Zarządzanie projektami - Komunikator programistów
Dokumentacja techniczna

Olga Zachariasz,
Aleksander Sobol,
Bartłomiej Bułat,
Bartłomiej Hyży,
Julian Król,
Łukasz Krzyżek,
Maciej Gąsiorski,
Tomasz Szczęśniak

Informatyka Stosowana, IV rok
WEAliE, AGH

30.11.2011

Spis treści

1 Analiza zadania **2**

1.1 Cel projektu 2

1.2 Komunikator - use case 3

2 Wybór technologii **4**

3 Architektura komunikatora **4**

4 Klient XMPP **5**

5 Komunikacja pomiędzy klientami **8**

5.1 Typy komunikatów 8

5.2 Przykład komunikacji 10

1 Analiza zadania

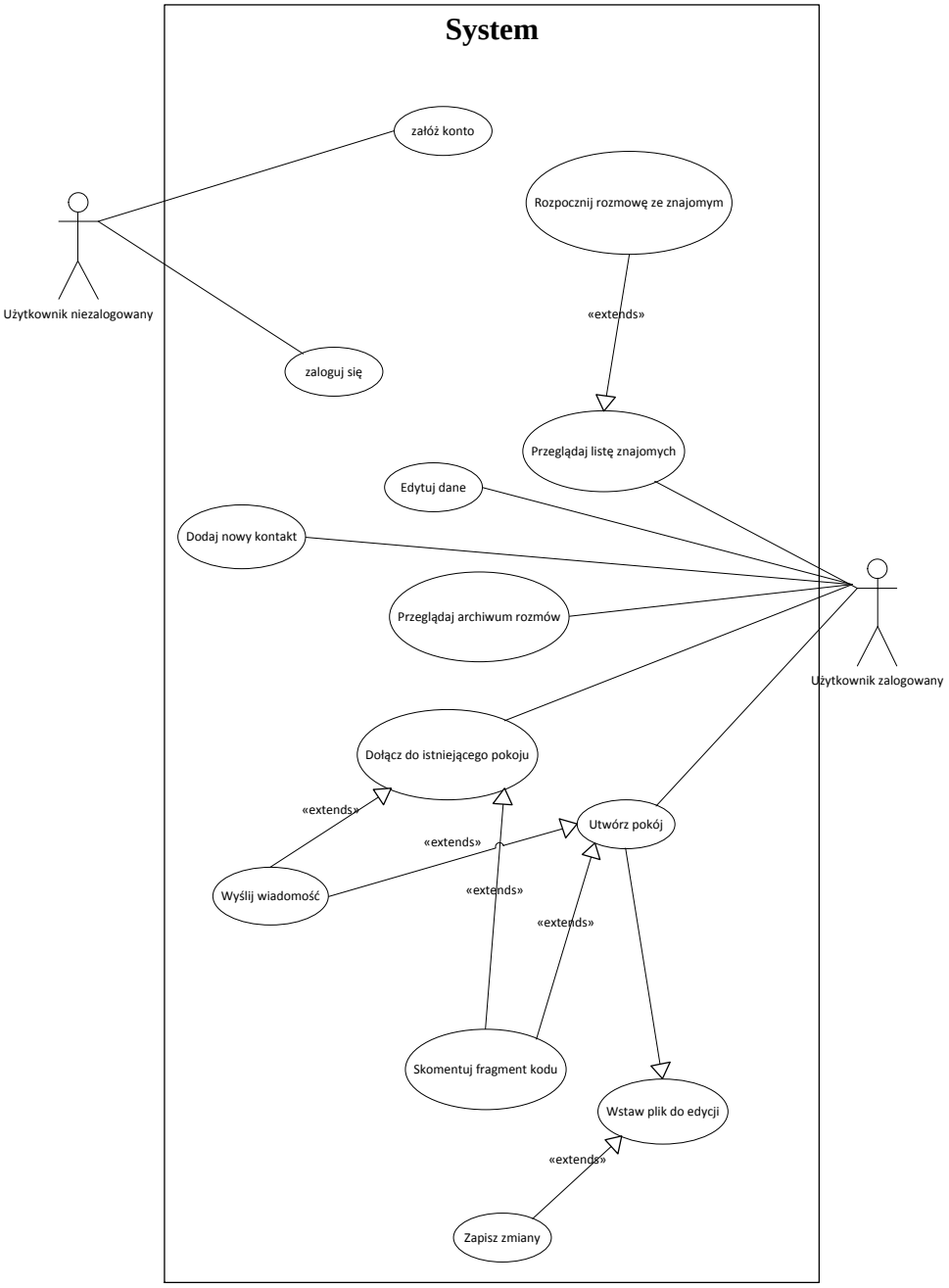
1.1 Cel projektu

Celem projektu było przygotowanie narzędzia wspierającego zdalną pracę programistów nad kodem źródłowym aplikacji.

Zadaniem naszej grupy było przygotowanie specyficznego komunikatora internetowego - przeznaczonego dla programistów. Poza typową funkcjonalnością wymiany komunikatów tekstowych, jako wymaganie postawiona została możliwość wymiany plików źródłowych, nad którymi potem przeprowadzana miała być wspólna edycja - zaznaczanie fragmentów kodu i modyfikacja istniejącego kodu. Dodatkowo komunikacja miała przebiegać pomiędzy wieloma programistami na raz, na wzór rozmów konferencyjnych znanych ze standardowych komunikatorów.

Aplikacja miała działać na zasadzie klient-serwer oraz wspierać dwa systemy operacyjne: Microsoft Windows (XP, 7) oraz Linux (osobna paczka dla dystrybucji Debian).

1.2 Komunikator - use case



2 Wybór technologii

Jako, że pozostawiono wolną rękę co do wyboru technologii, w których wykonany miał być komunikator, zespół zdecydował się na wybór bibliotek i środowisk dobrze mu znanych:

- Język programowania - Java
- IDE - NetBeans
- Protokół komunikacji - XMPP[1]
- Biblioteka do komunikacji przez XMPP - Smack API[2]
- Biblioteka GUI - Swing, elementy AWT

3 Architektura komunikatora

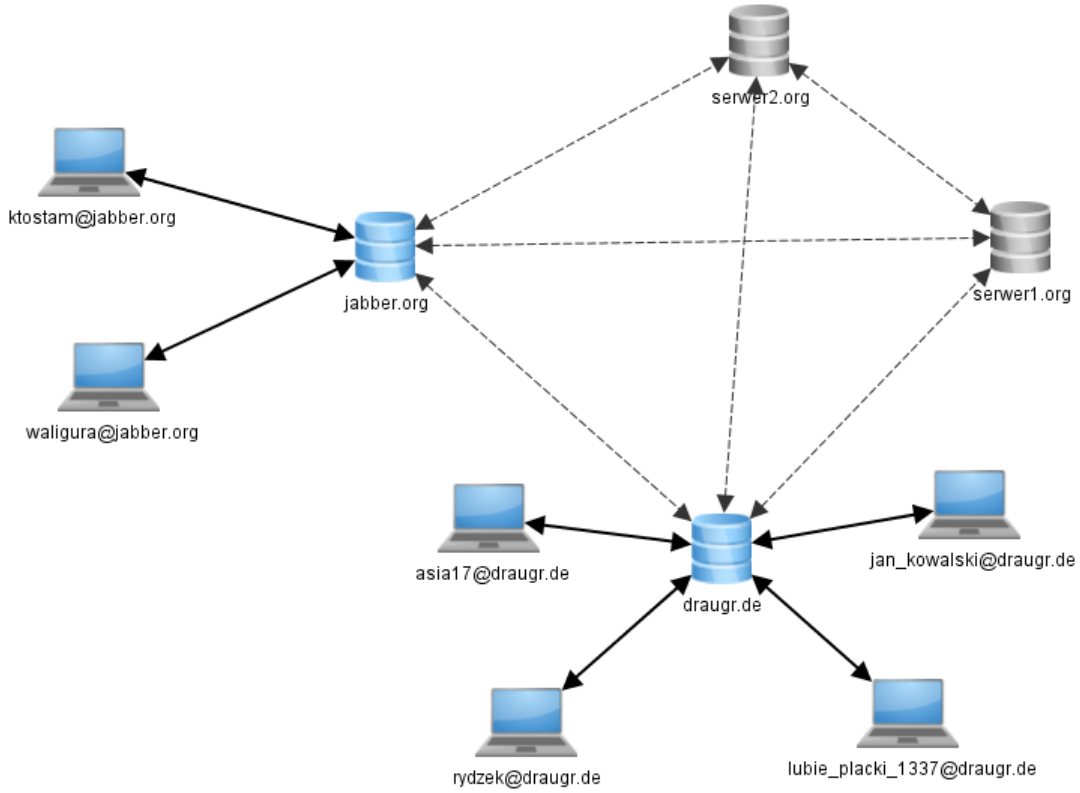
Na wybór protokołu XMPP do komunikacji pomiędzy programistami zdecydowano się głównie ze względu na jego otwartość, przystosowanie do rozszerzalności (np. o metadane informujące o edycji kodu) oraz użycie w wielu aplikacjach o podobnych możliwościach, służących do wspierania pracy grupowej zespołu programistów. Istotna okazała się również możliwość użycia istniejącej infrastruktury serwerów XMPP (dawniej Jabber), dzięki czemu wyeliminowana została konieczność implementacji programu serwera oraz dodatkowo zwiększona została niezawodność aplikacji, poprzez możliwość użycia do komunikacji dowolnego innego serwera w przypadku awarii aktualnie używanego.

Do komunikacji pomiędzy rozproszonymi klientami wybrany został serwer XMPP *draugr.de*, gdyż umożliwia on niezbędne w implementowanej aplikacji rozmowy grupowe (ang. MUC - Multi User Chat) oraz rejestrację nowych kont użytkowników bezpośrednio w aplikacji, bez konieczności odwiedzania jego strony internetowej. Poglądowy schemat architektury systemu przedstawia rysunek 1.

W aplikacji możliwe jest tworzenie nowych kont użytkowników tylko na serwerze *draugr.de*, jednak możliwe jest użycie istniejącego konta z dowolnego innego serwera. Komunikacja użytkowników posiadających konta na tym samym serwerze odbywa się za pośrednictwem tego serwera. W przypadku, gdy komunikacja następuje pomiędzy kontami na różnych serwerach, klient komunikatora wciąż wysyła swoje pakiety na swój serwer, zaś dopiero ten serwer przesyła go na serwer odbiorcy pakietu. Na rys. 1 liniami przerywanymi przedstawiono połączenia

[1] ang. Extensible Messaging and Presence Protocol - protokół bazujący na języku XML umożliwiający przesyłanie w czasie rzeczywistym wiadomości oraz statusu; protokół ma zastosowanie nie tylko w komunikatorach, ale również w innych systemach natychmiastowej wymiany informacji (źródło: Wikipedia)

[2] Strona domowa biblioteki: http://www.igniterealtime.org/projects/smack/



Rysunek 1: Architektura klient-serwer Komunikatora programistów

pomiędzy różnymi serwerami XMPP, nadzorującymi komunikację pomiędzy rozproszonymi po różnych serwerach klientami.

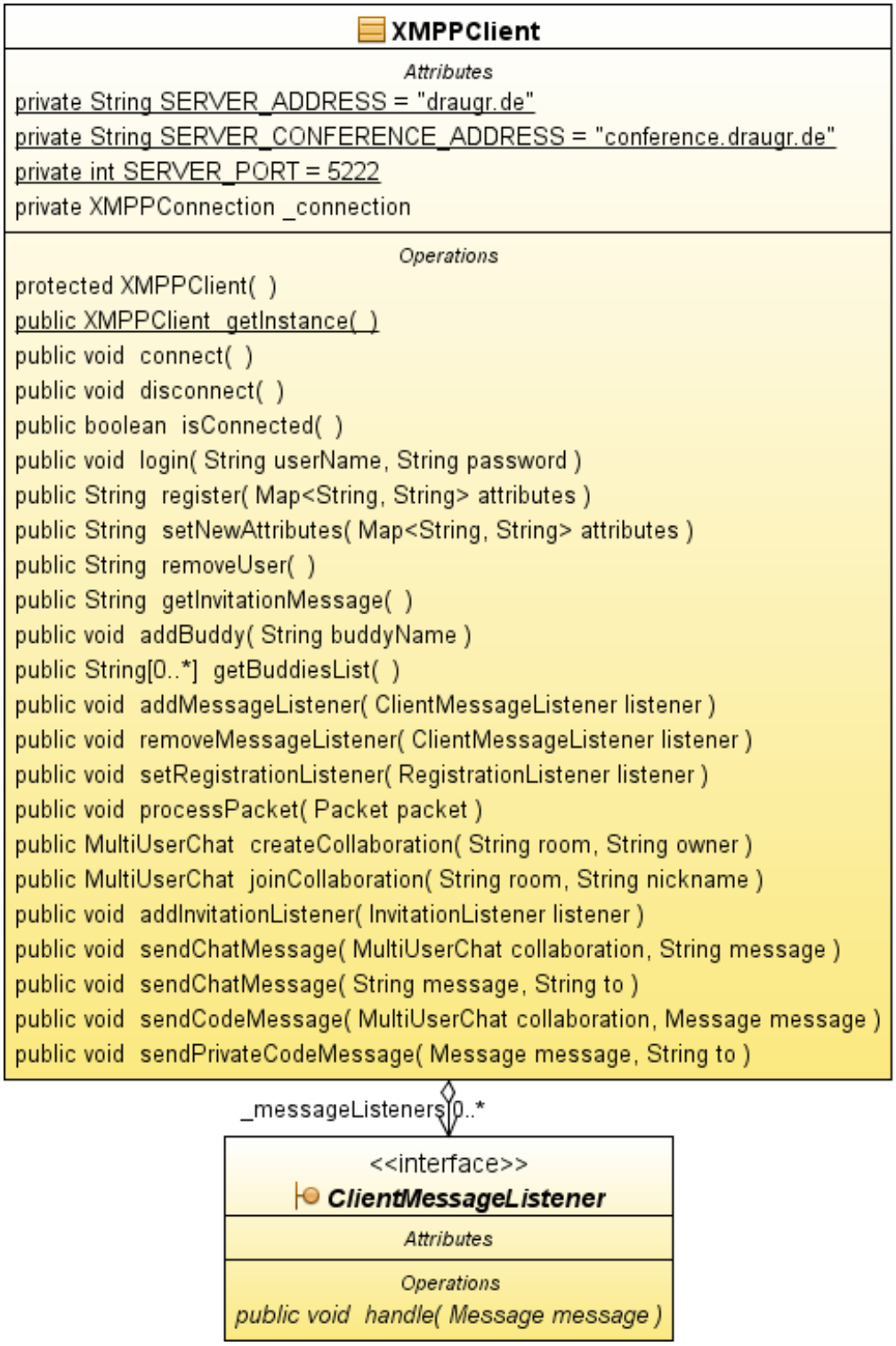
4 Klient XMPP

Klient XMPP zaimplementowany został w postaci klasy *XMPPClient*, której diagram przedstawiony jest na rys. 2. Klasa ta korzysta bezpośrednio z biblioteki Smack API, stanowiącej niskopoziomową warstwę sieciową komunikatora. *XMPPClient* implementuje wysokopoziome funkcjonalności pozwalające na wygodną abstrakcję pozostałej części aplikacji (graficznego interfejsu użytkownika) od szczegółów implementacyjnych komunikacji.

Z punktu widzenia komunikatora najbardziej istotne są następujące funkcje:

- *connect()*, *disconnect()* - połączenie/rozłączanie się z serwerem,
- *isConnected()* - sprawdzenie czy próba połączenia zakończyła się sukcesem,

- *login()* - zalogowanie się na serwer XMPP jako istniejący użytkownik,
- *register()* - zarejestrowanie nowego konta użytkownika na serwerze,
- *addMessageListener()*, *removeMessageListener()* - dodawanie i usuwanie nasłuchiwaczy wiadomości, patrz podrozdział 5
- *processPacket()* - obsługa odbieranych pakietów, zawierających różnego rodzaju wiadomości,
- *createCollaboration()*, *joinCollaboration()* - tworzenie nowych i dołączanie do istniejących kolaboracji,
- *sendChatMessage()*, *sendCodeMessage()* - wysyłanie wiadomości tekstowych i wiadomości informujących o modyfikacji kodu źródłowego.



Rysunek 2: Diagram UML klas XMPPClient i ClientMessageListener

5 Komunikacja pomiędzy klientami

5.1 Typy komunikatów

Celem realizacji przekazywania i interpretacji pakietów o różnych znaczeniach, zaimplementowana została rodzina klas wiadomości o różnych typach i różnych przeznaczeniach. Hierarchia tych klas przedstawiona jest na rys. 3.

Wszystkie klasy wiadomości dziedziczą po abstrakcyjnej bazie wiadomości - *Message*. Zawiera ona jedynie pole *UserID* z nazwą użytkownika-nadawcy wiadomości (np. *alicja@draugr.de*) oraz abstrakcyjną metodę *getType()*, która w klasach dziedziczących powinna być przeładowana celem zwracania swojego typu wiadomości jako jednej z wartości wyliczenia *MessageType*. Dzięki temu możliwe jest odczytanie typu odebranej wiadomości i odpowiednia jej obsługa w zależności od tego typu (patrz przykład w 5.2).

GroupMessage i *PrivateMessage* to wiadomości tekstowe, odpowiednio grupowe i prywatne. Zawierają pole Body z treścią wysłanej wiadomości.

Pozostałe wiadomości to klasy opisujące modyfikację plików źródłowych, nad którymi pracuje grupa programistów, w związku z czym zawierają dodatkowe metadane, np. *FileID* - identyfikator pliku, do którego odnosi się wiadomość.

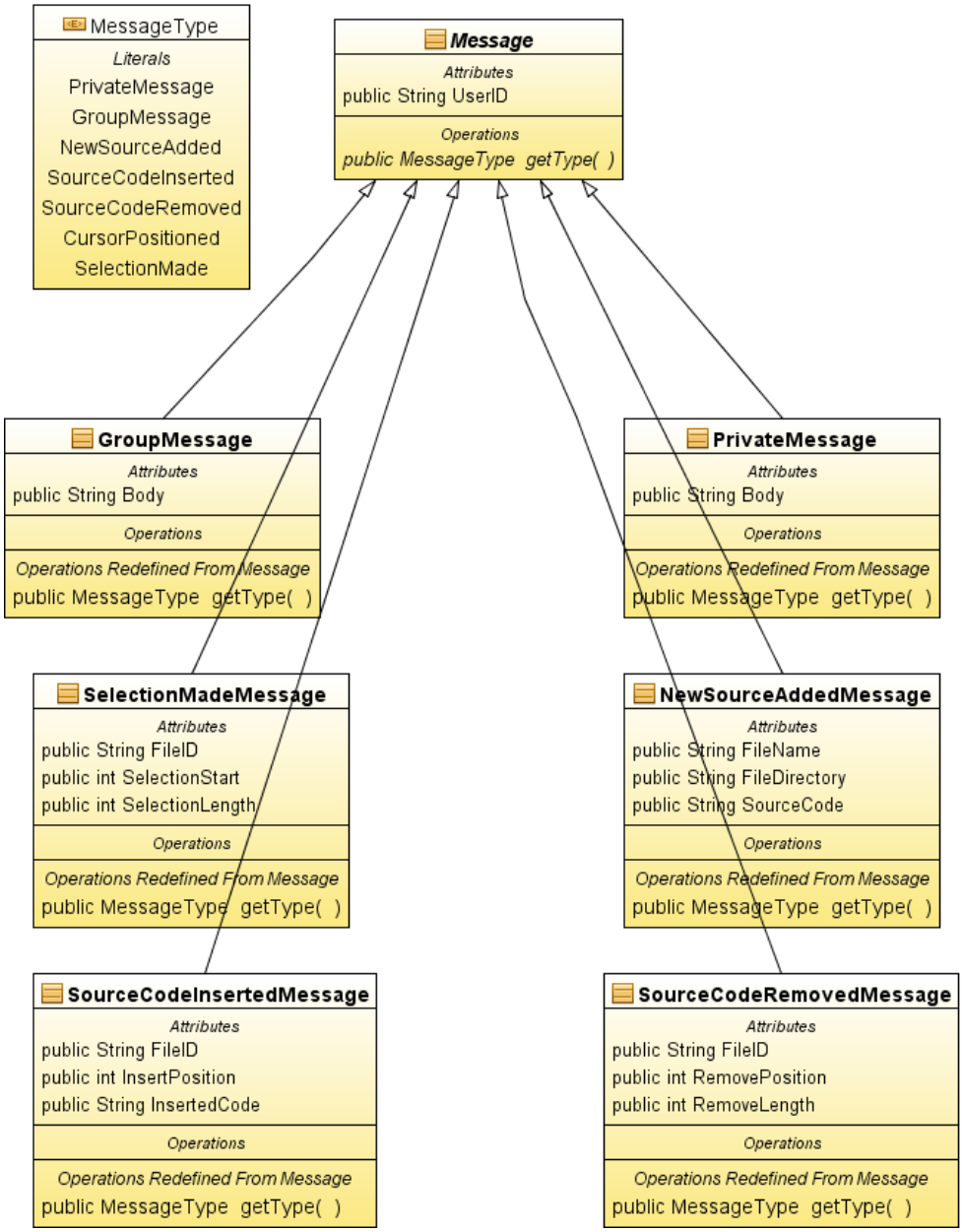
NewSourceAddedMessage informuje o dodaniu do edycji nowego pliku źródłowego o nazwie *FileName*, znajdującego się w katalogu *FileDirectory* i mającego treść *SourceCode*.

SelectionMadeMessage informuje o zaznaczeniu przez programistę fragmentu kodu źródłowego z pliku *FileID*, zaczynającym się na pozycji *SelectionStart* i mającym *SelectionLength* znaków.

SourceCodeInsertedMessage informuje o dodaniu nowego fragmentu kodu do pliku o id. *FileID*, zaczynającego się na pozycji *InsertPosition* i składającego się ze znaków *InsertedCode*.

SourceCodeRemovedMessage informuje o usunięciu fragmentu kodu z pliku o id. *FileID*, poczynając od pozycji *InsertPosition*, o długości *RemoveLength* znaków.

CursorPositioned informuje o ustawieniu przez programistę kursora na pozycji *CursorPosition* w pliku o id. *FileID*.



Rysunek 3: Diagram UML klas wiadomości

5.2 Przykład komunikacji

Rysunek 4 przedstawia przykładowy przepływ wiadomości pomiędzy użytkownikami komunikatora. W tym przypadku zaprezentowano przypadek, w którym jeden z programistów wprowadza zmiany do pliku źródłowego, nad którym pracuje grupa, w efekcie czego do reszty programistów (na rysynku zaprezentowano tylko jednego) trafia wiadomość pozwalającą na synchronizację stanu dokumentu.

Użytkownik inicjuje przepływ pakietów poprzez dokonanie modyfikacji w kodzie lub wysłanie wiadomości tekstowej, co na rys. 4 symbolizuje ciemnopomarańczowy romb *„Dodanie kodu"*. Za interakcję z programistą i tłumaczenie jego działań na wiadomości oraz przesyłanie ich za pomocą *XMPPClient* odpowiedzialny jest moduł GUI komunikatora.

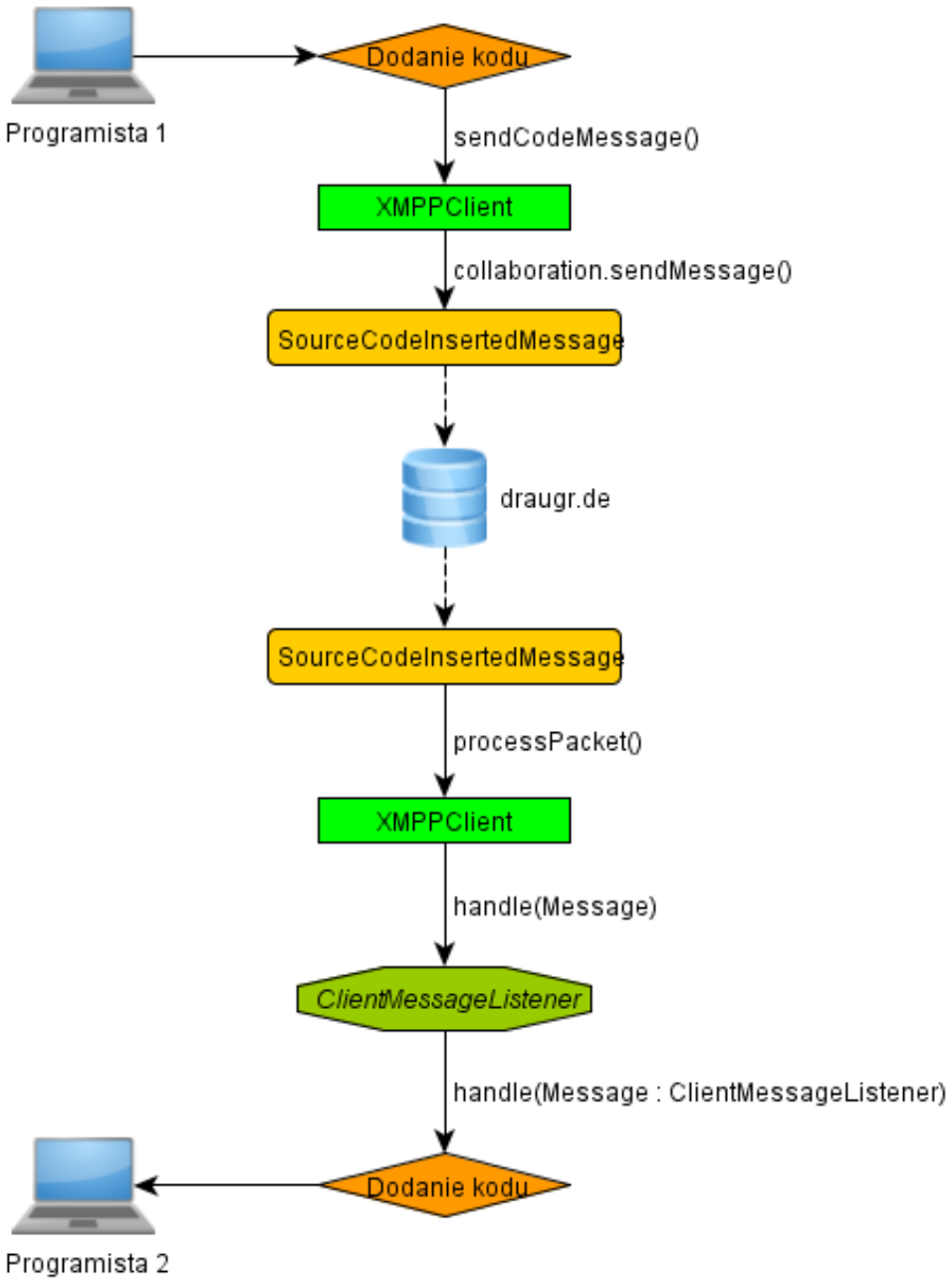
Tak sporządzona wiadomość trafia na serwer XMPP, skąd jest rozsyłana do odpowiednich użytkowników - w przypadku wiadomości prywatnej do pojedynczego adresata, zaś w przypadku wiadomości grupowej do wszystkich członków kolaboracji.

Instancje klienta XMPP u adresatów otrzymują wiadomości, które następnie są przetwarzane przez klasy implementujące interfejs *ClientMessageListener*. Pierwszym etapem takiego przetwarzania jest zrzutowanie otrzymanej wiadomości na odpowiedni typ, w zależności od wartości *message.getType()*. Drugi, ostatni etap to obsłużenie konkretnej wiadomości w zależności od jej typu. Przykładowe przetwarzanie wiadomości przedstawia listing 1.

Listing 1: Przykład obsługi otrzymywanych wiadomości po stronie klienta XMPP

```
public TestMsgListener implements ClientMessageListener {

  public void handle(Message message) {
    switch (message.getType()) {
      case NewSourceAdded:
        NewSourceAddedMessage msg = (NewSourceAddedMessage)message;
        // obsluz wiadomosc msg typu NewSourceAddedMessage
        break;
      case SourceCodeInserted:
        SourceCodeInsertedMessage msg = (SourceCodeInsertedMessage)message;
        // obsluz wiadomosc msg typu SourceCodeInserted
        break;
      // ... obsluga innych typow wiadomosci ...
    }
  }
}
```



Rysunek 4: Przykład przepływu wiadomości pomiędzy użytkownikami komunikatora