UMCS

Uniwersytet Marii Curie–Skłodowskiej
w Lublinie
Wydział Matematyki, Fizyki i Informatyki

Kierunek: Matematyka

Specjalność: Informatyczna

Aleksander Wójcik

nr albumu 243783

Aplikacja typu komunikator. Projekt i implementacja w języku programowania obiektowego

The communicator application. The project and implementation in object-oriented language

Praca licencjacka

napisana w Zakładzie Informatyki

pod kierunkiem dr Beaty Byliny

Lublin 2014

Spis treści

Rozdział 1

Wstęp

Celem tej pracy jest omówienie interfejsu umożliwiającego komunikację z bazą danych oraz technologii umożliwiającej komunikację w rozproszonej aplikacji typu serwer–klient oraz opisanie implementacji forum dyskusyjnego typu komunikator.

Praca składa się z dwóch części. W części teoretycznej szczegółowo opisano mechanizm JDBC (ang. *Java Database Connectivity*) będący podstawą działania aplikacji oraz przedstawiono mechanizm zdalnego wywołania metod RMI (ang. *Remote Method Invocation*) w Javie będący alternatywną wersją działania aplikacji. Wyjaśniono również sensowność wykorzystania w określonych sytuacjach wzorców projektowych.

Część praktyczna zawiera opis implementacji aplikacji forum dyskusyjnego typu komunikator oraz podstawową dokumentację. Aplikacja jest forum dyskusyjnym. Zalogowani użytkownicy mogą przeglądać forum, dodawać posty i tematy. Program składa się klienta żądającego wykonania zdalnych usług na serwerze. Treść (konta użytkowników, posty, tematy) przechowywana jest w bazie danych, do której dostęp ma tylko serwer.

Rozdział 2

Wprowadzenie

Aplikacja typu klient–serwer składa się z dwóch programów: klienta i serwera. Serwer udostępnia usługi. Klient żądając wykonania usługi, wysyła komunikat żądania (and. *request*) do serwera oraz czeka na odpowiedź. Program serwera pełni rolę nadrzędną w stosunku do programu klienta.

Program rozproszony może wykonywać się na różnych komputerach. To, co różni go od programu typu klient-serwer, to fakt, że żadna jego część nie jest wyróżniona – nie pełni roli nadrzędnej (serwera) w stosunku do pozostałych.

Aplikacja rozproszona jest programem, który może wykonywać się na różnych komputerach. Składa się z modułów komunikujących się ze sobą, które mogą znajdować się na różnych komputerach i mieć dostęp do różnych baz danych. Zatem z aplikacji może korzystać jednocześnie wielu użytkowników pracujących na komputerach fizycznie oddalonych od siebie. W odróżnieniu od aplikacji typu klient-serwer nie jest wyróżniona żadna jednostka centralna (serwer) w stosunku do pozostałych.
Aplikacja rozproszona może być napisana w wielu językach programowania, m.in. w języku Java.

Sposób programowania aplikacji rozproszonych w sposób naturalny jest trudniejszy i bardziej złożony od programów działających na jednym komputerze. Problemem jest komunikacja między modułami, która wymaga jednolistego protokołu sieciowego oraz może być niestabilna, a także zmniejszona szybkość działania. Jednym z mechanizmów rozwiązujących część problemów implementacyjnych jest RMI (ang. *Remote Method Invocation*). Najczęściej składa się z dwóch programów — klienta i serwera — uruchamianych na różnych maszynach wirtualnych Javy (choć możliwe jest uruchomienie programów na jednej maszynie wirtualnej). Główną zaletą technologii jest ukrycie sieciowego aspektu aplikacji przed programistą. Dzięki temu wywołanie metody z obiektu znadującego się na innej maszynie wirtualnej prawie nie różni się od wywołania z obiektu lokalnego. Wadą jest ograniczenie wykorzystania technologii wyłącznie do aplikacji napisanych w języku Java.

W przypadku obiektów komunikujących się za pośrednictwem RMI zazwyczaj stosuje się terminy klient i serwer nawet kiedy program jest rozproszony (tzn. brak wyróżnionej jednostki centralnej). Serwer oznacza obiekt, z którego wywołano metodę, natomiast klient jest tym, który ją wywołuje.

Rozdział 3

Język Java

3.1 Podstawowe reguły języka Java

3.1.1 Struktura klas

Każda klasa publiczna (tzn. jej definicja jest poprzedzona słowem kluczowym `public`) ma nazwę odpowiadającą nazwie pliku, w którym jest zdefiniowana. Zatem w jednym pliku może być zdefiniowana jedna klasa publiczna. Klasy są pogrupowane w pakiety w ten sposób, że nazwa pakietu odpowiada nazwie folderu, w którym są przechowywane pliki klas będącym w jednym pakiecie. Program napisany w języku Java[1] składa się z hierarchii klas pogrupowanych w pakiety i posiada co najmniej jedną klasę z metodą główną, tzn. z metodą, której sygnatura ma postać `public static void main(String[] args)`.

3.1.2 Obiektowość

Język Java[7] jest silnie zorientowany obiektowo. W kontekście własności języka o obiekcie można myśleć jako o składowej części programu, która może przyjmować określone stany i ma określone metody, które mogą zmieniać te stany bądź przesyłać dane do innych obiektów. Wyjątkiem od obiektowości są typy prymitywne opisane w 3.2.1 podpunkcie 3.2.1.

Inna definicja obiektu:

Obiekt[5] *jest elementem modelu pojęciowego obdarzonym odpowiedzialnością za pewien obszar świata rzeczywistego. Sposób realizacji tej odpowiedzialności zależy od samego obiektu.*

3.1.3 Dziedziczenie

W Javie wszystkie obiekty są pochodną obiektu nadrzędnego (klasy `java.lang.Object`), z którego dziedziczą podstawowe metody. Dzięki temu wszystkie klasy mają wspólny podzbiór podstawowych możliwości. W języku Java klasy mogą być wyprowadzone z innych klasy, w ten sposób dziedziną pola i metody tych klas. Z wyjątkiem klasy Object, która nie ma nadklasy, każda klasa ma jedną i tylko jedną bezpośrednią nadklasę (jednokrotne dziedziczenie). W przypadku braku wyraźnie określonej nadklasy, każda klasa jest bezwzględnie podklasą klasy Object. Podklasa dziedziczy wszystkie składowe (pola,

metody oraz zagnieżdżone klasy) z jej nadklasy. Konstruktory nie są składowymi, więc nie są dziedziczone przez podklasy, ale konstruktor może być wywołany z podklasy poprzez użycie słowa kluczowego `super()`. Klasa `Object`, zdefiniowania w pakiecie `java.lang`, definiuje i implementuje zachowania wspólne dla wszystkich klas.

Podstawową różnicą między klasami i interfejsami jest fakt, że klasy mają pola, podczas gdy interfejsy ich nie mają. Dodatkowo, instancją klasy jest obiekt, podczas gdy nie można mieć instancji interfejsu, z wyjątkiem anonimowej implementacji interfejsu. Obiekt przechowuje swoje stany w polach, które są zdefiniowane w klasie. Powodem, dla którego język Java nie pozwala na rozszerzanie więcej niż jednej klasy jest uniknięcie wielodziedziczenia stanu, które to może prowadzić do dziedziczenia pól z wielu klas. Kiedy jest tworzony obiekt przez utworzenie instancji klasy, obiekt będzie dziedziczył pola z nadklasy.

3.1.4 Klasa abstrakcyjna

Klasa abstrakcyjna jest zadeklarowana za pomocą słowa kluczowego `abstact`. Może, ale nie musi, zawierać abstrakcyjnych metod. Abstrakcyjna klasa nie może mieć instancji, ale może mieć podklasy. Abstrakcyjna metoda jest metodą zadeklarowaną bez implementacji (bez nawiasów klamrowych i zakończona średnikiem). Jeżeli klasa zawiera abstrakcyjne metody, wtedy musi być zadeklarowana jako abstrakcyjna. Kiedy abstrakcyjna klasa ma podklasę, to podklasa zazwyczaj zawiera implementacje wszystkich abstrakcyjnych metod klasy rodzica. Jednakże, jeśli nie zawiera, to musi sama być zadeklarowana, jako abstrakcyjna. Metody w interfejsie muszą być bezwzględnie abstrakcyjne, więc modyfikator abstrakt nie musi być użyty wraz z interfejsami metod (może, ale nie jest to konieczne).

Metoda wirtualna –w Javie wszystkie metody są domyślnie wirtualne[4], tzn.:

- o możliwych różnych definicjach przy konkretyzacji
- o nieznanym (w chwili kompilowania) dokładnie sposobie działania
- niekoniecznie już istniejące

Metodami niewirtualnymi[4] są:

- metody statyczne
- metody oznaczone specyfikatorem `final`
- metody prywatne

3.1.5 Polimorfizm

Słownikowa definicja polimorfizmu[4] odnosi się do zasady w biologii w której organizm lub gatunek może mieć różne formy lub etapy. Ta zasada może również być zastosowana do programowania zorientowanego obiektowo i języków takich jak Java. Mając zadeklarowaną dowolną metodę wirtualną kompilator nie wie w czasie kompilacji, której konkretnie implementacji metody będzie używał, ponieważ metoda może być zawsze nadpisana przez jej podklasę, a więc zawsze może się zmienić jej implementacja. Zatem decyzja o wyborze implementacji musi zostać odłożona do momentu wykonywania programu. Rodzaj takiego polimorfizmu nazywamy polimorfizmem czasu wykonania[6].

3.2 Typy danych w Javie

W Javie mamy dwa typy danych: typy pierwotne i typy obiektowe.

3.2.1 Typy pierwotne

Do typów pierwotnych (inaczej: prostych) należą:

- typ znakowy: `char`
- typ całkowitoliczbowy: `int, long`
- typ zmiennoprzecinkowy: `float, double`
- typ logiczny: `boolean`
- specjalny typ: `void` informujący maszynę wirtualną Javy, że dana metoda nie zwraca żadnej wartości i żadnego obiektu, a więc nie zawiera w sobie słowa kluczonego `return`

Typy pierwotne nie podlegają dziedziczeniu. Język Java nie pozwala na definiowanie własnych typów pierwotnych.

3.2.2 Typy obiektowe

Do typów obiektowych należą wszystkie typy, które dziedziczą z klasy `java.lang.Object`. Możliwe jest tworzenie nowych typów obiektowych poprzez rozszerzanie już istniejących klas i interfejsów.

3.3 Wzorce projektowe

W języku Java wielodziedziczenie klas zastąpiono:

- pojedynczym dziedziczeniem klas
- wielodziedziczeniem intefejsów
- polimorficznym pokrywaniem metod

Ułatwiło to odseparowanie logiki od implementacji oraz umożliwiło wyciąganie abstrakcji pomiędzy klasami w postaci interfejsów i klas abstrakcyjnych. Dzięki temu pewne powtarzające się rozwiązania dot. architektury aplikacji mogły zostać zauważone i opisane. Nazwano je wzorcami projektowymi.

Wzorzec projektowy — to wielokrotnie powtórzony (powtarzający się) i pozytywnie zweryfikowany schemat rozwiązania często spotykanego problemu projektowego.

Wzorzec musi być zastosowany wielokrotnie i potwierdzić swoje znaczenie poprzez częste pojawianie się w pracy projektowej.
Dzięki stosowaniu wzorców programy mogą być lepsze, łatwiej modyfikowalne, bardziej efektywne i mniej narażone na błędy.
Wzorce zostały spopularyzowane w połowie lat 90-tych przez wydanie książki "*Design Patterns: Abstraction and Reuse of Object Oriented-Design*" autorstwa E. Gamma, R. Helm, R. Johnson, J. Vlissides, w której sklasyfikowano i dobrze opisano 23 wzorce projektowe.
Wzorce dzielimy na:

- Wzorce konstrukcyjne: kontrola nad sposobem tworzenia obiektów oraz przeniesienie procesu tworzenia na inne klasy (np. `singleton`)
- Wzorce strukturalne: zarządzanie strukturą obiektów (np. `proxy`–zostanie opisany w punkcie 5.2, `fasada`–zostanie opisana w punkcie 6.4.2)
- Wzorce behawioralne: zachowanie obiektów i komunikacja między nimi

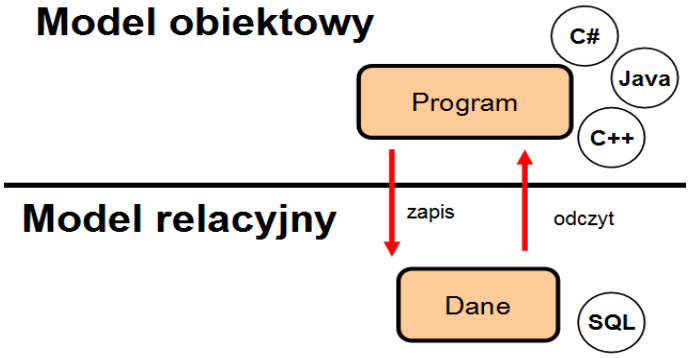
Rozdział 4

Java Database Connectivity (JDBC)

4.1 Aplikacja bazodanowa

Aplikacja bazodanowa jest to program, który w trakcie działania komunikuje się z bazą danych najczęściej w celu zapisu, odczytu lub usuwania danych. Rzadziej zmienia strukturę przechowywanych danych.

Współczesne aplikacje bazodanowe implementowane są w językach obiektowych. Do najpopularniejszych należą Java, C#. Program jest zatem zaimplementowany w obiektowym paradygmacie, natomiast dane są przechowywane w strukturze tabel, tzn. w modelu relacyjnym. Opracowanie komunikacji pomiędzy dwoma modelami jest problematyczne. Jednym z gotowych rozwiązań jest interfejs JDBC[2]. Na rysunku 4.1 przedstawiono schemat komunikacji pomiędzy modelem obiektowym a modelem relacyjnym. Interfejs JDBC umożliwia programowi zapis i odczyt danych z bazy (podpisane strzałki na rysunku).



Rysunek 4.1: Komunikacja pomiędzy modelem obiektowym a modelem relacyjnym

4.2 JDBC

Programistyczny interfejs dostępu do baz danych z poziomu Javy JDBC[2] (Java Database Connectivity API):

1. jest niezależny od maszyny bazodanowej (RDBMS, ang. `Relational Database Management System`)

2. jest niezależny od platformy sprzętowej

3. jest niezależny od systemu operacyjnego

4.2.1 Łączenie z bazą danych

Aby połączyć się z bazą należy:

1. załadować sterownik JDBC,

2. zażądać od sterownika połączenia i uzyskania go w postaci obiektu typu Connection

Załadowanie sterownika odbywa się za pomocą wywołania statycznej metody klasy `Class` o nazwie `forName` i z argumentem – nazwa klasy (sterownika) typu `String`. Metoda ta zwraca obiekt-klasę (tzn. obiekt typu `java.lang.Class`) o podanej nazwie. Jeśli klasa ta nie jest załadowana do maszyny wirtualnej Javy, następuje jej załadowanie. Klasy-sterowniki są tak skontruowane, że przy ich ładowaniu rejestrują się one jako obiekty typu Driver.

Zwykle obiekt zwracany (klasa) nie interesuje nas, dlatego w wywołaniu pomijamy zwracany rezultat.

Listing 4.1: Ładowanie sterownika i nawiązywanie połączenia w mojej aplikacji

```
1     public static final String driver = "com.mysql.jdbc.Driver";
2     public static final String databaseUrl = //...
3     public static final String user = //...
4     public static final String password = //...
5
6     public Connection getConnectionJDBC() {
7         Connection conn = null;
8         try {
9             Class.forName(driver).newInstance();//...
10            conn = DriverManager.getConnection(databaseUrl, user, password);
11        } catch (ClassNotFoundException | InstantiationException | IllegalAccessException | SQLException e){
12//...
13}
14        return conn;
15    }
```

Nad załadowanymi sterownikami kontrolę sprawuje DriverManager (nazwa klasy). Prowadzi on listę zarejestrowanych sterowników. Statyczna metoda getConnection z klasy DriverManager pozwala na uzyskanie połączenia z bazą, której URL podajemy jako argument metody. DriverManager przegląda listę zarejestrowanych sterowników i wybiera ten, który może połączyć się z podaną bazą. Po połączeniu z bazą zwracany jest obiekt typu Connection, który reprezentuje połączenie.

4.2.2 Wykonywanie instrukcji SQL

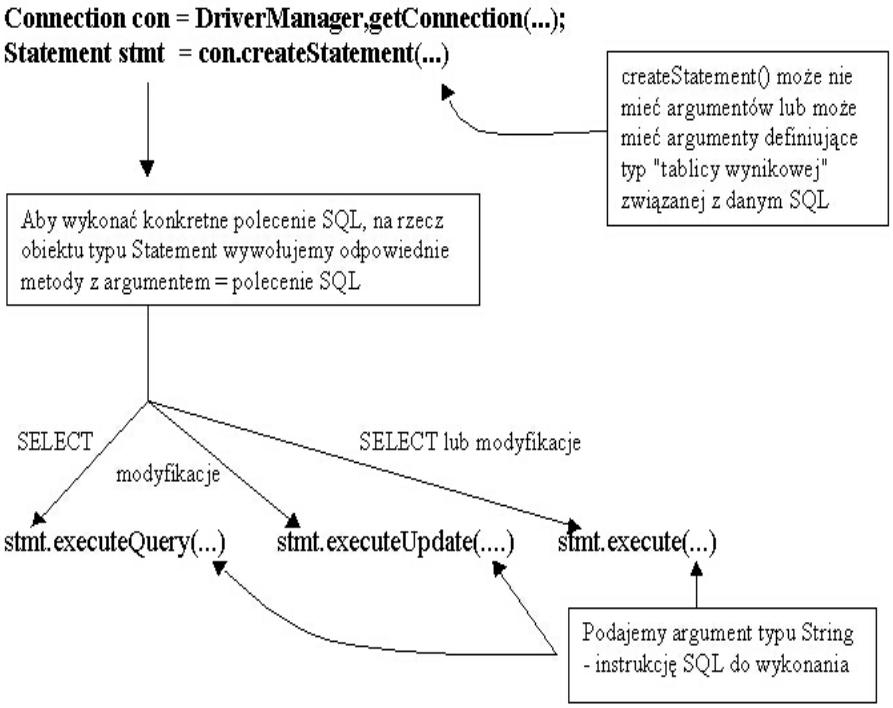
Do wykonywanie instrukcji SQL służy obiekt typu Statement(oznacza instrukcje SQL), a także obiekty typu interfejsów pochodnych:

- PreparedStatement (prekompilowane instrukcje SQL)
- CallableStatement (przechowywane procedury)

Uzyskujemy je wykonując następujące metody na obiekcie typu Connection:

- `Statement createStatement()`
- `PreparedStatement prepareStatement(String sql)`
- `CallableStatement prepareCall(String sql)`

Możliwe są również inne – mniej typowe – argumenty powyższych metod.



Rysunek 4.2: Schemat posługiwania się interfejsem Statement oraz interfejsami, które po nim dziedziczą

Na uzyskanym obiekcie implementującym interfejs Statement możemy wykonać następujące metody:

- `ResultSet executeQuery(String sql)`

  Zwraca tabelę wynikową dla polecenia SELECT...

  Obiekt ResultSet reprezentuje wiersze tabeli i można uzyskać dostęp do wszystkich jej krotek m.in. za pomocą metody `boolean next()` przesuwającą kursor do przodu. Dostęp do kolejnych kolumn w wierszu uzyskujemy za pomocą metod: `String getString(int columnIndex), Integer getInt(int columnIndex)` i tym podobnych, gdzie columnIndex to numer kolumny w tabeli wynikowej.

- `int executeUpdate(String sql)`

  Używany do poleceń CREATE TABLE..., DROP TABLE..., INSERT..., UPDATE..., DELETE...

Nie wykonuje polecenia SELECT...

Zwraca liczbę zmodyfikowanych rekordów lub –1 (np. dla polecenia CREATE...)

- `boolean execute(String sql)`

  Wykonuje dowolną instrukcję SQL i zwraca wartość logiczną: true – jeśli powstała tabela wynikowa, false – jeśli nie

4.2.3 Wyjątki

Wymienione powyżej metody:

- `int executeUpdate(String sql)`
- `boolean execute(String sql)`
- `ResultSet executeQuery(String sql)`
- `Statement createStatement()`
- `PreparedStatement prepareStatement(String sql)`
- `CallableStatement prepareCall(String sql)`

mogą zgłaszać wyjątek klasy SQLException. Ponieważ nie dziedziczy ona z klasy RuntimeException, to taki wyjątek musimy obsłużyć, tzn. umieścić wywołanie metody w klauzuli try-catch lub umieścić w sygnaturze metody zewnętrznej informacje, że metoda może wyrzucić dany wyjątek (tzn. przekazać obsłużenie wyjątku wyżej).
Z obiektu klasy SQLException możemy się dowiedzieć m.in. jaki kod SQL błędu został zgłoszony (`int getErrorCode()`).

Rozdział 5

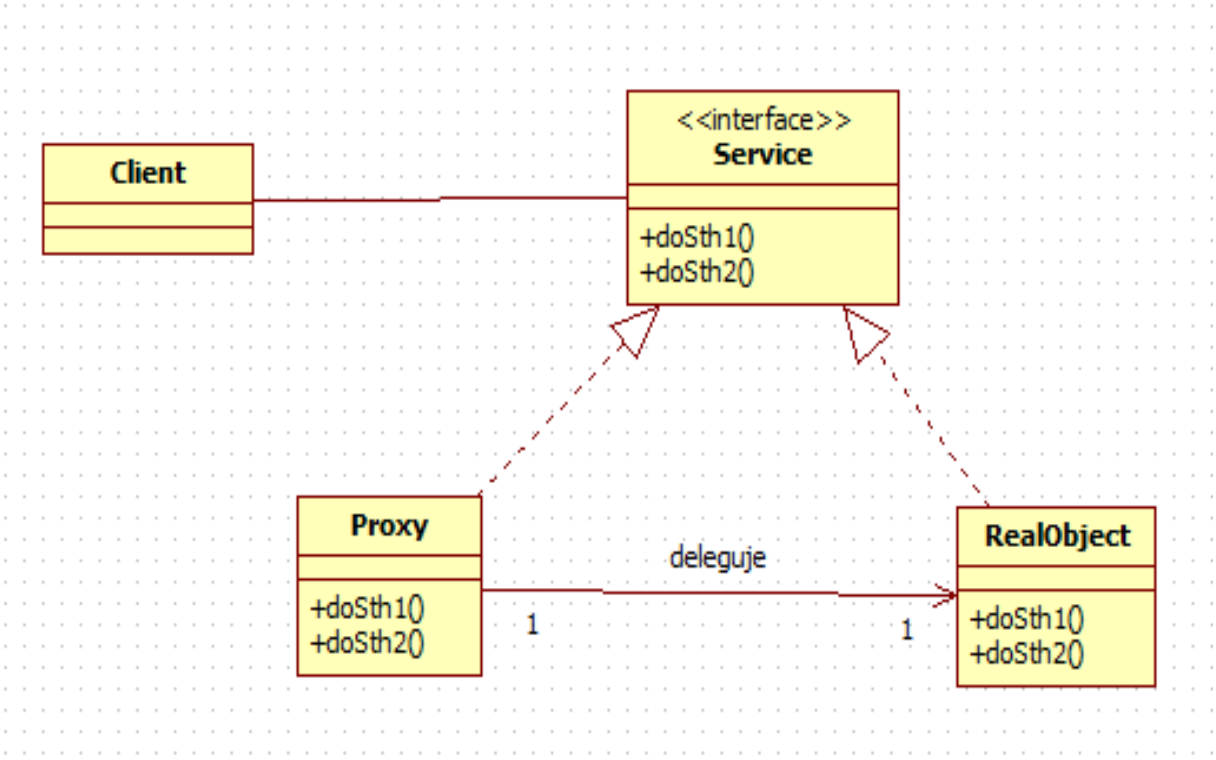
Remote Method Invocation (RMI)

5.1 Zastosowanie

Typowe zastosowanie RMI[3] polega na umożliwieniu obiektowi klienta wywołanie metody z obiektu serwera, który działa na innej maszynie.
Wywołanie metody polega na:

- przekazaniu argumentów z klienta do serwera
  (jeśli argumenty lub wynik jest typu innego niż pierwotny należy przesłać obiekty)
- ewentualnym przekazaniu definicji klas obiektów niepierwotnych
- wstrzymaniu wątku klienta do czasu odebrania wyniku
- odebraniu wyniku

5.2 Wzorzec proxy



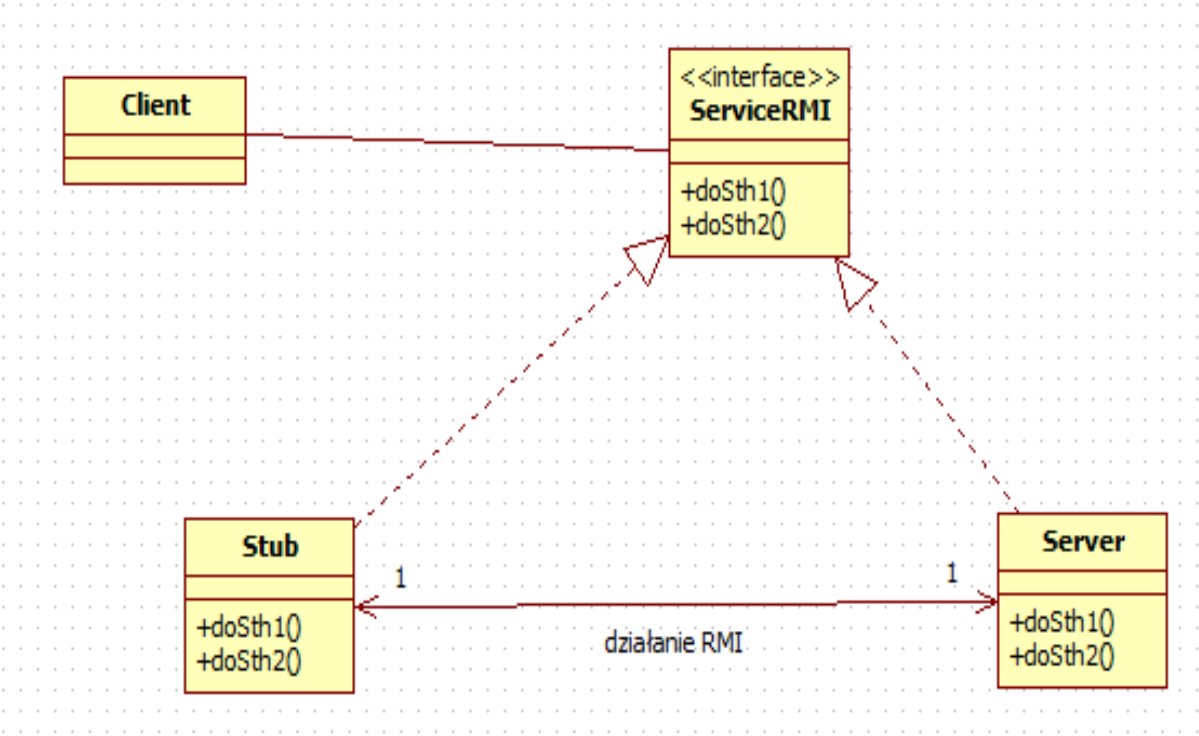
Rysunek 5.1: Wzorzec proxy

Diagram klas przedstawia:

- klienta (Client) żądającego od serwera usługi
  (tzn. chcącego wywołać na nim jakieś metody)
- interfejs Service, w którym są deklaracje metod implementowanych przez obiekt zdalny i lokalny. Są to metody, które będzie mógł klient wykonywać na obiekcie zdalnym.
- obiekt klasy RealObject — obiekt zdalny, który może znajdować się daleko.
- obiekt klasy Proxy — obiekt lokalny, który pośredniczy w wywołaniach metod z obiektu zdalnego. Reprezentuje on lokalnie obiekt klasy RealObject. Klient ma do niego dostęp — może znajdywać się na tym samym komputerze co program klienta.

Wzorzec proxy jest używany kiedy mamy jeden obiekt główny (tu: klasy RealObject), którego nie chcemy tworzyć kopii i przekazywać ją klientowi, ale chcemy, aby klient mógł się z nim komunikować, dokładniej: wywoływać metody. Wówczas delegowana jest namiastka obiektu głównego implementująca wybrane jego metody, zdefiniowane we wspólnym intefejsie. Klient wówczas wywołuje metody obiektu głównego przy pośrednictwie obiektu lokalnego.

5.3 RMI jako zastosowanie wzorca proxy



Rysunek 5.2: Diagram klas w RMI

Jednym z typowych zastosowań wzorca proxy jest RMI.

Diagram klas przedstawia:

- klienta (Client) żądającego od serwera usługi
- interfejs ServiceRMI, w którym są deklaracje metod implementowanych przez obiekt zdalny i lokalny
  (musi rozszerzać interfejs java.rmi.Remote. Każda z metod deklarowanych w interfejsie powinna zgłaszać wyjątek typu java.rmi.RemoteException.)
- obiekt zdalny (Server), który może znajdować się daleko, a który pełni rolę serwera
  (implemetuje interfejs java.rmi.Remote oraz dziedziczy z klasy java.rmi.server.UnicastRemoteObject)
- namiastka serwera (Stub) — obiekt lokalny, który pośredniczy w wywołaniach metod z obiektu zdalnego. Reprezentuje on lokalnie obiekt klasy Serwer
  (implemetuje interfejs java.rmi.Remote)

Inne nazwy obiektu lokalnego to: namiastka (z ang. *stub*), pieniek, proxy.
Pieniek (Stub) u klienta służy mu jako obiekt pomocniczy, reprezentujący faktyczny zdalny obiekt. W rzeczywistości klient odwołuje się bezpośrednio do swojego pieńka, zatem również referencja obiektu używana przez klienta jest w istocie referencją do jego pieńka. Dopiero w obiekcie pieńka następuje odwołanie

do zdalnego obiektu. Pieniek zajmuje się również przekształcaniem wywołań metod na komunikaty protokołu sieciowego.
Klient może wywoływać metody `doSth1()` oraz `doSth2()`, może być ich więcej. Przy tak zdefiniowanej architekturze aplikacji komunikacją pomiędzy obiektem zdalnym a lokalnym zajmuje się RMI, a szczegóły kominukacji są ukryte przed klientem.

5.4 Rejestr RMI

Poza definicjami interfejsów i klas ważnym elementem jest rejestr RMI. Jego zadanie polega na umożliwieniu uzyskania odniesienia do elementu zdalnego poprzez sieć. Odpowiada za znajdywanie obiektów poprzez sieć. Rejestr jest bazą danych, która przechowuje odniesienia do obiektów zdalnych (czyli implementujących interfejs Remote) i ich nazw, pod którą wcześniej zostały zarejestrowane, a które zostały uruchomione na tym samym komputerze, co rejestr.

Aby umieścić w rejestrze zdalny obiekt trzeba go najpierw zarejestrować. Służą do tego metody statyczne klasy java.rmi.Naming:

- `static void bind(String name, Remote obj);`
- `static void rebind(String name, Remote obj);`

Rejestrują one obiekt obj pod nazwą name. Jeśli podana nazwa name jest już w użyciu, to pierwsza (bind) zgłasza wyjątek AlreadyBoundException, natomiast druga (rebind) zastępuje stary obiekt nowym. Aby uzyskać odniesienie (referencje) do zdalnego obiektu trzeba znać i podać jego nazwę.

5.5 Zarządca bezpieczeństwa

Zarządca bezpieczeństwa jest obiektem klasy java.lang.SecurityManager lub jej podklasy (aktualnie tylko RMISecurityManager). Jego zadaniem jest zezwalanie lub nie na wykonywanie przez aplikację czynności, które mogą być potencjalnie niebezpieczne (np. zakończenie maszyny wirtualnej, zapis na dysk lokalny czy też drukowanie). Zarządca bezpieczeństwa ma możliwość sprawdzenia części skutków wykonywanej operacji przed jej wykonaniem i ma możliwość ją anulować jeśli uzna, że wykonanie jej jest niebezpieczne dla systemu - wtedy zgłosi wtedy wyjątek SecurityException. W przypadku RMI niebezpieczne może być wykonywanie kodu pobranego z sieci.

Zarządcę bezpieczeństwa należy zainstalować jednokrotnie, najlepiej na początku programu. Należy również zadbać o to, by na maszynie wirtualnej działał tylko jeden zarządca bezpieczeństwa. W programach, które korzystają z RMI, powinien to być obiekt klasy RMISecurityManager. Zarządcę instalujemy zazwyczaj kodem:

Listing 5.1: Zainstalowanie zarządcy bezpieczeństwa.

```
1 if (System.getSecurityManager() == null) {
2     System.setSecurityManager(new RMISecurityManager());
3 }
```

Zarządcę bezpieczeństwa musi instalować klient pobierający z sieci odniesienie do obiektu zdalnego, którego klasa nie jest znana lokalnie. Wtedy musi zostać przesłana i wprowadzona do maszyny wirtualnej klienta klasa namiastki zdalnego obiektu. W przypadku serwera nie zawsze jest to konieczne.

5.6 Pliki polityki

Zachowanie zarządcy definiują pliki polityki (tekstowe), w których opisuje się zakres pozwoleń udzielanych aplikacji (np. umożliwienie dostępu do określonego katalogu na dysku, bądź obiór danych z określonego komputera w sieci). Składnia tych plików jest opisana :

- w dokumentacji klasy java.lang.SecurityManager
- w dokumentach docs/guide/security/permissions.html oraz
- docs/guide/security/PolicyFiles.html

dostarczanych wraz z dokumentacją SDK Javy. Przykład pliku polityki, zezwalającego na wszystko:

Listing 5.2: Plik polityki

```
1 grant {
2     permission java.security.AllPermission;
3 };
```

5.7 Codebase

Kolejną bardzo ważną dla RMI właściwością jest java.rmi.server.codebase (w skrócie codebase) określającą położenie klas, które muszą być przesyłane do klientów podczas komunikacji. Należą do nich przede wszystkim klasy namiastek obiektów zdalnych, jak również zdalne interfejsy.

Przesyłanie klasy nie jest konieczne jeśli klasa przesyłanego obiektu jest widoczna u odbiorcy na jego ścieżce poszukiwań klas (classpath). Wówczas maszyna wirtualna załaduje klasę, którą widzi na ścieżce. Takie zachowanie może być przyczyną subtelnych błędów. Np. klasa dostawcy przesyłana przez sieć może nazywać się tak samo jak inna klasa widoczna na ścieżce classpath odbiorcy, albo — co gorsza — inna wersja tej samej klasy.
Położenie klas jest podawane w formacie URL, który oprócz lokalizacji w systemie plików specyfikuje też protokół służący do transportowania pliku. RMI obsługuje 3 protokoły: FILE, FTP i HTTP. Pierwszy z nich ma zastosowanie tylko w przypadku, gdy maszyny wirtualne serwera i klienta pracują na tym samym systemie plików. Pozostałe wykorzystują zewnętrzne aplikacje w celu przesłania plików z klasami - serwer WWW lub FTP.

Rejestr będzie poszukiwał klas obiektów, które są w nim rejestrowane na ścieżce poszukiwań klas classpath. Jeśli ich tam nie znajdzie, to wykorzysta URL zawarty we właściwości codebase przesłanej podczas rejestracji obiektu do odnalezienia jego klasy. Podobnie zachowuje się klient pobierający odniesienie do obiektu z rejestru.

5.8 Kroki RMI

Projektując aplikację z wykorzystaniem RMI należy kierować się następującym schematem postępowania:

1. Przygotowanie interfejsów

   Dostęp do obiektu zdalnego odbywa się za pośrednictwem interfejsu, który definiuje zachowanie się obiektu, tzn. definiuje metody, które muszą być zaimplementowane w klasach implementujących go. Interfejs musi być dostępny po stronie klienta tzn. na jego ścieżce poszukiwań klas (codebase), tak aby mógł on wywoływać metody serwera. Interfejs musi też być znany po stronie serwera, ponieważ jest implementowany przez jego klasę.

   Po przygotowaniu potrzebnych interfejsów można przystąpić do implementacji klas serwera i klienta.
   Dzięki temu, że usługi udostępniane przez zdalny obiekt (serwer) w postaci metod zostały już zdefiniowane w interfejsie można niezależnie stworzyć implementację klienta i serwera. W szczególności możliwe jest zaimplementowanie klasy klienta kiedy mamy już działający serwer

2. Przygotowanie obiektu zdalnego

   W metodzie startowej `public static void main(String[] args)` klasy serwera należy umieścić umieścić instrukcje, które:

   (a) zainstalują zarządcę bezpieczeństwa

   (b) utworzą eksportowane obiekty

   (c) zgłoszą eksportowane obiekty w rejestrze, aby były dostępne dla innych maszyn wirtualnych

   (d) skompilować kod serwera

   (e) wygenerować namiastki klas obiektów zdalnych

   (f) utworzyć rejestr RMI

   Po wykonaniu tych czynności serwer jest gotowy do uruchomienia.

3. Przygotowanie klienta

   W metodzie startowej `public static void main(String[] args)` klasy klienta należy umieścić umieścić instrukcje, które:

   (a) zainstalują zarządcę bezpieczeństwa

   (b) pobiorą odniesienie do zdalnego obiektu z rejestru

   Po wykonaniu tych czynności możemy pozostałą część kodu tworzyć w taki sposób jakby wszystko działało lokalnie.

Rozdział 6

Implementacja

Poniższy rozdział omawia szczegóły projektowe i techniczne związane z implementacją rozproszonej aplikacji.

6.1 Założenia aplikacji

Aplikacja w swoim działaniu dostarcza usługę podobną do tradycyjnych for internetowych typu komunikator. Zwykle fora są dostępne z wykorzystaniem przeglądarki WWW. Poniższa aplikacja nie wymaga korzystania z przeglądarki. Niezbędne jest jednak zainstalowane oprogramowanie JDK(ang. *Java Development Kid*) w wersji 1.7, a także znajomość adresu oraz portu pod którym działa uruchomiona usługa nazewnicza, poprzez którą serwer udostępnia usługi.

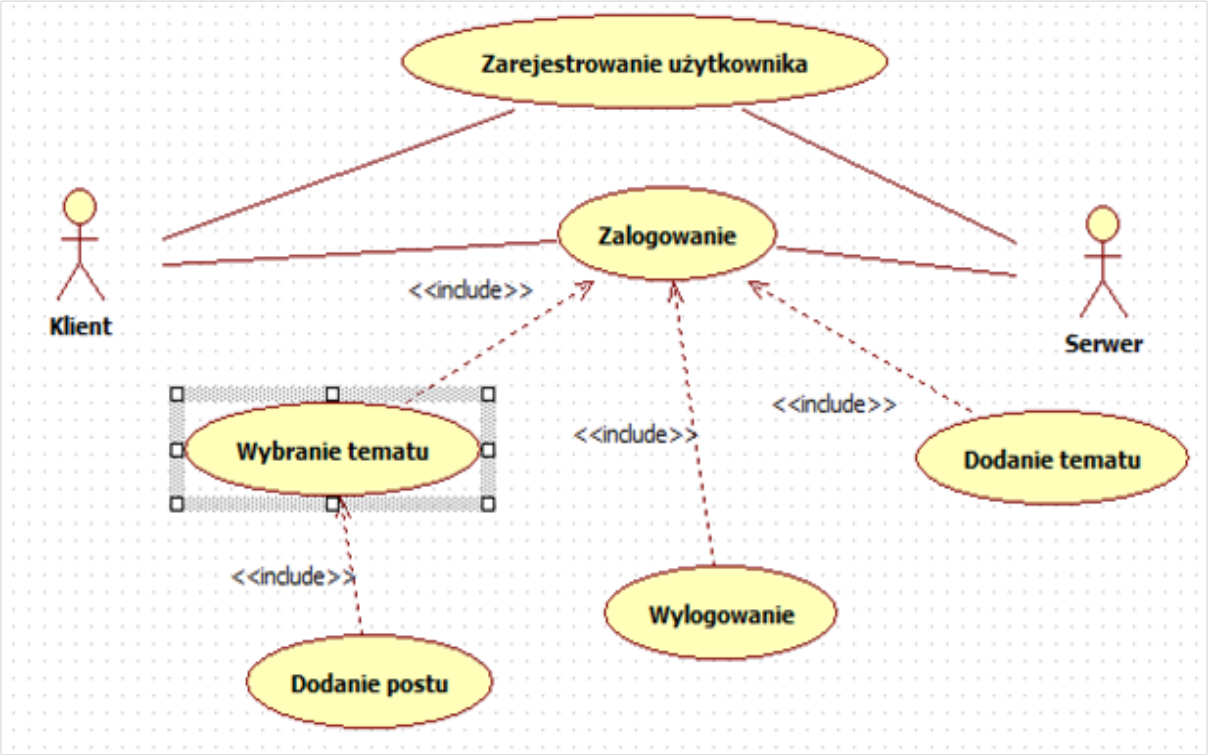
6.2 Projekt aplikacji

UML (ang. *Unified Modeling Language*, czyli Zunifikowany Język Modelowania) jest to język modelowania graficznego przeznaczony do modelowania różnego rodzaju systemów. Tworzy się w nim modele systemów informatycznych, które mają dopiero powstać. UML jest zazwyczaj używany wraz z jego reprezentacją graficzną — jego elementom przypisane są symbole, które wiązane są ze sobą na diagramach. Celem twórców języka UML było zdefiniowanie jednolitego systemu modelowania w formie graficznej. Tak się jednak nie stało – każdy program do tworzenia diagramów UML udostępnia inne narzędzia i różniące się w niewielkim stopniu oprawą graficzną. Wszystkie schematy przygotowane w tej pracy korzystają z narzędzia WhiteStartUML.

Dobrym narzędziem do opisu funkcjonalności systemu w formie graficznej jest diagram przypadków użycia zaprojektowany w języku UML. Przestawia on przypadki użycia, aktorów oraz związki między nimi.

Główne funkcjonalności systemu:

- Niezalogowany użytkownik może się zarejestrować i zalogować
- Zalogowany użytkownik może napisać post lub założyć nowy temat
- Zalogowany użytkownik może się wylogować



Rysunek 6.1: Diagram przypadków użycia opisujący jakie działania może podjąć użytkownik aplikacji

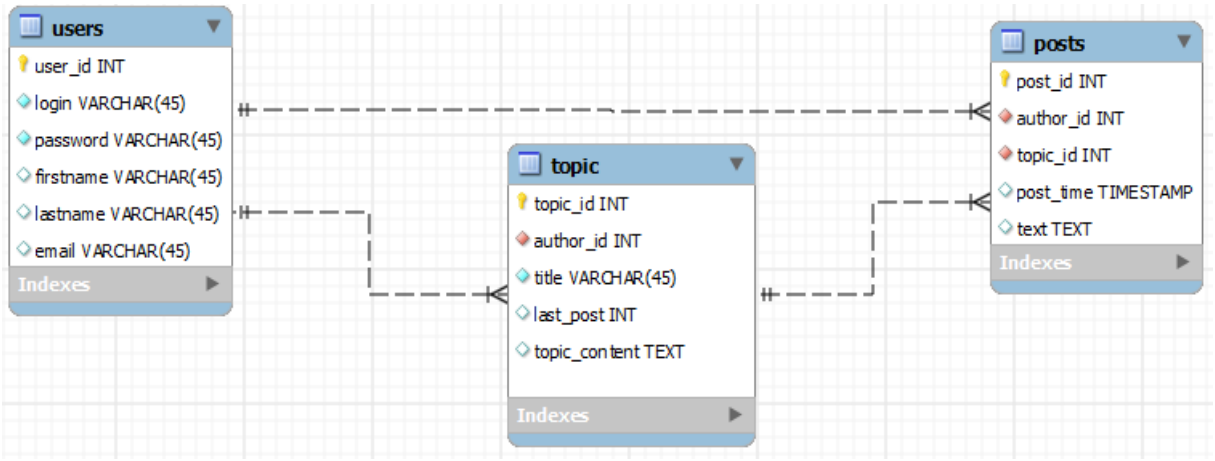
6.3 Model bazy danych

Wszystkie dane przechowywane na serwerze bazodanowym to:

- konta użytkowników
- treści tematów
- treści postów

Aplikacja wykorzystuje serwer MySql hostowany pod adresem: `http://db4free.net/`.

Fizyczna budowa bazy - tj. nazwy tabel, ich atrybuty i powiązania - są widoczne na poniższym rysunku.



Rysunek 6.2: Model bazy danych wygenerowany w programie WorkBench

1. Tabela users jest odpowiedzialna za przechowywanie informacji o użytkownikach, którzy już posiadają konto na forum. Każdy jej wiersz reprezentuje dane profilu zarejestrowanej osoby, tj. login, widoczny dla innych użytkowników forum, którym będą podpisywane wypowiedzi na forum, imię,nazwisko oraz adres e-mail.
   Kluczem głównym tej tabeli jest id użytkownika (user_id).

2. W tabeli posts przechowywane są treści postów oraz metadane: informacja który użytkownik jest autorem posta, którego tematu dotyczy post oraz którego dnia został opublikowany.
   Kluczem głównym tabeli jest id postu (post_id).

3. Tabela topic przechowuje informacje o tytułach tematów, na które rozpoczęła się na forum dyskusja. Zawiera krótki opis tematyki tematu, id autora oraz id ostatniego postu.
   Kluczem głównym tabeli jest id postu (topic_id).

6.4 Aplikacja klienta

Każda aplikacja w języku Java składa się z klas pogrupowanych w pakiety. Pliki mojej aplikacji są pogrupowane w następujące pakiety, tworzące niepodzielną całość:

6.4.1 common

W pakiecie `common` znajdują się podstawowe klasy, z których korzysta aplikacja kliencka:

common.api W pakiecie znajduje się interfejs Service.java widoczny na listingu 6.1, który opisuje całość logiki biznesowej aplikacji, tj. wszystkie funkcjonalności związane z przetwarzaniem po stronie bazy danych, które muszą zostać zaimplementowane, aby aplikacja mogła w pełni funkcjonować.

1. Service.java

Listing 6.1: Interfejs Service.java

```
1 public interface Service {
```

```
    int logIn(String login, String passwd) throws Exception;
    int register(String login, String password, String firstName, String surname, String email) throws Exception;
    int addTopic(int user_id, String title, String description) throws Exception;
    int addPost(int user_id, int topic_id, String content) throws Exception;
    Post[] loadPosts(int topic_id) throws Exception;
    Topic[] loadTopics() throws Exception;
    User getUser(int user_id) throws Exception;
}
```

common.domain W pakiecie znajdują się definicje obiektów odpowiadającym tabelom w bazie danych opisanych w poprzedniej sekcji

1. Post.java
2. Topic.java
3. User.java

common.utils Wszystkie publiczne klasy pomocnicze powinny się znajdować w pakiecie z nazwą utils.

1. StringUtils.java
   Klasa zawiera metodę `isEmpty(String s)` sprawdzającą czy napis jest pusty widoczną na listingu 6.2

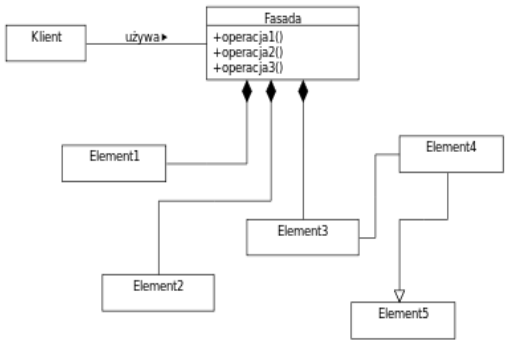
Listing 6.2: Klasa StringUtils

```
public final class StringUtils {
    public static boolean isEmpty(String s){
        return s==null||"".equals(s);
    }
}
}
```

6.4.2 client

Pakiet zawiera ścisłą logikę odpowiedzialną za całość działania aplikacji tj. wyświetlanie okienek widocznych dla użytkownika tzn. GUI (ang. *Graphical User Interface*), nawiązywanie połączenia z bazą danych oraz nawigację pomiędzy okienkami i wyświetlanie odpowiednich komunikatów.

client.facade Fasada[8] jest wzorcem projektowym, który pozwala na ukrycie zaawansowanej logiki biznesowej pod prostym interfejsem. Klient kontaktuje się tylko z fasadą, która zawiera w sobie referencje do różnych innych obiektów (do których klient nie ma dostępu) i wywołuje na nich metody. Dla klienta widoczna jest tylko mała część systemu, co jest widoczne na rysunku 6.3



Rysunek 6.3: Wzorzec fasada przedstawiony w postaci diagramu klas w UML–u

W aplikacji najpierw przekazujemy fasadzie informację, której konkretnie implementacji interfejsu `common.api.Service.java` będziemy używali. Następnie zwracamy się do fasady o wykonanie odpowiedniej metody serwisu i zwrócenie wyniku

1. FacadesEnum.java
   jest to obiekt typu wyliczeniowego (enum), za pomocą którego informujemy Fasadę, której konkretnie implementacji serwisu chcemy korzystać (listing 6.3):

Listing 6.3: Klasa FacadesEnum

```
1 public enum FacadesEnum {
2     FasadaJDBC,FasadaRMI;
3 }
```

2. Fasada.java najważniejsza klasa pakietu: to ona udostępnia dwie implementacje serwisu: FasadaJDBC oraz FasadaRMI za pomocą metody `getService()`, listing 6.4.

Listing 6.4: Najważniejsza metoda klasy – zwraca odpowiednią implementacje serwisu

```
1     public Service getService() {
2         Service intf = null;
3         switch (facade) {
4             case FasadaJDBC:
5                 intf = FasadaJDBC.getInstance();
6                 break;
7             case FasadaRMI:
8                 intf = FasadaRMI.getInstance();
9                 break;//...
10        }
11        return intf;
12    }
```

3. FasadaJDBC.java
   Implementacja serwisu, wykorzystująca mechanizm JDBC do obsługi logiki bazodanowej.

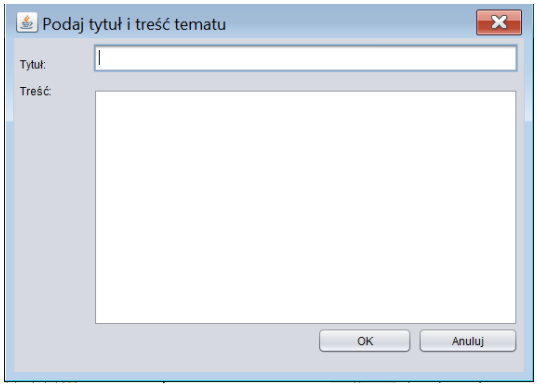
4. FasadaRMI.java
   Implementacja serwisu `common.api.Service` (listing 6.1), wykorzystująca mechanizm RMI do obsługi logiki bazodanowej. Jest zaprojektowana w sposób abstrakcyjny tak, że może obsłużyć wykonywanie powierzonych jej zadań na dowolnej maszynie wirtualnej Javy. Na listing 6.5 przedstawiono konstruktor klasy

   Listing 6.5: W konstruktorze fasady odnajdywany jest serwer. Wywołanie każdej metody serwisu `common.api.Service` jest przekazywane do serwera

```
1  public class FasadaRMI implements common.api.Service {
2  //...
3  private static Server server = null;
4          private FasadaRMI() {
5          if (System.getSecurityManager() == null) {
6              System.setSecurityManager(new RMISecurityManager());
7          }
8          try {
9              server = (Server) Naming.lookup("//localhost/RmiServer");
10         } catch (NotBoundException | MalformedURLException | RemoteException e) {
11         }
12     }
13
14     @Override
15     public int logIn(String login, String passwd) throws java.rmi.RemoteException, Exception {
16         return server.logIn(login, passwd);
17     }//...
18 }
```

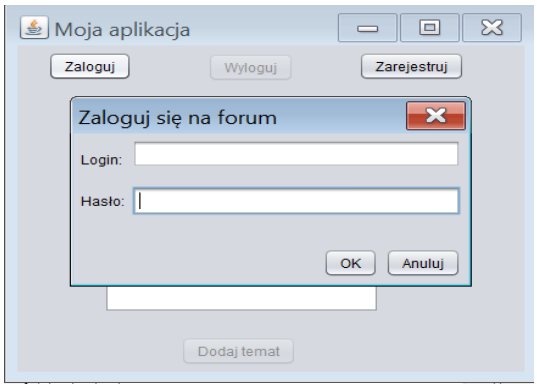
client.gui Pakiet odpowiedzialny za graficzną obsługę aplikacji oraz za wysyłanie żądań wykonania metody serwisu do fasady.

1. AddNewTopicWindow.java
   Rysunek 6.4 pokazuje okno, w którym możemy dodać nowy temat. Oczywiście tytuł tematu nie może być pusty



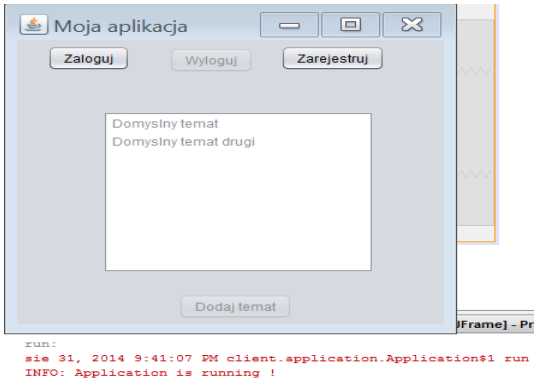
Rysunek 6.4: W oknie podajemy tytuł i opis tematu a następnie zapisujemy go na liście

2. LogInWindow.java
   Po podaniu loginu oraz hasła następuje weryfikacja czy danych użytkownik istnieje i czy podał prawidłowe hasło (Rysunek 6.5)

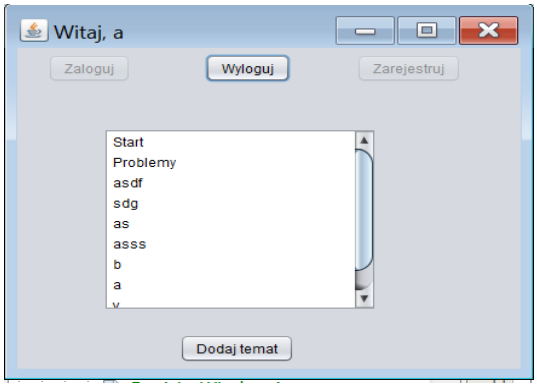


Rysunek 6.5: Formularz logowania

3. MainWindow.java
   Główne okno aplikacji pojawiające się przy starcie. Umożliwia zalogowanie i zarejestrowanie się niezalogowanemu użytkownikowi. Na rysunkach przedstawiono aktywne przyciski dla niezalogowanego (Rysunek 6.6) i zalogowanego (Rysunek 6.7) użytkownika

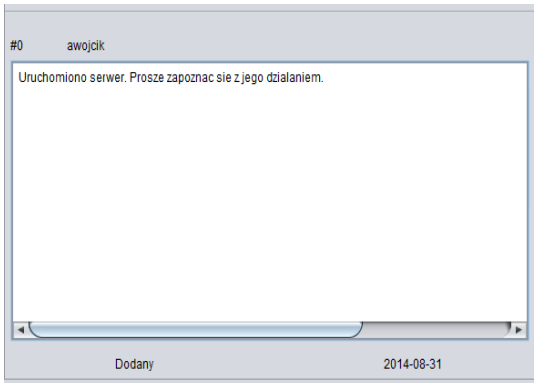


Rysunek 6.6: Wygląd okna dla niezalogowanego użytkownika. W logach odczytujemy, że aplikacja rozpoczęła działania



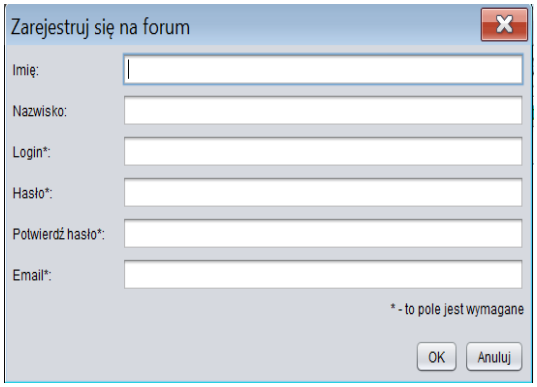
Rysunek 6.7: Wygląd okna dla zalogowanego użytkownika o loginie "a"

4. PostPanel.java
   Każdy post jest pakowany w jedno okienko opisane w tej klasie. W oknie (Rysunek 6.8) umieszczony jest autor postu, treść postu i data dodania



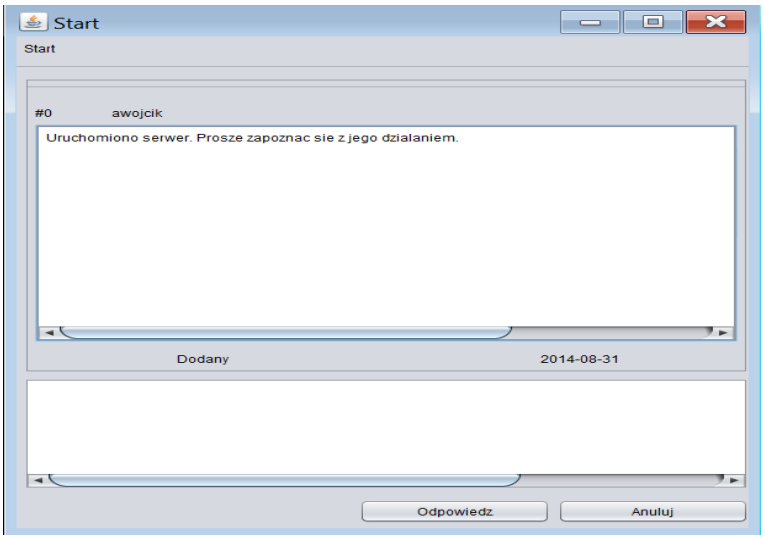
Rysunek 6.8: Przykładowy post

5. RegisterWindow.java
   Niezalogowany użytkownik może się zarejestrować wypełniając formularz (Rysunek 6.9)



Rysunek 6.9: Formularz rejestracji

6. TopicForm.java
   W oknie dyskusji (Rysunek 6.10) można przeglądać starsze posty i odpowiadać na nie pisząc nowe posty



Rysunek 6.10: Okienko dyskusji dot. konkretnego tematu

client.application Pakiet zawiera klasę Application uruchamiającą aplikację, klasę ConnectionJDBC nawiązującą połączenie z bazą oraz klasę Configuration, która odczytuje podstawową konfigurację, (przede wszystkim login użytkownika, hasło oraz ustawienia bazy danych) z pliku `/resources/config.properties`. Klasa Configuration ma wszystkie metody i konstruktor chroniony (oznaczony słowem kluczowym `protected`), zatem komunikuje się tylko z klasami z tego pakietu

1. Application.java
   Klasa uruchamiająca aplikację, listing 6.6

Listing 6.6: Metoda główna aplikacji uruchamiająca program

```
1  public static void main(String args[]) {
2      java.awt.EventQueue.invokeLater(new Runnable() {
3          public void run() {
4              final FacadesEnum facade = new Configuration().getFacade();
5              new MainWindow(facade).setVisible(true);
6              logger.info("Application is running !");
7          }
8      });
9  }
```

2. ConnectionJDBC.java
   Odpowiedzialnością tej klasy jest połączenie się z bazą i zwrócenie obiektu połączenia (`java.sql.Connection`), na którym fasada będzie wykonywać operacje bazodanowe. Na listingu 6.7 przedstawiono najważniejszą jej metodę

Listing 6.7: Metoda łącząca się z bazą i zwracająca obiekt połączenia

```
1   public Connection getConnectionJDBC() {
2       Connection conn = null;
3       this.loadConfigurationFromProperties();
4       try {
5           Class.forName(driver).newInstance();
6           logger.info("New Instance");
7           conn = DriverManager.getConnection(databaseUrl, user, password);
8       } catch (ClassNotFoundException | InstantiationException | IllegalAccessException | SQLException e) {
9           final String message = "Nie polaczylem sie z baza w FasadaJDBC";
10          logger.log(Level.SEVERE, message, e);
11      }
12      return conn;
13  }
```

3. Configuration.java

6.4.3 resources

Jest to katalog, w którym trzymane są zasoby, które mogą pomóc w uruchomieniu i dokumentacji aplikacji. W katalogu mogą się pojawić pliki z rozszerzeniem innym niż .java.

1. create_table.sql
   Skrypt tworzący niezbędne tabele i uzupełniający je przykładowymi danymi

2. config.properties
   W pliku przechowywane są ustawienia dot. konfiguracji, tj. adres i port serwera z bazą, login użytkownika oraz hasło, sterownik do bazy oraz wersję fasady, z której przy danym uruchomieniu aplikacja będzie korzystała. Wyciągnięcie konfiguracji poza aplikację pozwala nam na połączenie się z zupełnie inną bazą danych bez konieczności ponownej kompilacji kodu źródłowego

6.4.4 server

Pakiet składa się z jednej klasy:

1. Server.java
   Jest to klasa napisana w sposób abstrakcyjny i może być wykonana na dowolnej maszynie wirtualnej Javy.

6.5 Uruchomienie aplikacji

Aby uruchomić program należy wejść do katalogu `program` (na płycie CD) i odpalić komendę
`java -jar Forum.jar`
Ponieważ aplikacja korzysta z zewnętrznego serwera hostowanego pod adresem `http://db4free.net/` mogą pojawić się błędy serwera, na które nie mam wpływu jako autor tej pracy. Wówczas można utworzyć tabele (gotowy skrypt znajduje się w katalogu `resources`) na dowolnym innym serwerze, zmienić w ustawieniach pliku `config.properties` adres serwera, nazwę użytkownika i hasło i spróbować uruchomić program. W przypadku problemów z niezgodnością wersji biblioteki `javax.swing` lub jej brakiem odpowiedni komunikat będzie wyświetlony na konsoli.

Rozdział 7

Podsumowanie

Celem tej pracy było napisanie programu, który oferuje funkcjonalność forum dyskusyjnego typu komunikator oraz opis technologii, które program wykorzystuje, a także zagadnień projektowych związanych z architekturą programu. Do graficznego interfejsu użytkownika została wykorzystana biblioteka `javax.swing`, natomiast do przetwarzania bazodanowego skorzystałem z interfejsu JDBC. W tej postaci aplikacja działa bez zarzutu. Opisałem również mechanizm RMI, który należy wykorzystać, jeśli chcielibyśmy, aby część aplikacji (serwer) została uruchomiona na innej maszynie wirtualnej Javy.

W tym celu należy:

1. Zaimplementować metody interfejsu `common.api.Service.java` w klasie `server.Server.java`, które teraz zgłaszają wyjątek `UnsupportedOperationException`
2. Dokonać podstawowej konfiguracji opisanej w rozdziale dot. RMI
3. W pliku /resources/config.properties zmienić wpis `"fasada=JDBC"` na `"fasada=RMI"`
4. Skompilować i uruchomić program

Na płycie CD umieściłem:

1. kod źródłowy aplikacji napisany w środowisku NetBeans 7.4 (katalog projekt)
2. elektroniczną wersję pracy (katalog praca)
3. spakowany projekt wraz z folderem `resources`, wraz z plikiem wykonywalnym Forum.jar (katalog program)

Bibliografia

[1] *Java Tutorial*, Oracle Corporation, http://docs.oracle.com/javase/tutorial/

[2] *PJWSTK* http://edu.pjwstk.edu.pl/wyklady/mpr/scb/W11/W11.html

[3] *PJWSTK* http://edu.pjwstk.edu.pl/wyklady/mpr/scb/W7/W7.htm

[4] *PJWSTK* http://edu.pjwstk.edu.pl/wyklady/poj/scb/Polimorf/Polimorf.html

[5] *Ważniak* http://wazniak.mimuw.edu.pl/images/9/9d/Zpo-1-wyk.pdf

[6] *Wikipedia* http://pl.wikipedia.org/wiki/Polimorfizm_(informatyka)

[7] *Wikipedia* http://pl.wikipedia.org/wiki/Java

[8] *Wikipedia* http://pl.wikipedia.org/wiki/Fasada_(wzorzec_projektowy)