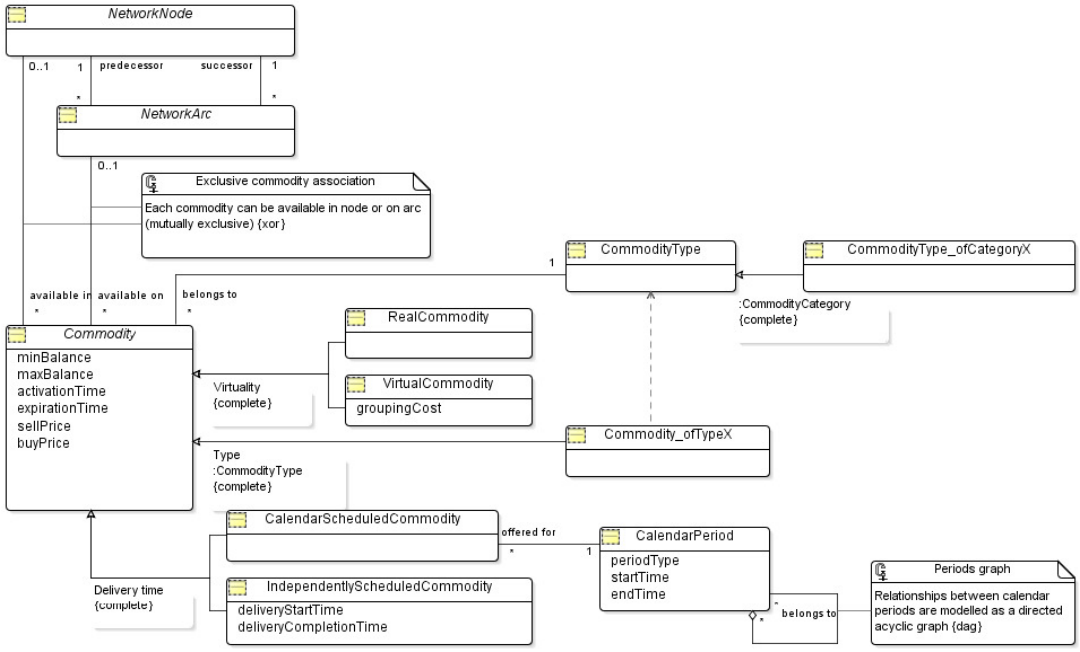
a) **Uczestnik giełdy** (*ang. market entity*) Rys. 2.




Rys. 2. Diagram zależności encji dla uczestnika giełdy.

Uczestnik giełdy jest identyfikowany przez pola URI (możliwość zamiany na adres IP), nazwę, rolę. Dodatkowo każdy uczestnik giełdy jest powiązany relacją z węzłem sieci (wiele do wielu). Ponieważ uczestnicy mogą stanowić grupę hierarchiczną, istnieje relacja agregacji do encji *MarketEntity*. Ponadto istnieje możliwość rozróżnienia rzeczywistych i wirtualnych uczestników – ci ostatni mogą tworzyć grupę posiadającą wspólną cechę, taką jak położenie geograficzne, stosowane technologie itp.

b) **Towar/usługa** (*ang. commodity*) wraz z relacją czasu (Rys. 3.).



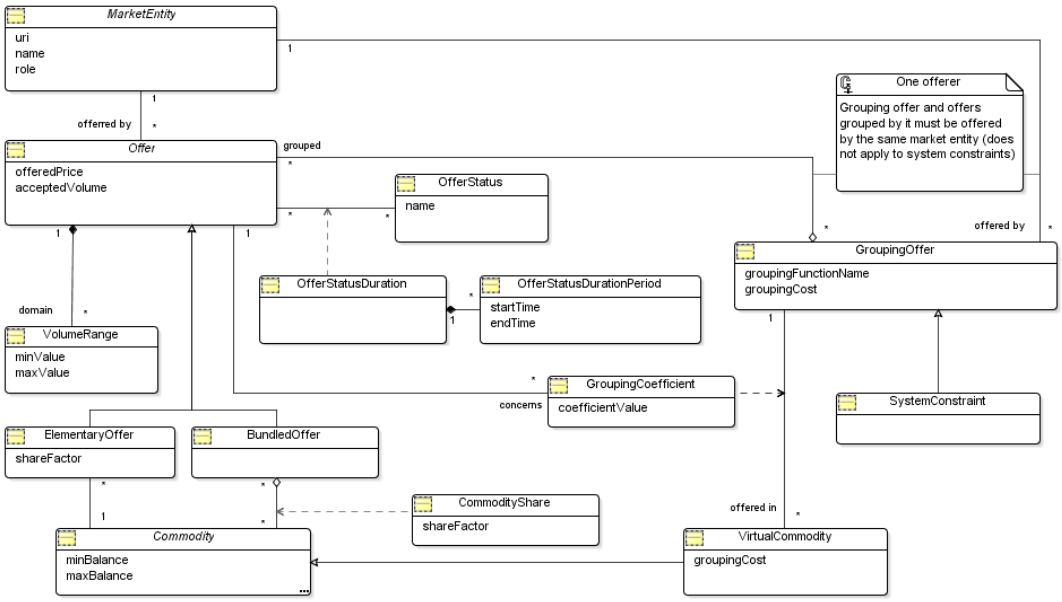

Rys. 3. Diagram zależności encji dla usługi.

Każda usługa jest ściśle powiązana relacjami z:

- kalendarzem (jeden do wiele),
- typem (jeden do wiele),
- węzłem sieci lub łukiem sieci (wiele do 0 lub 1), przy czym powiązanie jest wzajemnie wykluczające się.

Ponadto istnieje możliwość rozróżnienia pomiędzy usługą rzeczywistą i wirtualną. W tym drugim przypadku stosuje się je wraz z ofertami grupowymi w celu wyrażenia bardziej skomplikowanych zależności i ograniczeń.

c) **Oferty kupna/sprzedaży** (Rys. 4.).



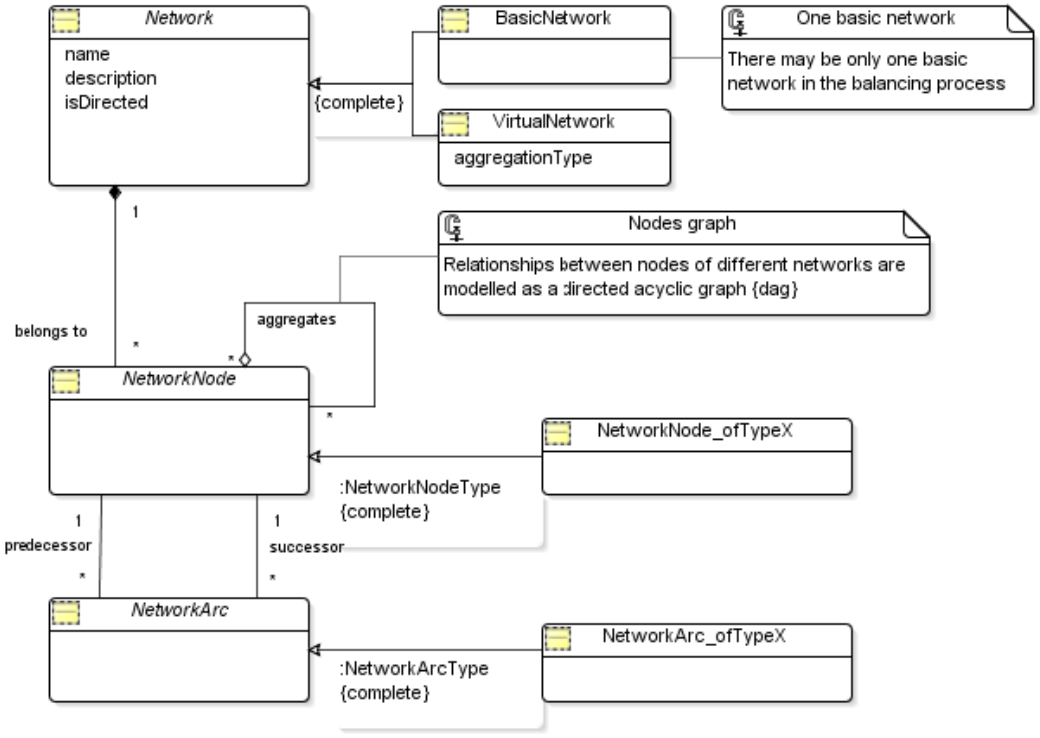

Rys. 4. Diagram zależności encji dla oferty.

Każda oferta jest połączona relacjami z:

- Uczestnikiem giełdy (wiele do jeden),
- Zakresem wartości (agregacja jeden do wiele),
- Rodzajem oferty (podstawowa lub wiązana),
- Usługą (wiele do jeden),
- Statusem (wiele do wiele),
- Znacznikiem kupno/sprzedaż (*shareFactor*),
- Ofertą grupową (poprzez klasę *GroupingCoeficient*).

Oferty zgrupowane mogą być użyte w celu wyrażenia wielu fizycznych ograniczeń – np. oferty wymagające tych samych, ograniczonych zasobów.

d) **Sieci i jej elementy** (Rys. 5.)




Rys. 5. Diagram zależności encji dla sieci.

Każdy węzeł sieci jest połączony relacjami z:

- siecią (jako całość) – wiele do jeden,
- łukiem – jeden do wiele,
- innym węzłem (jako wyrażenie agregacji w sieciach wirtualnych).

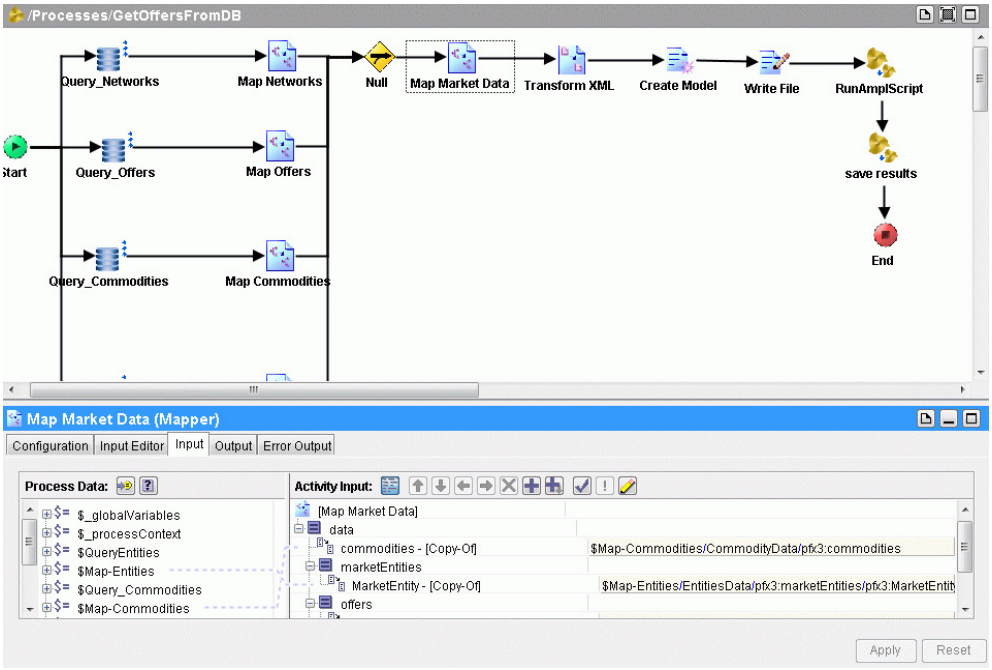
W założeniach modelu $M^3$ może istnieć tylko jedna sieć podstawowa – reprezentująca zasoby na najniższym poziomie (np. przepustowość łącza pomiędzy punktami styku międzyoperatorskiego). W przypadku sieci wirtualnych, mogą one wyrażać zależności na wyższym poziomie – np. sieć dostawcy treści i dostawcy aplikacji jako jeden element oferujący konkretną usługę.

Przestawione rysunki i ich opis mają charakter poglądowy – wprowadzający w struktury modelu. Ma to na celu zobrazowanie stopnia skomplikowania modelu danych i trudności przy implementacji platformy wykorzystującej go. Oprócz prezentowanych zależności model posiada dodatkowe, generyczne struktury typów, które umożliwiają elastyczne zdefiniowanie słownika danych. Szczegółowy opis modelu $M^3$ znajduje się w bibliografii [1].

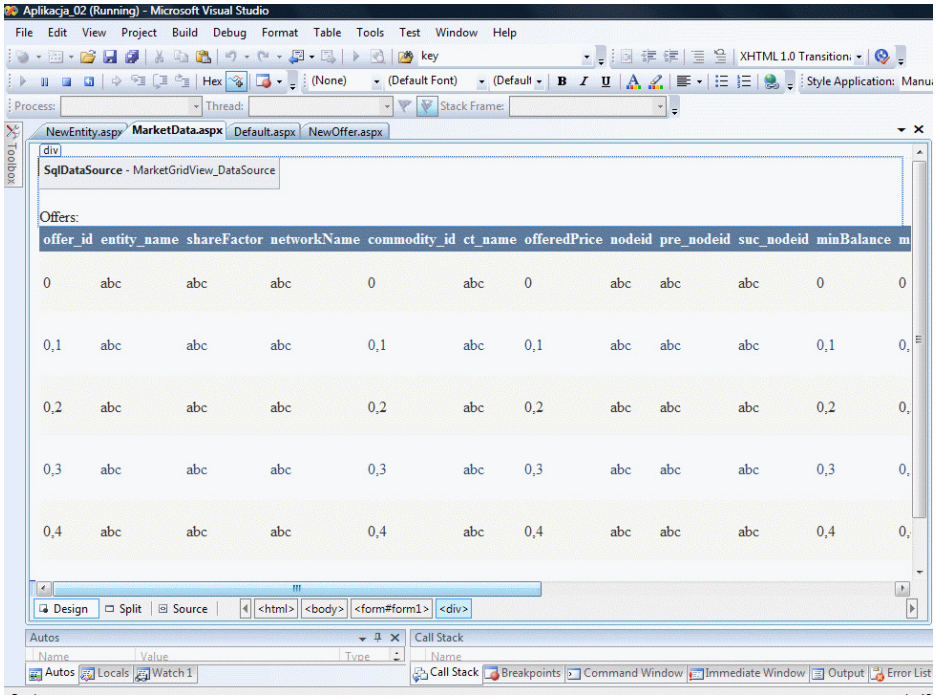
# 2. Studium wykonalności i badanie dostępnych technologii

We wstępnym etapie opracowania koncepcji zostało wykonane studium wykonywalności. Miało to na celu określenie stopnia skomplikowania oprogramowania i wyznaczenie ścieżki prowadzącej do uzyskania oczekiwanego efektu.

W pierwszej kolejności stworzono prototyp stanowiący z *ang. Proof of Concept* – dowód na to, że przedstawione wymagania mogą być zaimplementowane zgodnie z założeniami. Prototyp wykonano w technologii TIBCO® i .NET. TIBCO stanowi platformę integracyjną dla wielu systemów, dostarcza gotowych narzędzi do obsługi komunikacji w licznych protokołach i formatach danych w środowisku rozproszonym. W połączeniu z przyjaznym interfejsem użytkownika, może również stanowić narzędzie do szybkiego prototypowania. Natomiast wykorzystanie .NET miało na celu wykonanie namiastki interfejsu użytkownika do obsługi eksperymentów. Prace wykonane w tym etapie przedstawiono na Rys. 6 i 7.




Rys. 6. Prototyp interfejsu wykorzystujący dane modelu M3.




Rys. 7. Fragment makiety interfejsu użytkownika.

Wykonanie prototypu udowodniło, iż powstanie platformy w oparciu o model $M^3$ jest wykonalne i ma swoje uzasadnienie. Dalsze prace w rozwoju mogły być kontynuowane w w/w technologiach, jednak rozwiązanie to zostało odrzucone. Powodem jest przede wszystkim fakt, iż technologie te są komercyjne i niedostępne dla przeciętnego użytkownika. Zatem postanowiono wybrać rozwiązania darmowe i otwarte na modyfikacje. Alternatywnym wyborem w takim przypadku jest platforma Java. Zaletą wykorzystania $M^3$ jako modelu danych jest pewność, że informacje przenoszone za pomocą komunikatów są spójne, akceptowane i rozumiane przez wszystkie elementy składowe platformy i jej użytkowników (uczestników). Dodatkowo dostępność definicji modelu w formacie XML w postaci schematów XSD umożliwia szerokie zastosowanie bez względu na wybraną technologię. Jednak sam wybór Javy jako języka programowania jest zbyt ogólnym pojęciem. Obecnie wokół tego języka jest prowadzonych wiele alternatywnych projektów i każdy z nich ma swoje wady i zalety. Wybór jest trudny i czasochłonny – wymaga wąskiej specjalizacji w określonej technologii i strukturze programu (ang. *framework*). Często decydując się na jedno rozwiązanie można napotkać na problem, którego obejście wymaga żmudnego studiowania dokumentacji i forów użytkowników, co w rezultacie może prowadzić do decyzji i o wyborze innego podejścia. Co więcej, wybrane rozwiązanie może się okazać nie do końca kompatybilne z istniejącymi (lub przyszłymi) zależnościami. To z kolei powoduje konieczność powrotu i zrewidowania poprzednich kroków, nieraz oznaczając przebudowywanie aplikacji niemal od

nowa. W ramach pracy badawczej dokonano przeglądu i analizy dostępnych rozwiązań, biorąc pod uwagę:

- dostępność dokumentacji i pomocy technicznej,
- stabilność technologii,
- skalowalność,
- rozszerzalność
- powszechną dostępność i darmowość.

Dostępność dokumentacji i pomocy technicznej jest nieodłącznym elementem powodzenia każdego projektu. Również stabilność technologii odgrywa kluczową rolę w rozwoju oprogramowania – obecnie powstaje dużo alternatywnych rozwiązań opartych na języku programowania Java, jednak tylko część z nich przyjmuje się w środowisku i jest sukcesywnie rozwijana. Pochopne wybranie określonej technologii rodzi ryzyko, że w przyszłości aplikacja nie będzie mogła być rozwijana. Podczas wstępnych prac określono skalowalność jako ważny element projektu – dlatego zdecydowano się na implementację interfejsu użytkownika jako aplikacji internetowej. Projektowanie aplikacji z uwzględnieniem rozszerzalności daje gwarancję, że wprowadzanie zmian nie powinno być czasochłonne i powodować konieczności tworzenia całej koncepcji od nowa. Zastosowanie podejścia „luźnego powiązania" (*ang. loose coupling*) zapewnia, że aplikacje (serwisy), nawet jeżeli zostały wykonane w niekompatybilnych technologiach, mogą być łączone i tworzyć większe elementy logiczne. Zmiana w jednym komponencie może być przezroczysta dla innego komponentu, ponieważ pomiędzy nimi jest ustalony pewien zestaw interfejsów.

Po uwzględnieniu wymienionych czynników zdecydowano o wykorzystaniu następujących technologii realizacji projektu:

- JAXB (*ang. Java XML Binder*) – biblioteka do konwertowania danych w formacie XML do obiektów POJO Javy (*ang. Plain Old Java Object*) [2],
- Hibernate – framework O/RM (*ang. Object-relational mapping*) do mapowania obiektów Java na struktury bazodanowe, implementacja JPA (*ang. Java Persistance API*) [3],
- Spring – platforma aplikacji J2EE, umożliwia scentralizowaną, automatyczną konfigurację obiektów i powiązań pomiędzy nimi [4],

- GWT (*ang. Google Web Toolkit*) – schemat tworzenia dynamicznego graficznego interfejsu użytkownika stworzony przez firmę Google. Umożliwia kodowanie aplikacji na przeglądarki internetowe w sposób naśladujący implementację w klasycznym AWT [5],
- JMS (*ang. Java Message Service*) – standard określający specyfikację sposobu wymiany komunikacji komponentów J2EE [6],

Wybranie powyższych rozwiązań poprzedzone było dogłębną analizą i poddane dyskusji. Zastosowanie JAXB do generacji klas jest jedynym sensownym rozwiązaniem w obliczu stopnia skomplikowania modelu $M^3$. Ręczna implementacja zajęłaby o wiele więcej czasu, a efekty byłby dokładnie taki sam. Ograniczenie modelu $M^3$ powodowało konieczność stworzenia dodatkowych struktur przechowujących dane specyficzne dla eksperymentów. Łączenie ich w spójną całość i umożliwienie zapisu do bazy danych również jest ogromnym wyzwaniem, a przede wszystkim żmudną i narażoną na błędy pracą. Dlatego Hibernate został wybrany jako technologia automatyzująca sposób, w jaki generowany jest schemat bazy danych, jak również metody dostępu do nich. W pewnym sensie było to konsekwencją wybrania JAXB w pierwszym kroku. Wygenerowane klasy POJO trzeba było w jakiś sposób zapisać do bazy danych. Ręczne tworzenie schematu zależności encji (nawet z wykorzystaniem narzędzi graficznych) jest narażone na błędy i rodzi ryzyko, że w ich wyniku wszystkie wyższe warstwy platformy (aplikacji i prezentacji) będą musiały być do niego dopasowane. Dlatego położono szczególny nacisk na luźne powiązanie modułów i odseparowanie szczegółów implementacyjnych w każdej z warstw aplikacji.

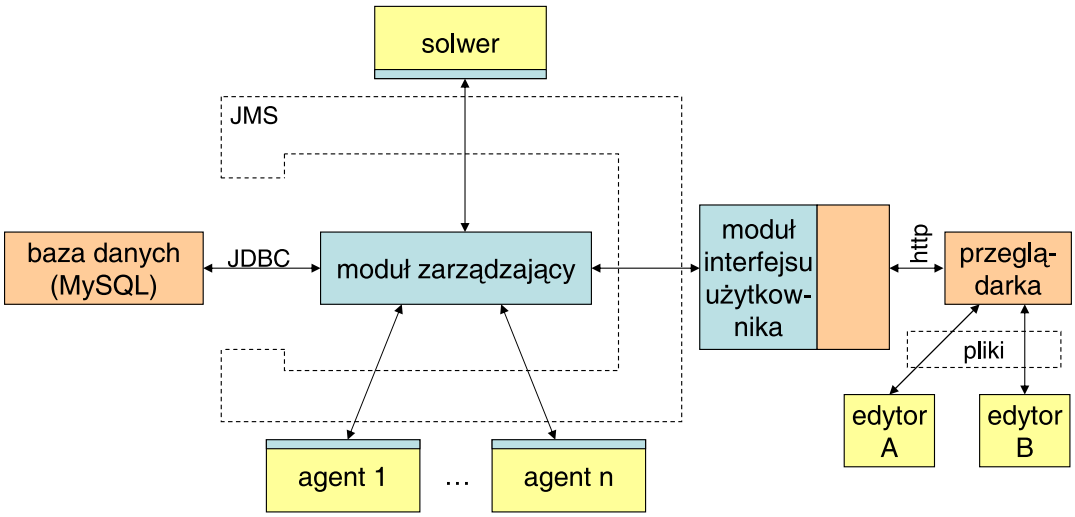
Dzięki luźnemu powiązaniu modułów poszczególne elementy aplikacji mogą być niezależnie implementowane – podstawową formą interakcji pomiędzy modułami są definiowane interfejsy. Z kolei powstanie pewnego zbioru luźno powiązanych klas i pakietów może nastręczać kolejnych problemów z integracją poszczególnych części potrzebnych do uruchomienia całości. Dlatego Spring jako platforma spajająca poszczególne elementy jest idealnym rozwiązaniem. Podczas uruchomienia aplikacji internetowej przy użyciu tego rozwiązania potrzebna jest jedynie informacja o zależnościach pomiędzy klasami – Spring automatycznie zainicjuje wszystkie klasy. Kolejnym argumentem przemawiającym za użyciem Springa jest automatyczne zarządzanie transakcjami i synchronizacja obiektów POJO (a więc w tym przypadku $M^3$ i danych eksperymentów) ze stanem bazy danych. Dzięki temu w oprogramowaniu aplikacji nie jest konieczna żadna szczególna implementacja metod synchronicznych, a zapisywanie i odczytywanie encji z bazy odbywa się nie na poziomie języka SQL, lecz na poziomie nazw klas i ich pól. Dlatego też w modelowaniu funkcjonalności aplikacji można w większym stopniu skupić się na implementacji wymagań, abstrahując od szczegółów i ograniczeń modelu relacyjnego.

Konieczność integracji powstałego kodu z systemem (lub systemami) zewnętrznymi powodowała potrzebę opracowania rozwiązania, które byłoby najlepsze pod względem prostoty użytkowania, wydajności i skalowalności. Dlatego wybór architektury JMS do wymiany komunikatów jest tutaj jak najbardziej sensowną i słuszną decyzją. Jest ona stworzona do wydajnego przetwarzania ogromnej ilości wiadomości pomiędzy użytkownikami, a do tego zapewnia pewność dostarczenia do odpowiednich odbiorców. Docelowo, w obliczu coraz bardziej popularnej architektury SOA, JMS mógłby zostać wykorzystany jako warstwa transportowa. Tak więc jest to wybór perspektywiczny, ponieważ w bardzo łatwy sposób umożliwia szybką implementację standardu *SOAP over JMS*.

Ostatnim omawianym elementem jest uzasadnienie wyboru technologii GWT jako warstwy prezentacji tworzonej aplikacji. W początkowym etapie projektowania Spring, dokładniej biblioteka *Spring Layout,* został użyty także do tworzenia graficznego interfejsu użytkownika. Jednak podstawową wadą tego rozwiązani okazała się statyczność w interakcji z użytkownikiem (standardowy wzorzec MVC), uzupełniona jedynie o elementy dynamiczne w napisanych ręcznie bibliotekach JavaScript. W toku prac GWT okazało się rozwiązaniem lepszym. Wykorzystanie tej technologii pozwala na tworzenie aplikacji internetowych przypominających standardowe aplikacje okienkowe Javy. Innowacyjność rozwiązania Google polega na tym, że stworzony kod Javy jest kompilowany do JavaScript – dynamicznego języka skryptowego przeglądarek. Dzięki temu aplikacja internetowa jest bardziej przyjazna w użytkowaniu i ewentualnych późniejszych modyfikacjach. W odróżnieniu od dotychczasowych metod tworzenia interfejsu użytkownika ze statycznymi stronami HTML, ułatwia to i znacznie przyspiesza etap tworzenia warstwy prezentacji. Architektura GWT jest stosunkowo nowa, jednak pozycja firmy Google na rynku IT daje pewność, że jej rozwiązania są przemyślane i mają duże szanse na szersze zastosowanie w rzeczywistych aplikacjach.

# 3. Architektura rozwiązania

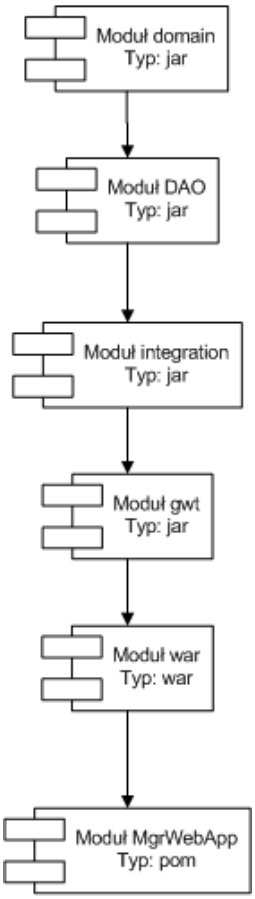
Projekt architektury komunikacyjnej platformy badawczej prezentuje rys. 8. Należy zauważyć dwie ortogonalne osie działania platformy: pionowa, aktywowana podczas wykonywania eksperymentów. Wówczas platforma jawi się modułom użytkownika (agentom i solwerowi) jedynie jako centrum dystrybucji komunikatów zawierających aktualny model M3, wzbogacone o funkcje łączenia ofert agentów oraz archiwizowania ofert w bazie danych. Natomiast oś pozioma służy analizie eksperymentów i jest aktywowana po ich zakończeniu. Wówczas główną rolę odgrywa struktura bazy danych platformy oraz walory interfejsu użytkownika.




**Rysunek 8. Architektura komunikacyjna platformy**

## *3.1. Struktura modułów*

Ze względu na stopień skomplikowania projektu, został on podzielony na kilka modułów. Każdy z nich może być kompilowany i instalowany niezależnie, co ułatwia zarządzanie rozwojem projektu jako całością (może być podzielony na kilka niezależnie rozwijanych projektów, łączonych na końcu w jedną dystrybucję). Wykorzystanie Mavena jako narzędzia do zarządzania projektem w znaczący sposób usprawniło organizację projektu. Każdy moduł posiada własną konfigurację w pliku POM, gdzie umieszczone są informacje o typie dystrybucji (archiwum) poszczególnych modułów, jego zależnościach, skonfigurowanych w zależności od potrzeb wtyczek. Struktura modułów przedstawiona jest na Rys. 9.




Rys. 9. Struktura modułów aplikacji.

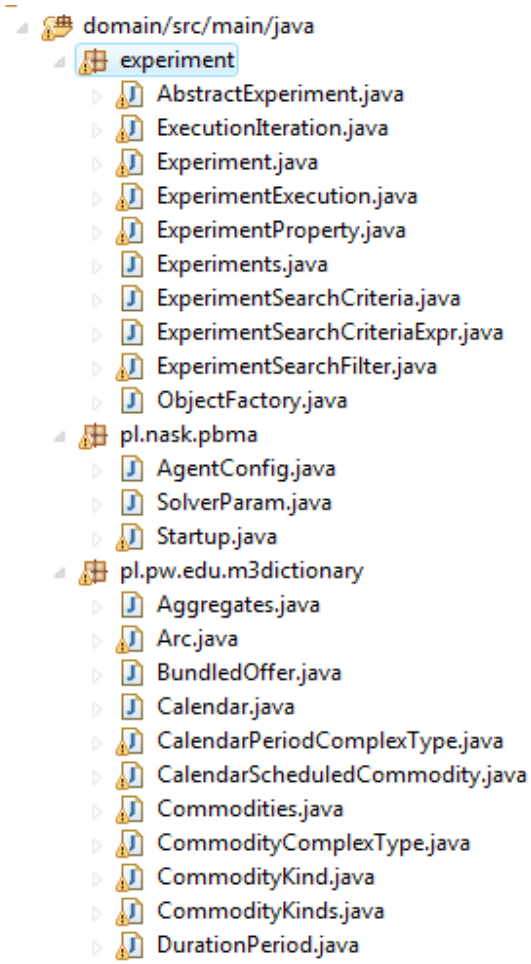
Na Rys. 9. przedstawiona jest zależność pomiędzy modułami aplikacji i rodzaj dystrybucji. Kolejność budowania zgodna jest z kierunkiem przebiegu strzałek. Ostatni moduł (*MgrWebApp*) jest tworem wirtualnym (nie jest tworzone żadne archiwum typu POM) i służy do scentralizowanego zarządzania projektem. Umieszczone są w nim informacje o zależnościach i ich wersjach na poziomie globalnym – dzięki temu w jednym miejscu utrzymywane są wartości dla wszystkich bibliotek modułów podrzędnych.

Uruchomienie procesu budowania aplikacji w ostatnim module automatycznie skompiluje wszystkie moduły zależne do odpowiednich typów. Właściwym (rzeczywistym) wynikiem zbudowanej aplikacji jest archiwum WAR (*ang. Web Application Archive*) – gotowa paczka przeznaczona do uruchomienia na serwerze J2EE.

## 3.2. Warstwa danych (Moduł domain)

Moduł *domain* stanowi zestaw podstawowych obiektów tworzących daną domenę biznesową. Z reguły model danych tworzony jest podczas zbierania wymagań i nie podlega znaczącym zmianom w cyklu

rozwoju oprogramowania – każda zmiana pociąga za sobą konsekwencje w postaci potrzeby migracji danych, co często jest zadaniem trudnym i czasochłonnym. Zaletą umieszczenia klas w oddzielnym module jest to, iż raz stworzone i zainstalowane w repozytorium archiwum może być szeroko rozpowszechnione i wykorzystywane w wielu aplikacjach. Dodatkowo, wersjonowanie archiwum daje gwarancję, iż model danych będzie wykorzystywany w sposób spójny przez wszystkich użytkowników i, w przypadku konieczności wprowadzenia modyfikacji, każda nowa wersja może być testowana i wdrażana w zsynchronizowany sposób. Struktura pakietów przedstawiona jest na Rys. 10.



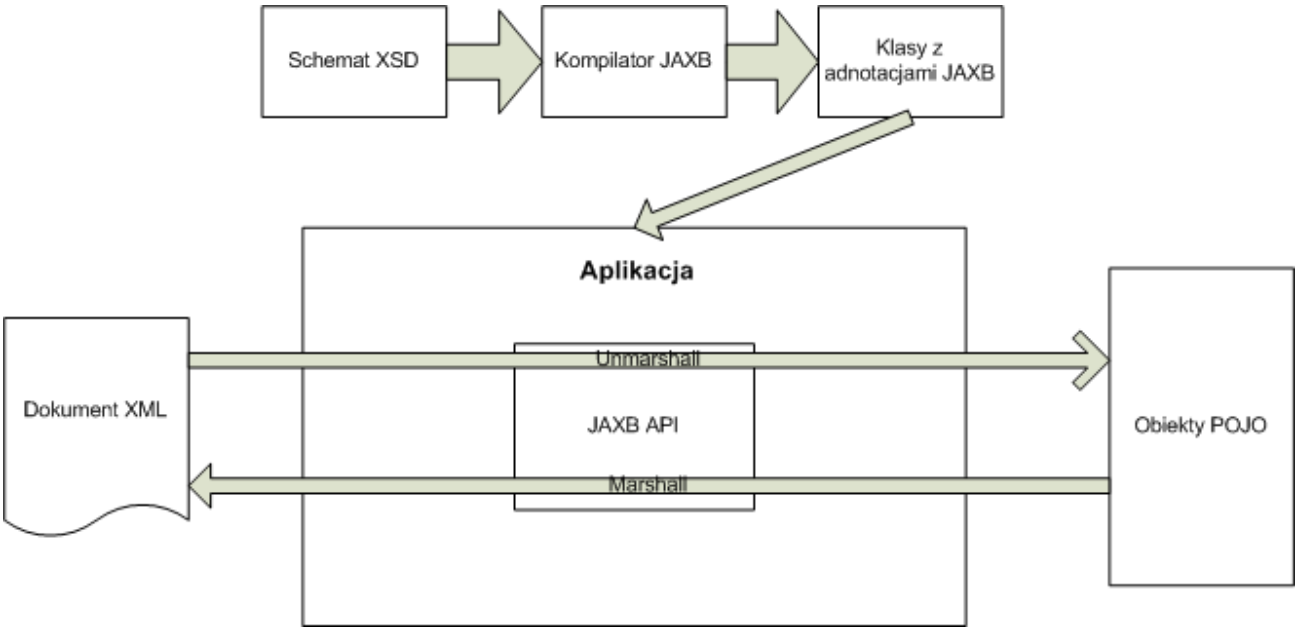

Rys. 10. Struktura pakietów modułu *domain*.

Pakiet *experiment* zawiera model danych do reprezentacji eksperymentów i ich właściwości. Eksperyment traktowany jest jako obiekt zawierający dane modelu $M^3$ wraz z dodatkowymi parametrami. Każdy eksperyment może być uruchomiony wiele razy (klasa *ExperimentExecution*), gdzie uruchomienie dodatkowo może się składać z wielu iteracji (klasa *ExecutionIteration*). Dodatkowo w pakiecie zawarte są klasy stanowiące model dla kryteriów wyszukiwania

eksperymentów – jednym z wymagań aplikacji było umożliwienie użytkownikowi wyszukiwanie eksperymentów według dynamicznie określanych kryteriów i ich zapisywanie w celu późniejszego wykorzystania jako filtr widoku.

Pakiet *pl.nask.pbma* definiują klasy dla parametrów uruchomieniowych agentów biorących udział w wymianie danych, jak również parametrów solwera. W ramach eksperymentu użytkownik posiada możliwość deklarowania w sposób dynamiczny parametrów do uruchomienia agentów i przekazywania specyficznych ustawień solwera z modułu przydziału zasobów.

Pakiet *org.openm3.m3* stanowi główną część całego modelu danych. Stanowią go klasy utworzone zgodnie z modelem $M^3$. W celu przeniesienia modelu XSD do POJO został użyty *framework* JAXB (*ang. Java Architecture for XML Binding*) – służący do wiązania danych pomiędzy ich reprezentacją w XML a POJO. Koncepcję działania tego schematu przedstawia Rys. 11.




Rys. 11. Koncepcja działania JAXB.

Punktem wejściowym dla kompilatora JAXB jest zestaw schematów XSD definiujących model danych. Na podstawie zależności pomiędzy elementami i typów poszczególnych pól, kompilator tworzy obiekty Java, które mogą być wprost wykorzystane w aplikacji. Każda z wygenerowanych klas posiada adnotacje, dzięki którym możliwa jest konwersja odwrotna – z POJO do XML. Ponieważ każdy z takich obiektów stanowi encję i jest zapisywany w bazie danych – zostały dodane również adnotacje dla JPA. Na Rys. 12. przedstawiony jest przykład dla klasy *OfferComplexType*, reprezentującej złożony typ XSD dla ofert.

```
@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "Offer_complexType", propOrder = {
    "name",
    "description",
    "offeredBy",
    "offerStatus",
    "volumeRange",
    "elementaryOffer",
    "bundledOffer"
})
@Entity
@Table(name="m3_offers")
public class OfferComplexType implements Serializable{

    private static final long serialVersionUID = -754515394104893500L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int offerId;
    @XmlElement(namespace = "http://www.openM3.org/m3")
    protected String name;
    @XmlElement(namespace = "http://www.openM3.org/m3")
    protected String description;
    @XmlElement(namespace = "http://www.openM3.org/m3")
    @OneToOne
    @Cascade(org.hibernate.annotations.CascadeType.ALL)
    protected OfferedBy offeredBy;
    @XmlElement(namespace = "http://www.openM3.org/m3")
    @OneToMany
    @Cascade(org.hibernate.annotations.CascadeType.ALL)
    @LazyCollection(LazyCollectionOption.TRUE)
    protected List<OfferStatus> offerStatus;
    @XmlElement(namespace = "http://www.openM3.org/m3")
```

Rys. 12. Przykład adnotacji JAXB i JPA dla klasy *OfferComplexType*.

Na Rys. 12. widoczne są adnotacje specyficzne dla JAXB:

- *@XmlAccessorType* – instrukcje dla JAXB o sposobie serializacji poszczególnych pól/metod danej klasy
- *@XmlType* – definicja typu elementu XSD i kolejności elementów podrzędnych
- *@XmlElement* – definicja elementu XML

Domyślne ustawienia kompilatora JAXB ustawiają adnotację *@XmlAccessorType* na wartość *XmlAccessType.FIELD*. Oznacza to, że wszystkie pola danej klasy zostają poddane serializacji i umieszczone przy marshallingu do XML. W niniejszym projekcie opcja ta została zmieniona na wartość *XmlAccessType.NONE*, a pola definiowane przez schemat XSD zostały oznaczone odpowiednimi adnotacjami. Zabieg ten ma na celu zapobiegnięcie sytuacji, w której klasa rozszerzona o dodatkowe dane przestanie być zgodna z przyjętym schematem. Widać to także na Rys. 11. Klasa *OfferComplexType* jest

encją i stanowi obiekt zapisywany do bazy danych. Dlatego obiekt POJO został rozszerzony o dodatkowe pole *offerId*, które stanowi klucz główny tabeli. Pozostałe adnotacje JPA dla tego obiekt, to:

- @Entity – adnotacja informująca JPA o tym, że klasa jest encją (obowiązkowe)
- @Table – specyfikacja nazwy tabeli w bazie danych (opcjonalne)
- @GeneratedVaue – sposób w jaki generowany jest klucz główny (obowiązkowe)
- @OneToOne lub @OneToMany – definicja zależności pomiędzy encjami (obowiązkowe)
- @LazyCollection – opcja konfigurująca późne (*ang.* lazy) dołączanie kolekcji (opcjonalne)

Dzięki wykorzystaniu mechanizmu JPA projektowanie baz danych może zostać ściśle sprzężone z tworzeniem aplikacji. Tak więc nie ma potrzeby oddzielnego projektowania schematu ERD, a następnie integracji istniejących struktur relacyjnych z zależnościami pomiędzy obiektami – wszystko dzieje się w sposób równoległy. Adnotacje JPA umożliwiają stworzenie schematu bazy danych w dowolnej implementacji – w oddzielnym pliku konfiguracyjnym umieszcza się następujące informacje:

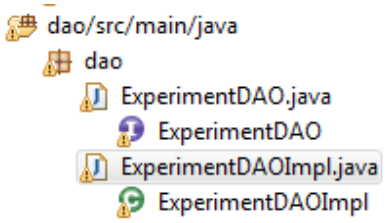
- sposób podłączenia do bazy (URL, nazwa użytkownika i hasło),
- docelowego zestawu poleceń SQL (tzw. dialekt) – pomimo istniejącego standardu SQL92, każdy dostawca baz danych posiada własną implementację pewnych elementów. Jako przykład można podać sposób generowania klucza głównego: sekwencje i wyzwalacze w bazach Oracle, w odróżnieniu od opcji *AUTO_INCREMENT* dla kolumny w MySQL,
- klasy wchodzące w skład schematu ER. W niniejszej pracy został wykorzystany Hibernate, więc konfiguracja JPA znajduje się w pliku *hibernate.cfg.xml* – umieszczony jest on w module WAR.

## 3.3. Warstwa aplikacji

### 3.3.1. Moduł dao

Moduł *dao* stanowi warstwę dostępu do obiektu danych (czyli w przypadku niniejszej pracy modelu zdefiniowanego w module *domain*). Zasadniczą cechą tego modułu jest luźne powiązanie – dostępne opcje dostępu do danych zdefiniowane są przez zestaw interfejsów i metod. Dzięki temu dla danego obiektu DAO istnieje interfejs stanowiący standard, natomiast konkretne implementacje mogą różnić się w zależności od środowiska lub platformy. W przypadku niniejszej pracy w aplikacji internetowej obiekty DAO są zarządzane przez Spring z wykorzystaniem wzorca projektowego Singleton. Tylko

jeden obiekt jest tworzony dla wszystkich żądań dostępu do danych – zatem obiekt ten jest bezstanowy. Na całość modułu składa się jeden pakiet *dao*, przedstawiony na Rys. 13.

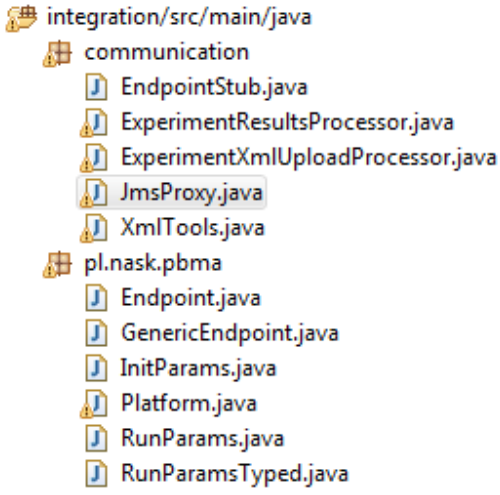



Rys. 13. Pakiet *dao*.

W ramy pakietu wchodzi interfejs *ExperimentDAO*, definiujący dostępne metody, i konkretna implementacja *ExperimentDAOImpl*, stanowiąca rzeczywisty obiekt dostępny przy żądaniu dostępu do danych.

### 3.3.2. Moduł integration

Zasadniczą rolą modułu *integration* jest zapewnienie możliwości wymiany danych powstałej aplikacji internetowej z modułem (modułami) przydziału zasobów. Po stronie aplikacji zapewnione są mechanizmy tworzenia, uruchamiania i wyszukiwania eksperymentów. Następnie uruchamiane eksperymenty trafiają do systemu zewnętrznego, gdzie są przetwarzane i zwracane ich wyniki, które z powrotem są zapisywane w bazie danych i udostępnione do dalszej analizy. Strukturę pakietów modułu przestawia Rys. 14.




Rys. 14. Moduł *integration*.

Pakiet *communication* zawiera klasy odnoszące się do implementacji części komunikacyjnej z systemem zewnętrznym. Jako warstwę transportową dla komunikacji wybrano standard JMS (*ang. Java*

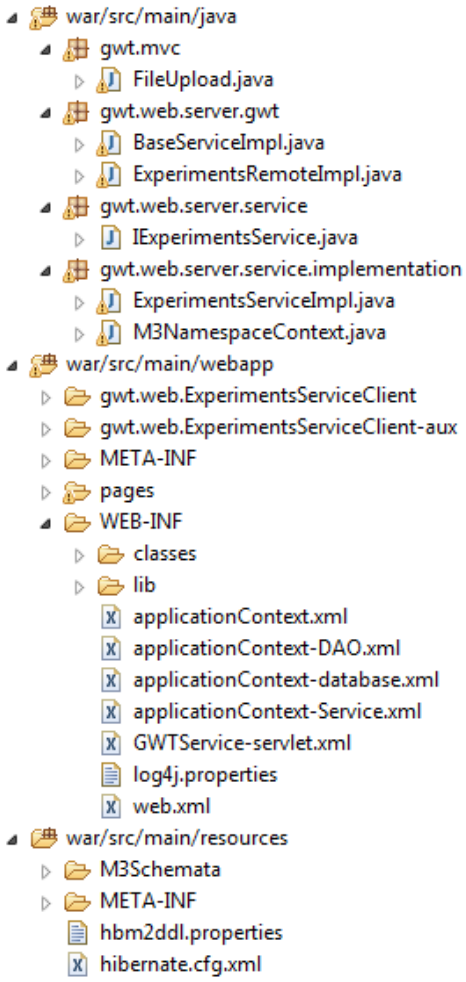
*Message Service*). Format wymiany komunikatów został uzgodniony jako w pełni zgodny z modelem $M^3$, wraz z dodatkowymi parametrami charakterystycznymi dla eksperymentu (lub jego uruchomienia).

Pakiet *pl.nask.pbma* zawiera klasy niezbędne do uruchomienia logiki mechanizmu przydziału zasobów. Do zadań tych należą m.in. uruchomienie agenta z odpowiednimi parametrami, obsługa zdarzeń wychodzących i przychodzących. Agenci uruchamiani są na podstawie danych zdefiniowanych przez użytkownika w interfejsie graficznym (moduł *gwt*). Natomiast solwer jest startowany jako oddzielny proces Javy wraz ze specyficznymi parametrami. Interfejsem tworzącym szablon dla tych bytów jest *Endpoint*, który definiuje metody *onInit()* i *onAllocation()* – odpowiednio dla inicjalizacji agenta (solwera) i dla zdarzenia alokacji zasobów dla danego uruchomienia eksperymentu (lub jego iteracji). Punkt styku z serwerem JMS stanowi klasa *JmsProxy*, która odpowiada za implementacje komunikacji pomiędzy agentami i solwerami. Dodatkowo, poprzez zależność definiowaną w konfiguracji Springa, implementacja tej klasy ma dostęp do interfejsów definiowanych w module *dao*, co pozwala na zapisywanie poszczególnych encji do bazy danych.

Elastyczność standardu JMS zapewnia możliwość wyboru dowolnej formy przesyłania wiadomości – może to być strumień znaków, strumień bajtów lub XML. Punktem styku niniejszej aplikacji i systemów zewnętrznych są w ogólności „punkty przeznaczenia" (ang. *destination*), z podziałem na kolejki lub tematy (*topics*). W terminologii JMS obiekt wysyłający wiadomość jest producentem, natomiast instancja odbierająca komunikat jest konsumentem. W przypadku kolejek zachodzi przypadek komunikacji wiele-jeden, czyli wiadomość wysyłana przez jednego (lub wielu) producentów jest odbierana przez jednego konsumenta – po pobraniu jest ona usuwana z kolejki. Natomiast w przypadku topików, zachodzi model komunikacji wiele-wiele. Tematy pełnią rolę tablicy ogłoszeń, na którą producenci mogą wysyłać komunikaty, które otrzymają wszyscy konsumenci zainteresowani danym tematem. Zastosowanie standardu JMS jako bardzo popularnego rozwiązania architektury MOM (*ang. Message-Oriented Middleware*) w znaczący sposób usprawnia komunikację w środowisku rozproszonym, przy istnieniu wielu heterogenicznych platform.

### 3.3.3. Moduł war

Moduł *war* jest końcowym rezultatem po zbudowaniu aplikacji. Jest to spakowane archiwum gotowe do umieszczenia i uruchomienia na serwerze J2EE. Cechą charakterystyczną tego modułu jest to, iż zawiera on wszystkie moduły podrzędne (archiwa *jar*), jak również elementy niezbędne do prezentacji (strony internetowe, skrypty *JavaScript*, pliki graficzne). Strukturę pakietów i katalogów modułu przestawia Rys. 15.

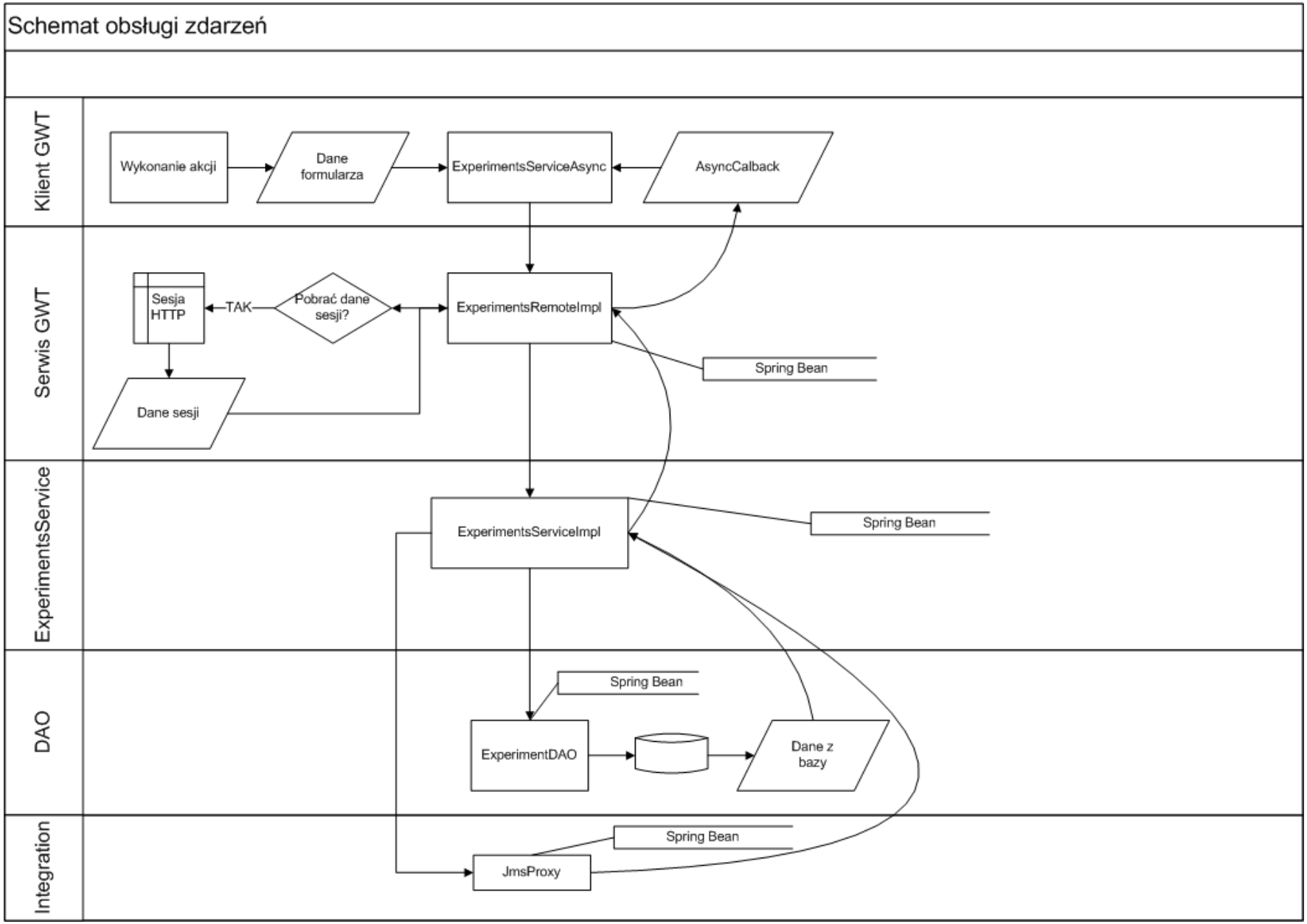



Rys. 15. Struktura modułu *war*.

Pakiet *gwt.mvc* zawiera klasę *FileUpload* – jest to prosty serwlet (dziedziczy klasę *javax.servlet.http.HttpServlet*) służący do wczytania pliku z przeglądarki do formularza strony. Jednym z wymagań stawianych aplikacji jest udostępnienie użytkownikowi możliwości wczytywania plików XML ze definicjami modelu $M^3$ z poziomu przeglądarki. W odróżnieniu od mechanizmu GWT RPC (stosowanego w pozostałych przypadkach i omówionego w rozdziale dotyczącym modułu *gwt*), w przypadku pobierania do przeglądarki pliku z lokalizacji dysku twardego jest obostrzona szeregiem ograniczeń. Ma to na celu zabezpieczenie przed nieuprawnionym dostępem do zasobów dyskowych na stronach internetowych. Dlatego jedynym możliwym sposobem wysłania pliku jest zastosowanie serwletu i przeciążenie metody *doPost()*, gdzie plik odbierany po stronie serwera znajduje się w argumencie *HttpServletRequest*. Po przetworzeniu obiektu do postaci *String* jest on zwracany w obiekcie *HttpServletResponse*, skąd następnie trafia do formularza strony.

Pakiet *gwt.web.server.gwt* zawiera klasy związane z serwisem odpowiadającym na żądania klienta GWT. Stanowi on pierwszy punkt w procesie komunikacji klient GWT – serwer. W ramach tego pakietu wchodzą dwie klasy: *BaseServiceImpl* i *ExperimentsRemoteImpl*. *BaseServiceImpl* stanowi szablon serwisu komunikującego się z klientem GWT. Klasa rozszerza serwlet GWT (*RemoteServiceServlet*) i implementuje interfejsy Springa: *Controller* i *ServletContextAware*. Zadaniem klasy *RemoteServiceServlet* jest automatyczna deserializacja przychodzących żądań od klienta i serializacja wychodzących odpowiedzi. Rolą zaimplementowanych interfejsów Springa jest zdecydowanie, która metoda obsługuje przychodzące zapytanie (*Controller*). Drugi interfejs umożliwia dostępu do obiektu *HttpSession*, w którym trzymane są obiekty związane z aktualnym stanem danych formularzy/widoków przeglądarki. Interfejs *ServletContextAware* sygnalizuje Springowi, iż dana klasa wymaga podania kontekstu serwletu, w którym jest uruchamiana (z reguły określana na podstawie kontekstu aplikacji). Zasadniczo sprowadza się to do wstrzyknięcia zależności do instancji klasy *javax.servlet.ServletContext*.

Klasa *ExperimentsRemoteImpl* stanowi właściwą klasę, do której trafia żądanie wysyłane przez klienta GWT. Dziedziczy cechy klasy *BaseServiceImpl*, jak również implementuje metody zdefiniowane w interfejsie klienckim (*gwt.web.client.ExperimentsService*). Omawiana klasa zawiera referencję do obiektu *IExperimentsService*, w pakiecie *gwt.web.server.service*. Jest to interfejs definiujący metody dla serwisu eksperymentów, który odpowiada za obsługę zdarzeń związanych z eksperymentami po stronie serwera.

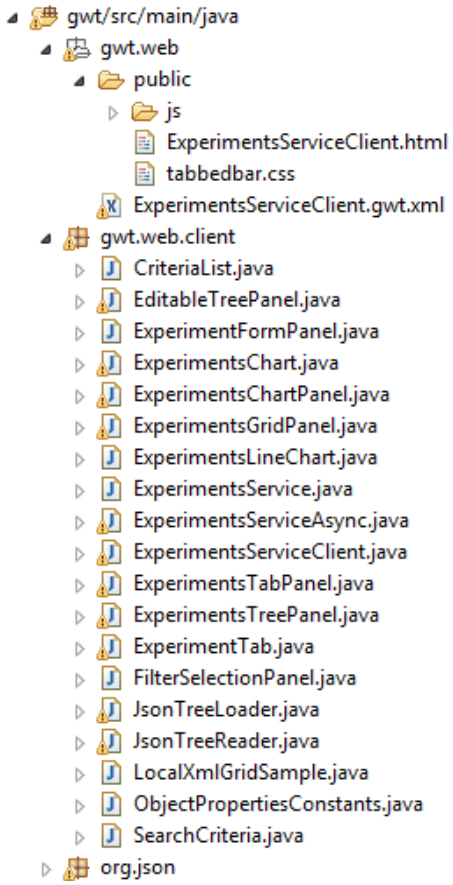


Rys. 16. Schemat obsługi zdarzeń aplikacji.

Na Rys. 16. przedstawiono poglądowy schemat obsługi zdarzeń aplikacji. Żądanie klienta jest kolejno przetwarzane przez poszczególne serwisy. W pierwszym etapie procesowania żądania uwidoczniony jest dostęp do danych zapisanych w sesji HTTP. Dostęp do obiektu sesji osiągalny jest z poziomu wątku żądania GWT poprzez dziedziczoną metodę *getThreadLocalRequest().getSession()* klasy *ExperimentsRemoteImpl*. W następnym etapie przetwarzania bierze udział instancja klasy *ExperimentsServiceImpl*, która w zależności od wywoływanej metody korzysta z metod modułu interfejsu *ExperimentDAO* i/lub *JmsProxy*. Instancje klasy inicjowane przez Springa oznaczone są komentarzem *Spring Bean*. Dzięki zastosowanemu podejściu w projektowaniu aplikacji osiągnięto luźne powiązanie modułów – każdy z nich komunikuje się z zależnymi klasami poprzez zestaw zdefiniowanych interfejsów. Zadaniem Springa jest zainicjowanie konkretnej implementacji interfejsu i wstrzykiwanie zależności do odpowiednich klas. Zaletą tego takiego rozwiązania jest niezależność w implementacji konkretnych części składowych aplikacji – w dowolnym momencie poszczególne serwisy mogą być zastąpione przez nowe, bez konieczności ingerencji w istniejące moduły dostępu do danych lub moduł kliencki.

## 3.4. Warstwa prezentacji (moduł gwt)

Moduł *gwt* stanowi ostatnią omawianą warstwę aplikacji – prezentacji. Zgodnie z wzorcem projektowym MVC warstwa prezentacji stanowi widok prezentowany w oknie przeglądarki. Jednak wraz z burzliwym rozwojem aplikacji internetowych różnica pomiędzy tzw. cienkim klientem, którego rola ogranicza się do wyświetlania danych przesyłanych przez serwer, a grubym klientem staje się coraz mniej zauważalna. Jest to konsekwencją dążenia do projektowania multimedialnych i dynamicznych aplikacji internetowych z intensywnym wykorzystaniem języka *JavaScript* i wewnętrznych silników przeglądarek. Jednakże istnienie różnic w sposobie, w jakim dokument HTML jest reprezentowany w wiodących przeglądarkach rodzi kolejne problemy w postaci potrzeby tworzenia kodu *JavaScript* w kilku wersjach. Dlatego pojawienie się schematu *Google Web Toolkit* jest przełomem w rozwoju aplikacji internetowych. Tworzenie dynamicznej aplikacji zostało w GWT sprowadzone do oprogramowania interfejsu użytkownika w języku Java, który następnie jest kompilowany na wykonywalny kod *JavaScript* – jest on zoptymalizowany względem wydajności i kompatybilności. W niniejszej pracy całość została umieszczona w oddzielnym module, którego struktura przedstawiona jest na Rys. 17.




Rys. 17. Struktura modułu *gwt*.

Na Rys. 17. widoczne są pakiety modułu *gwt*. W pakiecie *gwt.web* znajduje się plik konfiguracyjny kompilatora GWT (*ExperimentsServiceClient.gwt.xml*). Znajdują się w nim informacje dotyczące klasy startującej moduł kliencki – wymagana jest implementacja interfejsu *EntryPoint* i metody *onModule-Load()*. W pliku konfiguracyjnym znajduje się również lista modułów zależnych, skryptów *JavaScript*, stylów CSS. Wszystkie zasoby potrzebne do uruchomienia strony znajduje się w folderze *public* omawianego pakietu. Znajduje się tutaj strona *ExperimentsServiceClient.html*, w której znajduje się odniesienie do skryptu *JavaScript*, będącego rezultatem kompilacji kodu Java.

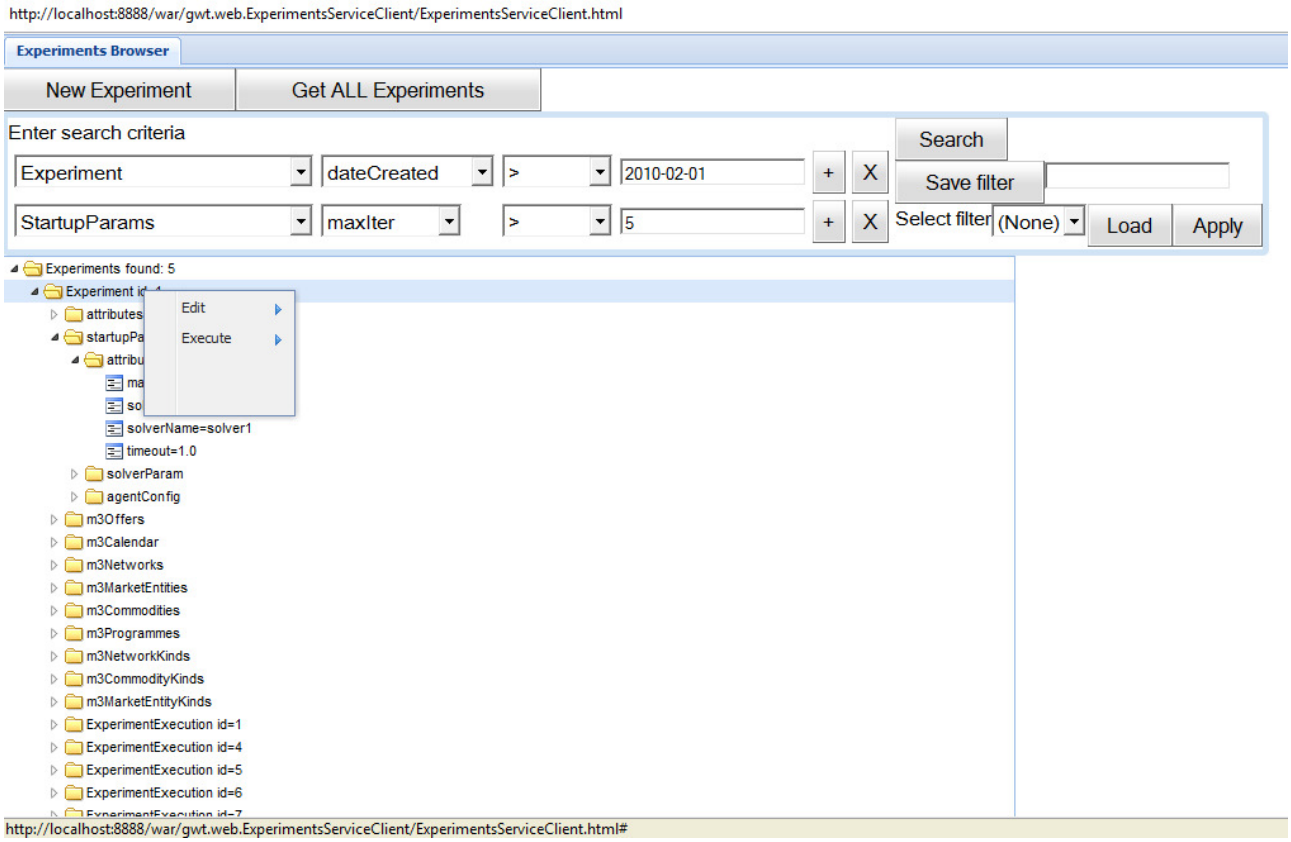
W pakiecie *gwt.web.client* znajdują się klasy tworzące logikę klienta. Klasy te zawierają elementy interfejsu graficznego użytkownika na wzór komponentów z AWT – są to panele, przyciski, szablony wyglądu (*ang. layout*) oraz tzw. widgety (specjalizowane komponenty posiadające powiązanie z pewnymi strukturami danych). Dokładniejszy opis poszczególnych klas można znaleźć w załączonej dokumentacji technicznej.

Pakiet *org.json* zawiera implementację struktur JSON (*ang. JavaScript Object Notation*) [7]. Jest on prostym formatem wymiany danych. Zapis i odczyt danych w tym formacie jest łatwy do opa-

nowania przez ludzi. Jednocześnie, z łatwością odczytują go i generują komputery. Jego definicja opiera się o podzbiór języka programowania JavaScript, Standard ECMA-262 3rd Edition - December 1999. JSON jest formatem tekstowym, całkowicie niezależnym od języków programowania, ale używa konwencji, które są znane programistom korzystającym z języków z rodziny C, w tym C++, C#, Java, JavaScript, Perl, Python i wielu innych. Właściwości te czynią JSON idealnym językiem wymiany danych.

### 3.4.1. Ekrany aplikacji klienckiej

Na Rys. 18. przedstawiono główny ekran aplikacji.




Rys. 18. Główny ekran aplikacji.

W głównym ekranie znajdują się elementy składające się na najważniejszą funkcjonalność modułu klienckiego:

a) Przycisk do tworzenia nowego eksperymentu – wciśnięcie lewego przycisku myszy na „New Experiment” otworzy nową zakładkę z polami formularzy do wypełnienia.

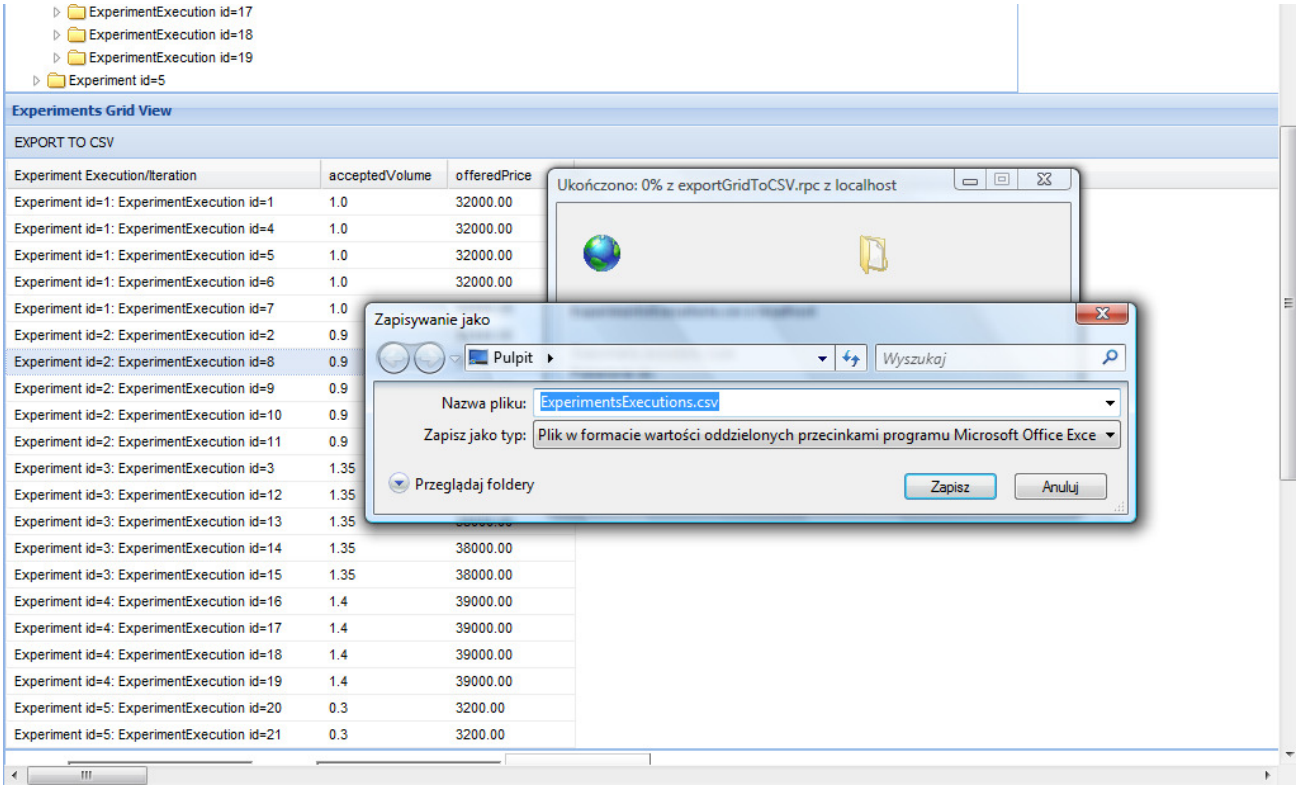
b) Przycisk do pobierania całego zbioru eksperymentów – wciśnięcie lewego przycisku myszy na „Get All Experiments”spowoduje wysłanie do bazy danych zapytania o wszystkie eksperymenty. W początkowej fazie użytkowania aplikacji może być pożądana taka możliwość.

c) Wyszukiwarka eksperymentów na podstawie dynamicznie określanych kryteriów. Zaimplementowano tutaj dynamiczną tabelę (*ang. FlexTable*) ze struktur GWT. Każdy wiersz został podzielony na sześć kolumn składających się na funkcjonalność wyszukiwarki. W pierwszej kolumnie użytkownik wybiera interesujący go obiekt (równoważny encji w bazie danych) – może to być obiekt z modelu $M^3$, jak również element charakterystyczny dla aplikacji (np. Eksperyment). W następnym kroku należy wybrać pole przypisane do wybranego wcześniej obiektu. Wybór odbywa się z list rozwijanych, stworzonych na podstawie modelu danych modułu *domain*. Listy obiektów i jego pól są wzajemnie zależne, to znaczy wybranie konkretnego obiektu z pierwszej listy zmienia dynamicznie dostępną listę opcji w liście drugiej. Dzięki temu ograniczono w znaczny sposób możliwość popełnienia błędu przy konstrukcji zapytania do bazy danych, jak również stanowi to pewnego rodzaju ułatwienie – dostępna jest pełna lista dostępnych opcji. Po wybraniu obiekt i jego właściwości następnym krokiem jest wybór wyrażenia logicznego (również z listy dostępnych opcji). Na końcu użytkownik ma możliwość wprowadzenia wartości warunku dla wybranego kryterium. Ostatnie kolumny dynamicznej tabeli służą do dodawania i usuwania wierszy konstruowanego zapytania

d) Możliwość zapisywania kryteriów wyszukiwania do bazy danych. W celu ułatwienia zarządzaniem eksperymentami zaimplementowano funkcjonalność do przechowywania warunków selekcji w celu późniejszego odczytu. W bazie danych stworzono odpowiednie struktury dla filtru widoku – podstawowym identyfikatorem jest nazwa filtru, która powinna być unikalna. Po wybraniu interesującego zbioru kryteriów należy wpisać nazwę do pola tekstowego obok przycisku „Save filter”. Wciśnięcie lewego przycisku myszy na wspomnianym elemencie zapisze warunki selekcji do bazy danych. Odczytanie i uruchamianie zapytań możliwe jest za pomocą dwóch następnych przycisków: „Load”i „Apply”. Pierwszy z nich uruchamia inną logikę, w zależności od wartości listy rozwijanej. W przypadku wartości „(None)” zostaną wyszukane wszystkie filtry z bazy danych i umieszczone w liście rozwijanej. Następnie wybranie nazwy widoku i ponowne wciśnięcie spowoduje wczytanie warunków filtru do dynamicznej tabeli kryteriów. Natomiast przycisk „Apply” służy do szybkiego wybrania nazwy konkretnego filtru z listy rozwijanej i uruchomienie wyszukiwania na podstawie jego zawartości.

e) Drzewo prezentujące strukturę eksperymentów. Tworzone jest dynamicznie na podstawie utworzonego w opisanej wcześniej wyszukiwarce zapytania (lub pobierania całego zbioru). Cechą

charakterystyczną w interakcji z użytkownikiem jest tzn. asynchroniczne doczytywanie danych. Cała struktura drzewa umieszczona jest po stronie serwera w postaci drzewa DOM, natomiast w aplikacji klienckiej uwidocznione są tylko elementy podrzędne pierwszego poziomu (ang. *children*). Wciśnięcie lewego przycisku myszy na węzeł prezentowanego drzewa spowoduje wysłanie żądania do serwera o kolejne elementy podrzędne wybranego elementu. Dane przesyłane do części klienckiej są w formacie JSON, dzięki czemu mogą być wprost zaprezentowane w strukturze drzewiastej. Dodatkową cechą omawianego komponentu jest obecność menu kontekstowego. W zależności od zawartości prezentowanego węzła tworzone jest dynamiczne menu. Na Rys. 23. została uwidoczniona sytuacja dla elementu zawierającego ciąg znaków „Experiment"- dostępne opcje to edycja i uruchomienie. Wciśnięcie lewego przycisku myszy na edycję spowoduje otwarcie eksperymentu w nowej zakładce i wczytanie do formularzy jego danych. Natomiast wybranie uruchomienia zainicjuje ciąg zdarzeń związanych z przesyłaniem danych do modułu przydziału zasobów. Bardziej szczegółowo zostało to opisane w module *integration*. Kolejnym elementem, na którym wyświetlane jest menu kontekstowe, są dane przeznaczone do analizy lub eksportu w widoku tabelarycznym oraz możliwość generowani wykresów.

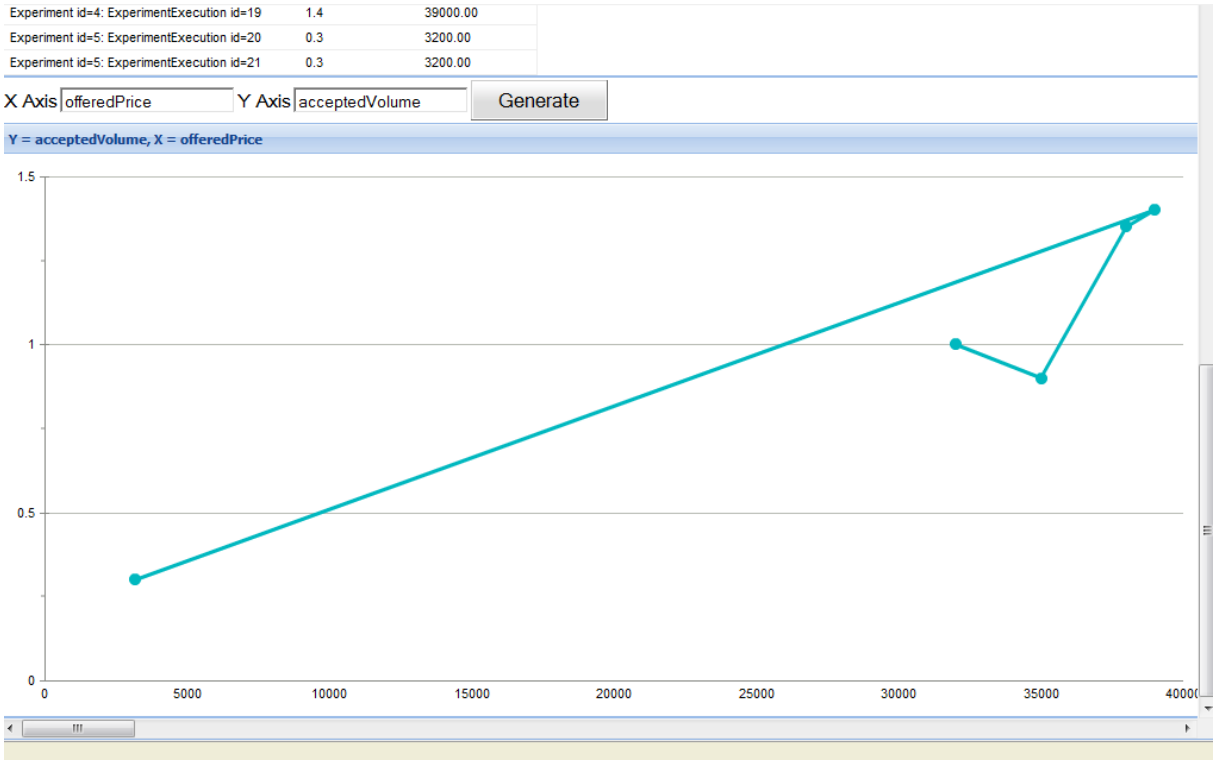



Rys. 19. Eksport danych tabeli do pliku CSV.

Na Rys. 19. widoczny jest tabelaryczny układ danych wybranych z drzewa do wyświetlenia. Widoczne jest również okno modalne przeglądarki z opcją zapisywania pliku. W momencie tworzenia tabeli w aplikacji klienckiej do nagłówka dodawany jest przycisk *EXPORT TO CSV*, który umożliwia zapisanie danych eksperymentów do pliku w formacie CSV. Ze względu na różnice w sposobie tworzenia plików przez różne przeglądarki, zdecydowano się na implementację tego kroku po stronie serwera. Tak więc ze strony klienta wysyłane jest żądanie HTTP *Get* do serwera, który na podstawie dokumentu DOM utrzymywanego w sesji użytkownika przetwarza dane do formatu CSV i wysyła odpowiedź. W odpowiedzi ustawiane są odpowiednie nagłówki HTTP: *content-type* (jako *text/csv*) oraz *content-disposition* (jako *attachment*). Pozwala to na rozpoznanie przez przeglądarkę zwracanej odpowiedzi jako załącznika i wyświetlenie okna z wyborem lokalizacji dla odbieranego pliku.

Należy również wspomnieć o tym, że istnieją różnice w specyfikacji formatu w programie Microsoft Excel – w programie tym wartości poszczególnych pól oddzielane są średnikami, podczas gdy odmienny format definiuje standard RFC 4810 [9]. W implementacji zastosowano się do wspomnianego standardu – daje to pewność, że większość aplikacji będzie w stanie prawidłowo odczytać wyeksportowane dane.

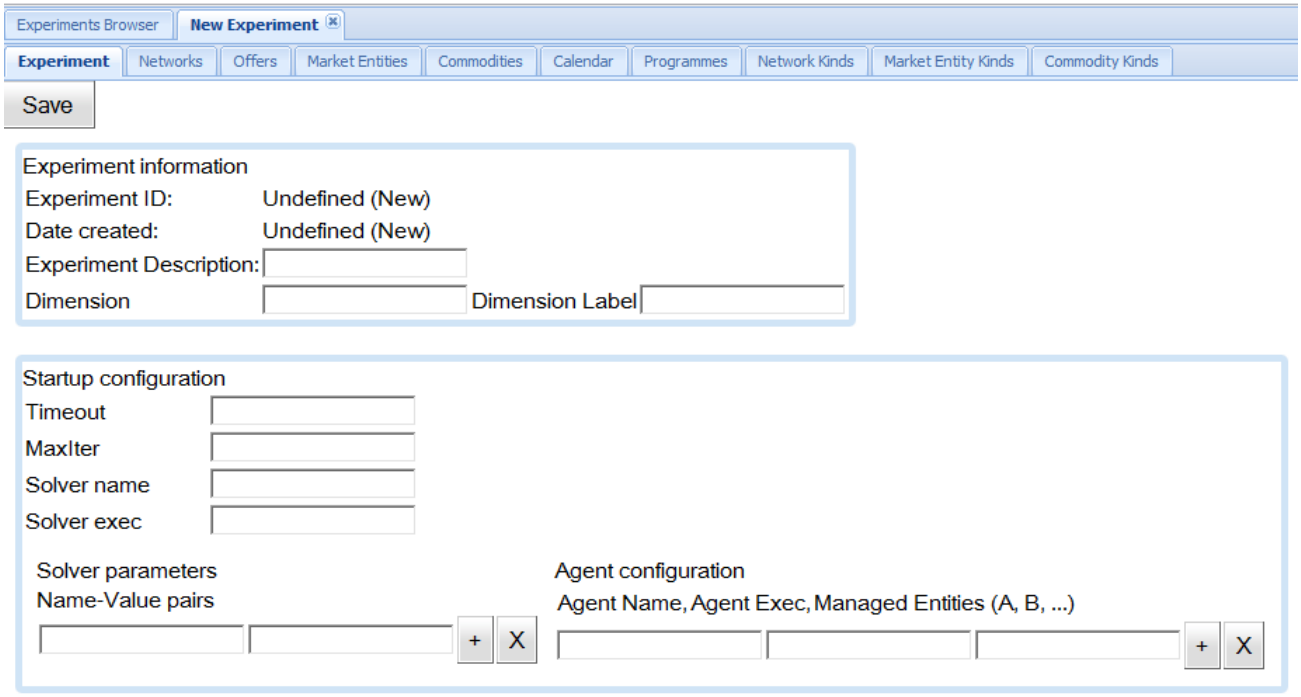
Na Rys. 20. przedstawiono ekran z widoczną funkcjonalnością generowania wykresów na podstawie danych z widoku tabelarycznego.




Rys. 20. Fragment ekranu użytkownika z wygenerowanym wykresem.

Na Rys. 20. widoczny jest wykres wygenerowany na podstawie wcześniej wybranych danych. W pola *X Axis* i *Y Axis* użytkownik wpisuje interesujące go pola z widoku tabelarycznego. W implementacji pól formularzy uwzględniono podpowiedzi – dane są pobierane dynamicznie na podstawie nazw kolumn tabeli – dzieje się to za pomocą specyficznych pól tekstowych *MultiWordSuggestOracle* z biblioteki GWT. W momencie wykonania akcji dodawania danych ze struktury drzewa eksperymentów do widoku tabelarycznego, do słownika każdego z tych pól trafia nazwa dodawanej kolumny. Następnie tak stworzony słownik dodawany jest do instancji wyżej wspomnianych obiektów. W momencie wpisywania pierwszych znaków zostaną wyświetlone podpowiedzi pasujące do wzorca słownika. Dzięki temu uproszczono sposób wprowadzania danych i zmniejszono możliwość nieświadomego popełnienia przez użytkownika błędów.

Na Rys. 21. przedstawiono kolejny ekran aplikacji.




Rys. 21. Ekran aplikacji służacy do definiowania nowego eksperymentu.

Na Rys. 21. widoczna jest nowa zakładka tworzona w momencie definiowania nowego eksperymentu. W zakładce tej dostępne są pola formularza podzielone na logiczne części:

- właściwości eksperymentu (w szczególności wymiar i etykieta wymiaru),

- konfiguracja startowa solwera,
- definiowane przez użytkownika w sposób dynamiczny parametry solwera (pary: nazwa parametru - wartość),
- konfiguracja agentów (również dodawane dynamicznie).

Widoczny jest również szereg innych zakładek – z podziałem na dane charakterystyczne dla modelu $M^3$. W każdej z nich zaimplementowana jest możliwość wysyłania plików XML z danymi zgodnymi z formatem $M^3$ do pola tekstowego. Po przesłaniu każdej z definicji do formularza, istnieje możliwość edycji przed zapisaniem w bazie danych.

Podczas implementacji interfejsu użytkownika dołożono starań, aby nawigacja była łatwa i intuicyjna – w dużym stopniu wpływa to na przyjazność interfejsu.

# 4. Podsumowanie

Stworzenie platformy do obsługi eksperymentów w oparciu o model $M^3$ było zadaniem trudnym i czasochłonnym. Podstawowe utrudnienie stanowiło duże uogólnienie modelu, uniemożliwiające stworzenie użytecznej bazy danych i struktur Javy w sposób automatyczny. Dodatkowym elementem utrudniającym tworzenie platformy była potrzeba dostosowania się do wymagań wielu użytkowników, które niekiedy okazywały się rozbieżne. Powodowało to konieczność zbudowania czegoś równie ogólnego jak sam model $M^3$. Na szczęście z pomocą przyszły nowoczesne technologie usprawniające pracę ze skomplikowanym projektem i automatyzujące pewną część zadań. Dzięki nim w ogóle stworzenie bazy danych do przechowywania eksperymentów i operacje na nich okazały się możliwe. Zastosowane narzędzia języka Java są najlepszym rozwiązaniem na potrzeby powstałej aplikacji. Technologie takie jak Spring, Hibernate czy GWT idealnie wpasowują się w obecne trendy tworzenia dynamicznych aplikacji internetowych. Wpisuje się to w tendencję rozwoju internetu w stronę Web 2.0, gdzie strona internetowa nie jest tylko zwykłym, statycznym plikiem HTML z elementami graficznymi, lecz w pełni dynamiczną i reagującą na akcje użytkownika aplikacją. Jednak same technologie nie mogą wykonać pracy koncepcyjnej i sprawić, że poszczególne elementy zaczną ze sobą współgrać. W tym celu wymagana jest duża znajomość konkretnych rozwiązań i doświadczenie w ich użytkowaniu. Dopiero po przekroczeniu tego progu można zacząć właściwe projektowanie architektury systemów. Obecnie burzliwy rozwój oprogramowania do automatyzacji zadań (w szczególności tworzenia kodu) powoduje, że praca koncepcyjna bardziej przesuwa się w stronę umiejętnego łączenia

różnych technologii. Zaproponowane rozwiązania w każdym stopniu starają się wykorzystać maksimum możliwości każdej z wybranych technologii.

Uniwersalność modelu M3 umożliwia zastosowanie platformy do rozwiązywania nie tylko zadań równoważenia rynku, ale również szeregu innych, niekoniecznie związanych bezpośrednio z grafowym opisem i charakterem zadania. Wśród potencjalnych zastosowań można wymienić:

- Rozwiązywanie zadania komiwojażera
- Kolorowanie grafów
- Rozwiązywanie zadań transportowych
- Implementacja algorytmów genetycznych, symulowanego wyżarzania itp.

Dzięki jego ogólności, model M3 można traktować jako uniwersalny pojemnik na wszelkiego typu dane. Należy zauważyć, że kodowanie dużej ilości danych w formacie XML nie jest efektywne – zatem prezentowane tutaj środowisko obliczeniowe będzie wydajne tylko dla zadań, w których konieczność wymiany informacji nie zachodzi często. Dlatego platforma nie będzie przydatna np. w zadaniu odwracania macierzy czy rozwiązywania równań różniczkowych o stałych rozłożonych.

Aby udowodnić stosowalność platformy do zadań grafowych, zaimplementowano i rozwiązano z jej pomocą klasyczne zadanie komiwojażera. Znaleziono rozwiązanie dokładne dla grafu o 10 wierzchołkach (użyto współrzędnych geograficznych 10 największych miast w Niemczech). Model M3 wykorzystano następująco: zdefiniowano w sekcji *networks* wierzchołki grafu. Zrezygnowano z definiowania krawędzi z uwagi na wydajność komunikacji. Tabele odległości (założono, że graf jest pełny) są obliczane przez każdego z agentów obliczeniowych podczas pierwszej iteracji. W sekcji *commodities* zdefiniowano „towary": są nimi kolejne etapy podróży komiwojażera. W sekcji *offers* zdefiniowano oferty: są nimi aktualne rozwiązania zadania, przedstawiane przez agentów. Dla każdego z etapów-towarów zdefiniowanych jest tyle ofert, ile jest agentów obliczeniowych. Oferta zawiera numer etapu (kodowany w polu *minVolume*) oraz odpowiadający mu numer wierzchołka w grafie (kodowany w polu *offeredPrice*). Agenci, w osobnych wątkach obliczeniowych, dokonują przeglądu kolejnych rozwiązań. W wątku głównym zaś zgłaszają okresowo, na żądanie solwera, kolejne najlepsze znalezione dotąd rozwiązania. Zadaniem solwera jest porównanie ofert-rozwiązań.

W opisywanym przykładzie posłużono się standardowymi, predefiniowanymi polami M3 w celu zamodelowania i przekazania rozwiązania zadania komiwojażera. Gdyby, w jakimś problemie obliczeniowym, nie udało się wykorzystać pól standardowych, użytkownik może zdefiniować własne pola (w sekcjach *network kinds*, *commodity kinds* i *market entity kinds*) modelu M3. Ich liczba i typ nie są ograniczone; będą one przetwarzane przez platformę w sposób równoprawny.

Należy zaznaczyć, że głównym zastosowaniem platformy jest nie *rozwiązywanie* zadań, lecz

*badanie algorytmów rozwiązywania* zadań. Zatem, największą korzyścią z zastosowania platformy do zadań obliczeniowych dużej skali mogłoby być szybkie (prostota interfejsów) prototypowanie nowych algorytmów, a następnie komfortowe i rzetelne badania porównawcze ich własności. Można w ten sposób rozwijać np. nowe heurystyki dla dobrze określonych problemów, z istniejącym już rozwiązaniem referencyjnym.

# Bibliografia


[1] *$M^3$ - Multicommodity Market Data Model* (http://www.openm3.org/)

[2] *Java architecture for XML binding* (JAXB)
(http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, https://jaxb.dev.java.net/)

[3] *Hibernate in action*, Ch. Bauer, G. King, Manning Publications Co., 2005

[4] *Spring in action*, C. Walls, R. Breidenbach, Manning Publications Co., 2005

[5] *GWT in practice*, R. Cooper, Ch. Collins, Manning Publication Co., 2008

[6] Java Message Service (http://java.sun.com/products/jms/)

[7] *JSON: Wprowadzenie* (http://www.json.org/json-pl.html)

[8] *Gwt-ext* (http://gwt-ext.com/docs/2.0.4/)

[9] *Standard RFC 4180* (http://tools.ietf.org/html/rfc4180)

[10] *SmartGWT* (http://code.google.com/p/smartgwt/)

[11] *Modeling Language for Mathematical Programming* www.ampl.com