Opasiak Krzysztof – program serwer i diagramy
Markiewicz Mikołaj – część shared i testy
Klara Mateusz - dokumentacja
Sosnowski Jacek – komunikacja i biblioteka klienta

UXP1A

Dokumentacja końcowa

1) Treść zadania:

Napisać wieloprocesowy system realizujący komunikację w języku komunikacyjnym Linda przy wykorzystaniu potoków nazwanych i centralnego procesu koordynującego.

2) Interpretacja treści zadania i założenia:

Zadanie polega na dostarczeniu biblioteki statycznej umożliwiającej procesom komunikacje zgodną z językiem Linda:

a) Procesy powinny komunikować się poprzez wspólną przestrzeń krotek, którą należy zapewnić.
b) Krotka jest to tabela o dowolnej długości, która może składać się z danych typu string, int oraz float
c) Należy dostarczyć funkcję, która umożliwi umieszczenie danej krotki w przestrzeni krotek.
d) Należy dostarczyć funkcję, która pobierze i w sposób atomowy usunie z przestrzeni, krotkę zgodną z zadanym wzorcem.
e) Wzorzec krotki jest to ciąg znaków, który specyfikuje typu danych poszczególnych pól krotki i może jednocześnie zawierać również następujące warunki logiczne dla danego pola ==, <, <=, >, >=. Wzorzec przyjmuje następujący format <typ danej>:<operator><wartość> lub w przypadku gdy wartość może być dowolna <typ danej>:*
f) Należy dostarczyć funkcję, która umożliwi analogiczne pobranie krotki, jednak bez usuwania jej z przestrzeni.
g) Próba pobrania krotki nie pasującej do wzorca powinna zakończyć się zawieszeniem procesu do czasu pojawienia się pasującej krotki, lub przekroczenia maksymalnego czasu oczekiwania.

Jako realizację projektu rozumie się dostarczenie:
- biblioteki udostępniającej API do umieszczania i pobierania krotek.
- procesu sterującego – demona, który będzie przechowywał krotki
- zestawu testów
- dokumentacji końcowej.

Założenia:
- maksymalny rozmiar krotki jest określony statycznie i jest on tak dobrany aby zapis do potoku był atomowy.
- dla danej typu float nie ma warunku ==
- dla stringów operacje logiczne rozumiane są jako porównanie leksykograficzne
- w systemie, w którym uruchamiany będzie program istnieje katalog \tmp\linda, a wszystkie procesy korzystające z biblioteki mają prawo zapisu i odczytu do i z tego katalogu
- biblioteka nie implementuje wielowątkowości i wszystkie funkcje należy zewnętrznie synchronizować
- timeout jest to czas liczony od otrzymania przez daemona żądania wydania krotki, jeśli przed jego upływem nie pojawi się krotka spełniająca zadany warunek uznaje się wtedy, że klient powinien zostać o tym poinformowany

3) Opis funkcjonalności oraz API

Biblioteka została wykonana przy użyciu metodologii obiektowej i dostarcza następujące API:

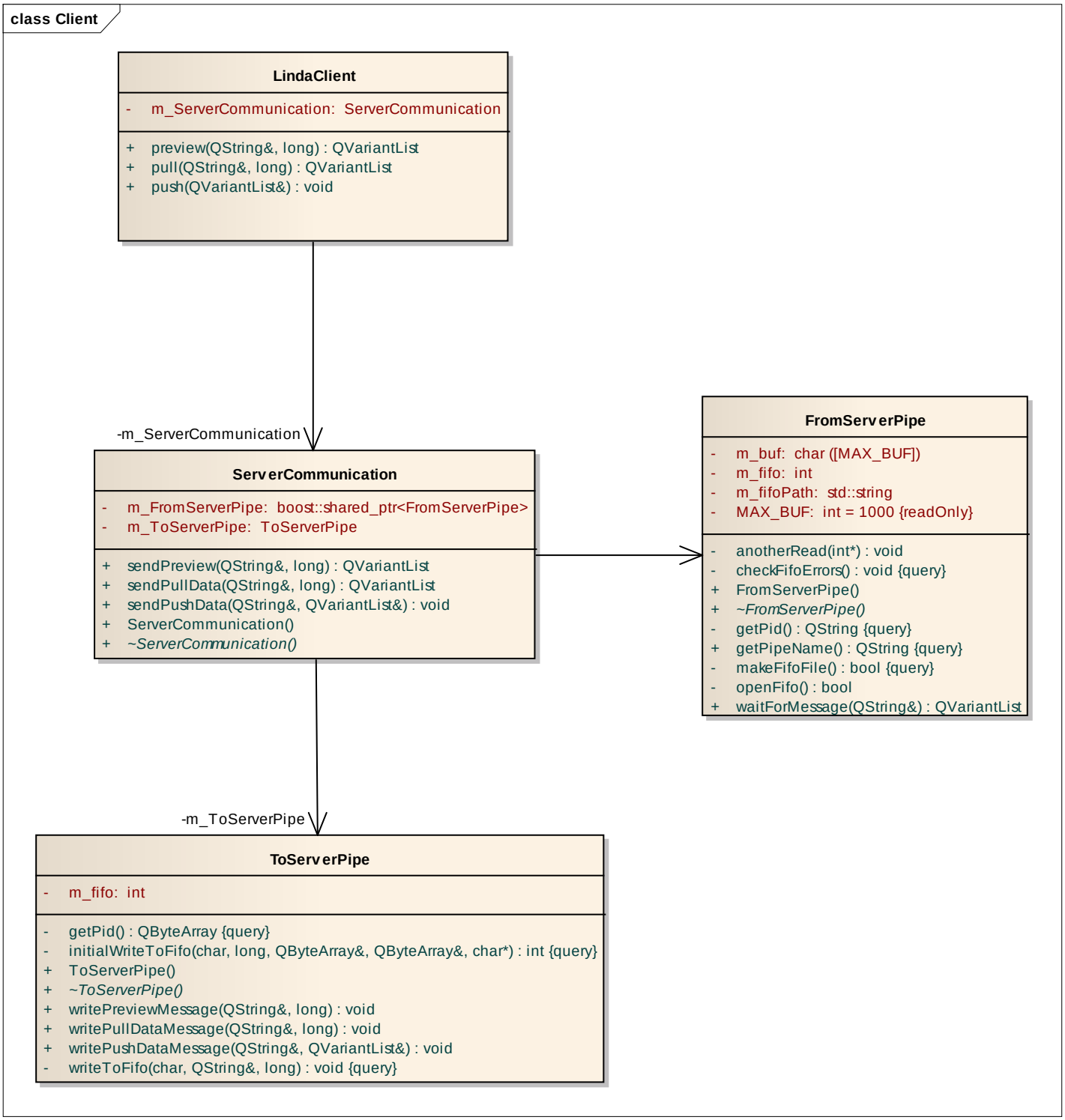
```
classLindaClient
{
public:
        //Umieszcza krotkę w przestrzeni
        static void push( QVariantList&record );

        //Pobiera krotkę I usuwa ja z przestrzeni
        static QVariantList pull( QString&format, long timeout );

        //Pobiera krotkę z przestrzeni ale jej nie usuwa
        static QVariantList preview(QString&format, long timeout );
};
```

Funkcja push umieszcza krotkę w przestrzeni krotek, funkcja blokująca do czasu zapisania krotki do potoku.
Funkcje pull oraz preview pobierają krotkę z przestrzeni i są blokujące do czasu dostarczenia do procesu zadanej krotki z serwera, lub upłynięcia timeoutu. Timeout podawany jest w milisekundach i jest to czas po jakim klient zostanie odblokowany jeśli krotka nie pojawi się w programie koordynującym. Podanie wartości timeout równej -1 oznacza czekanie w nieskończoność.Ponad to operacja pull atomowo usuwa pobieraną krotkę z przestrzeni krotek.

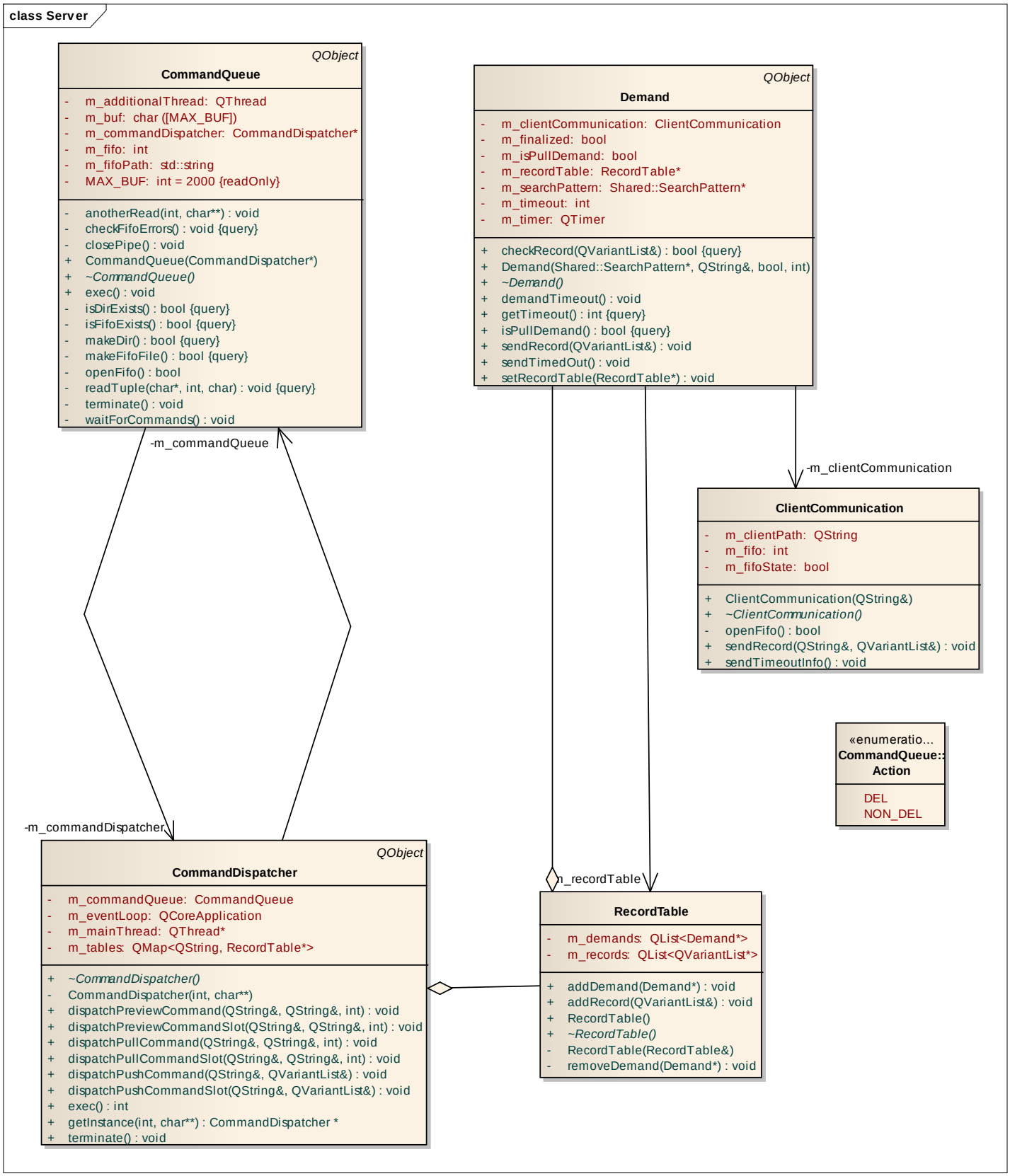
4) Podział na moduły

System składa się z następujących modułów:

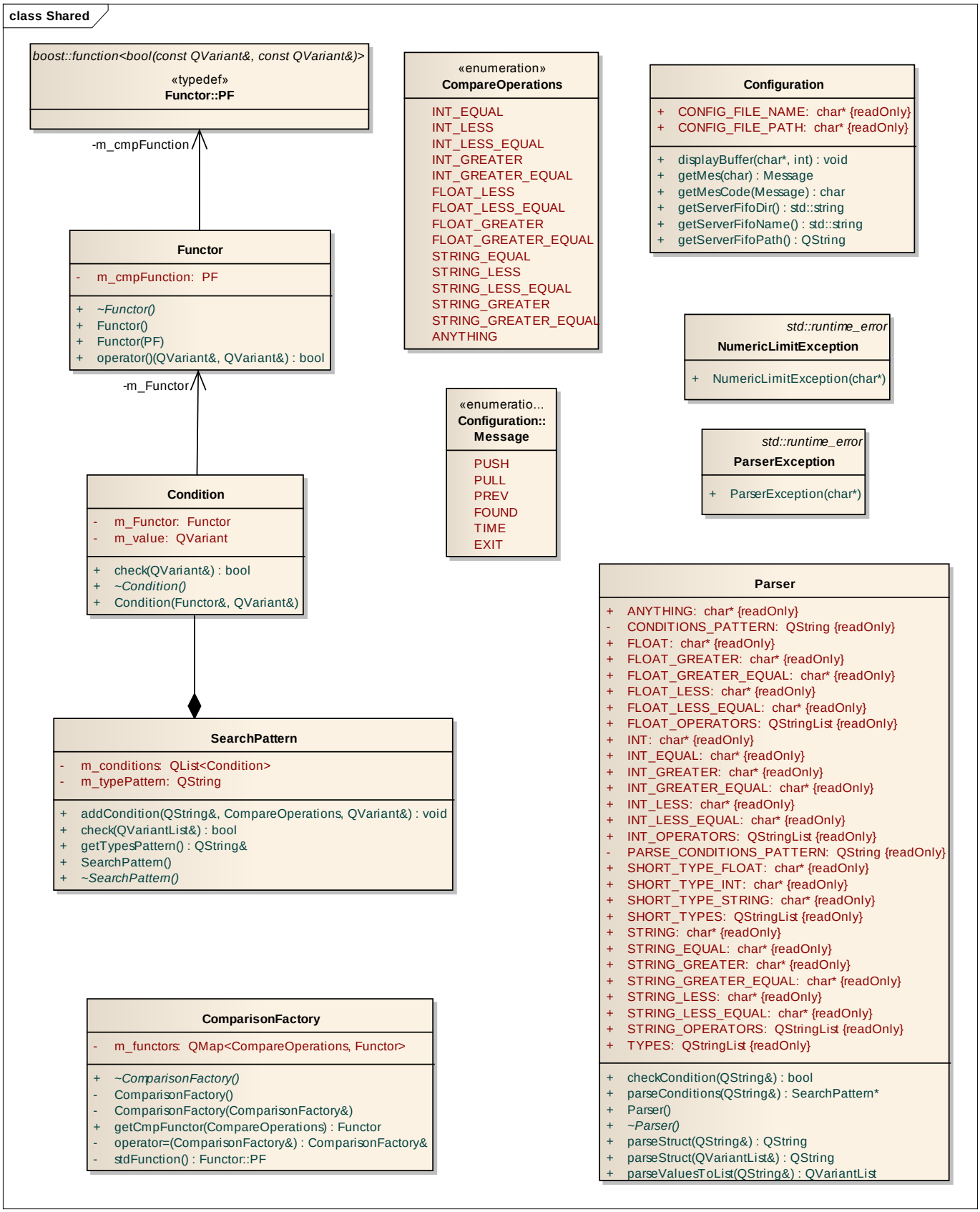
- moduł klienta – biblioteka linkowana statycznie, implementująca zaproponowane API. Komunikacja odbywa się poprzez klasę LindaClient posiadająca metody statyczne.



- moduł serwera – program uruchomiony w tle, który sprawuje rolę procesu sterującego, tj. obsługuje wszystkie żądania przesłane przez klientów oraz stanowi magazyn dla krotek.



-moduł współdzielony – statycznie linkowana biblioteka implementująca elementy programu wykorzystywane zarówno w module klienta jak i serwera.



- moduł testujący – stanowi zbiór programów i skryptów przeznaczonych do testowania. Można tu wyróżnić testy jednostkowe oraz testowy program klienta, który wykonuje działania klienta w zależności od danych podanych mu na standardowe wejście. Ponad to w celu automatyzacji testów wykonano skrypty i odpowiednie pliki wejściowe dla programów testowych.

5) Opis komunikacji pomiędzy modułami.

W systemie występują dwie strony komunikacji – klient czyli dowolny proces używający implementowanej biblioteki, oraz serwer czyli daemon dostarczony razem z biblioteką, który przechowuje krotki. Komunikacja odbywa się poprzez potoki nazwane i możemy wyróżnić w niej 3 przypadki tj. trzy rodzaje żądań, które mogą być kierowane przez klienta do serwera:

a) Umieszczenie krotki w przestrzeni

Klient zapisuje do potoku serwera dane krotki, która ma zostać zapisana tj. jej format oraz wartości danych. Operacja jest blokująca do czasu, gdy w potoku serwera nie będzie wystarczającej ilości miejsca do zapisania tej komendy. Serwer czyta po kolei komendy z swojego potoku, po czym je wykonuje. Wykonanie tej komendy przez serwer ma dwa warianty:

- jeśli nikt nie czeka na krotkę o takiej zawartości to jest ona dodawana do wielozbioru krotek przechowywanych przez serwer.
- jeśli istnieje proces, który zgłosił żądanie pobrania krotki o takich danych to krotka ta jest przekazywana do tego procesu i zależnie od rodzaju żądania jest ona zapisywana do pamięci lub nie.

b) Pobranie i usunięcie krotki z przestrzeni

Klient zapisuje do potoku serwera wzorzec krotki, którą chce pobrać oraz ścieżkę do pliku jego potoku (tworzenie potoku w chwili pierwszej potrzeby jego użycia) oraz maksymalny czas oczekiwania na krotkę. Serwer po odczytaniu takiego żądania sprawdza dostępność krotki:

- jeśli krotka jest dostępna to jest ona usuwana z przestrzeni, potok klienta jest otwierany, krotka zapisywana, a potok zamykany, jeśli wystąpi błąd, należy zaprzestać zapisu, zamknąć potok, a krotka powinna dalej być przechowywana w pamięci
- jeśli krotka nie jest dostępna, uruchamiany jest timer. Jeśli krotka pojawi się przed upływem czasu, jest ona wysyłana do klienta jak wyżej, jeśli czas upłynie to do klienta wysyłana jest wiadomość o przekroczeniu limitu czasu.

c) Pobranie krotki bez usunięcia jej z przestrzeni – czynność analogiczna do poprzedniej ale krotka nie jest usuwana z przestrzeni krotek.

Ponad to w przypadku kończenia procesu pełniącego role serwera krotek wpisuje on do swojego potoku komunikacyjnego wiadomość, która informuje wątek czytający z potoku o konieczności zakończenia.

Zatem w całym systemie możemy wyróżnićnastępujące rodzaje komunikatów o przedstawionej poniżej budowie:

Komunikacja Klient-Serwer:

a) umieść krotkę
   - nagłówek informujący o typie komunikatu
   - długość wiadomości
   - ciąg znaków mówiący o typach pól krotki
   - pola krotki

b) pobierz krotkę i usuń z przestrzeni
   - nagłówek informujący o typie komunikatu
   - długość wiadomości
   - pid procesu klienta
   - wyrażenie opisujące warunki, jakie musi spełniać krotka
   - maksymalny czas oczekiwania

c) pobierz krotkę
   - nagłówek informujący o typie komunikatu
   - długość
   - pid procesu klienta
   - wyrażenie opisujące warunki, jakie musi spełniać krotka
   - maksymalny czas oczekiwania

Komunikacja Serwer-Klient:

d) krotka znaleziona
   - nagłówek informujący o typie komunikatu
   - długość wiadomości
   - ciąg znaków mówiący o typach pól krotki
   - pola krotki

e) czas upłynął
   - nagłówek informujący o typie komunikatu

Komunikacja Serwer-Serwer

f) zakończ pracę (wysyłane po otrzymaniu sygnału SIGINT do Fifoserwara)
- nagłówek informujący o typie komunikatu

Wszystkie dane są przesyłane w formacie binarnym – w ramach jednej platformy systemowej długości zmiennych (np. int) są jednoznacznie określone.

Kody komunikatów wykorzystują protokół znakowy – reprezentacja za pomocą liter. Mieszczą się na jednym bajcie:

a – PUSH - umieść krotkę
b – PULL - pobierz krotkę i usuń z przestrzeni
c – PREV - pobierz krotkę (bez usuwania)
d – FOUND - krotka znaleziona
e – TIME - czas upłynął (timeout)
f – EXIT – zakończ działanie serwera

Wszystkie komunikaty za wyjątkiem timeout zawierają pole specyfikujące długość wiadomości. Powodem przesyłania takiej danej jest możliwość używania buforów odbiorczych o dowolnej długości. W szczególności mniejszej niż rozmiar struktury potoku. Proces odbiorczy (czy to po stronie klienta czy serwera) musi być w stanie jednoznacznie określić czy odebrał już całą wiadomość. Jest to najistotniejsze w przypadku przesyłania wraz z wiadomością danych o dowolnej długości.

Komunikaty pull oraz prev zawierają pole z numerem pid klienta oczekującego na krotkę. Powodem wystąpienia tego pola jest fakt iż plik fifo klienta przeznaczony na odbiór danych od serwera ma właśnie taką nazwę jak pid tego procesu. Wszystkie takie pliki znajdują się w tym samym katalogu, w którym przechowywany jest plik fifo serwera. Dlatego to rozwiązanie zapewnia jednoznaczną identyfikację tych plików bez konieczności przesyłania (czasami długich) ścieżek dostępowych do nich.

Ciąg znaków mówiący o typach pól krotki (pattern) występujący w komunikacie push jest skróconym wzorcem wysyłanej krotki. Dzięki niemu może być ona klasyfikowana i przydzielona do pewnej podgrupy krotek. Pattern jest budowany przez bibliotekę Linda na podstawie podanej przez proces klienta listy danych. Jeśli lista zawiera niedozwolone zmienne komunikat zostanie odrzucony.

Wyrażenie opisujące warunki, jakie musi spełniać krotka (condition) jest dłuższym ciągiem znaków niż pattern. Zawiera wyrażenia warunkujące wybór krotki (np. >, <).

Zarówno pattern jak i condition są przed wysłaniem do serwera sprawdzone składniowo i semantycznie. Dlatego wszelkie błędy w nich występujące spowodują odrzucenie komunikatu jeszcze przed wysłaniem do kolejki serwera. Te pola są przesyłane za pomocą protokołu znakowego – kodowanie ASCII.

Maksymalny czas oczekiwania (timeout) domyślnie nieskończoność. Wartość w tym polu oznacza czas, jaki będzie oczekiwał serwer od rozpoczęcia obsługi komunikatu, do

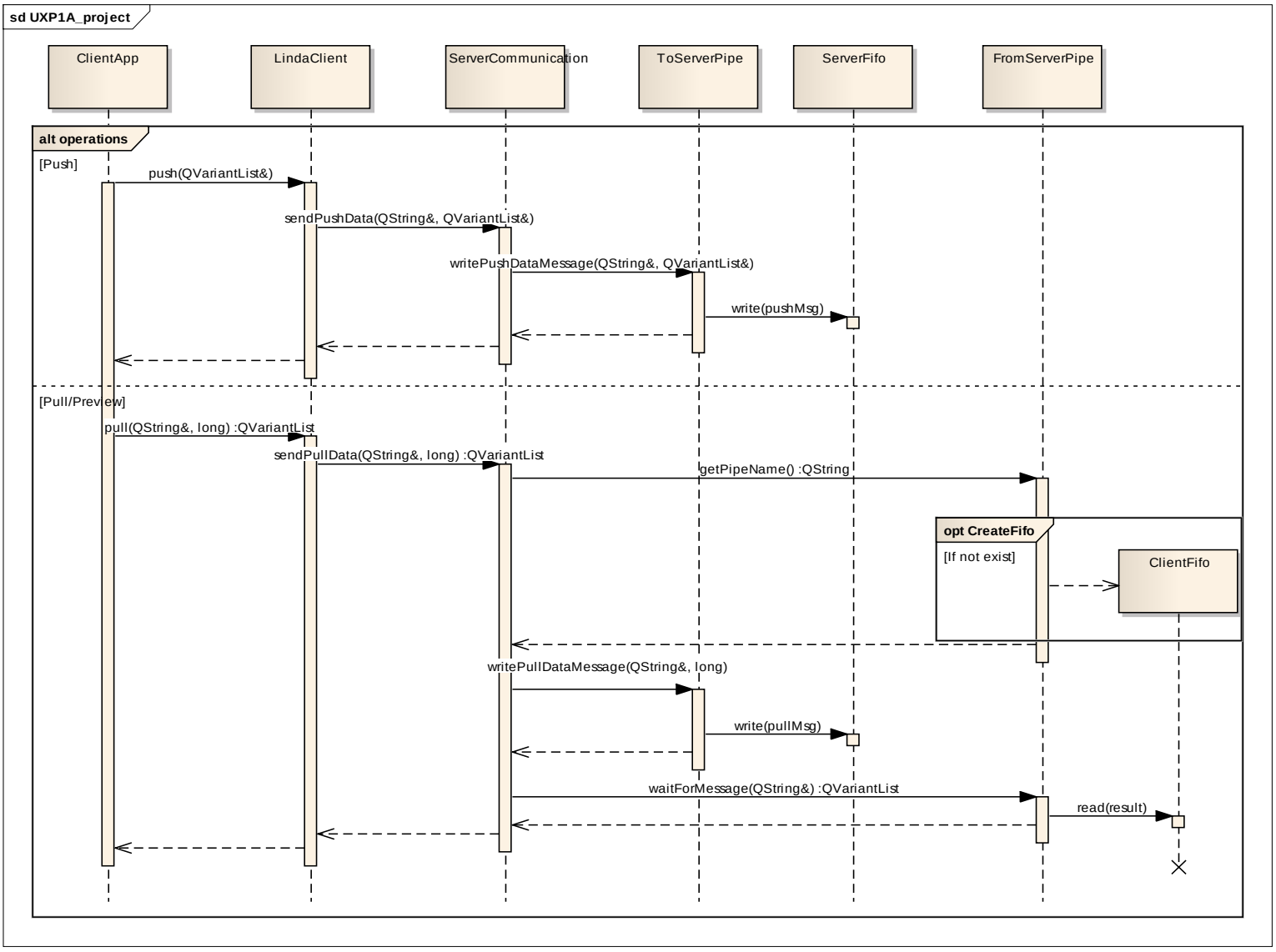
momentu, kiedy wyśle do klienta wiadomość timeout, jeśli nie ma zadanej krotki. Jeśli to pole ma wartość 0 to wiadomość oznacza: odpytaj serwer i jeśli w chwili obsługi mojej wiadomości nie ma krotki, to nie czekaj za nią w ogóle.

6) Opis zaimplementowanego rozwiązania.

Szczegółowy opis poszczególnych funkcji znajduje się w dokumentacji wygenerowanej przy pomocy programu Doxygen, natomiast w niniejszym punkcie przedstawiono jedynie zarys ogólny implementacji i ważniejszych funkcji

a) Moduł klienta

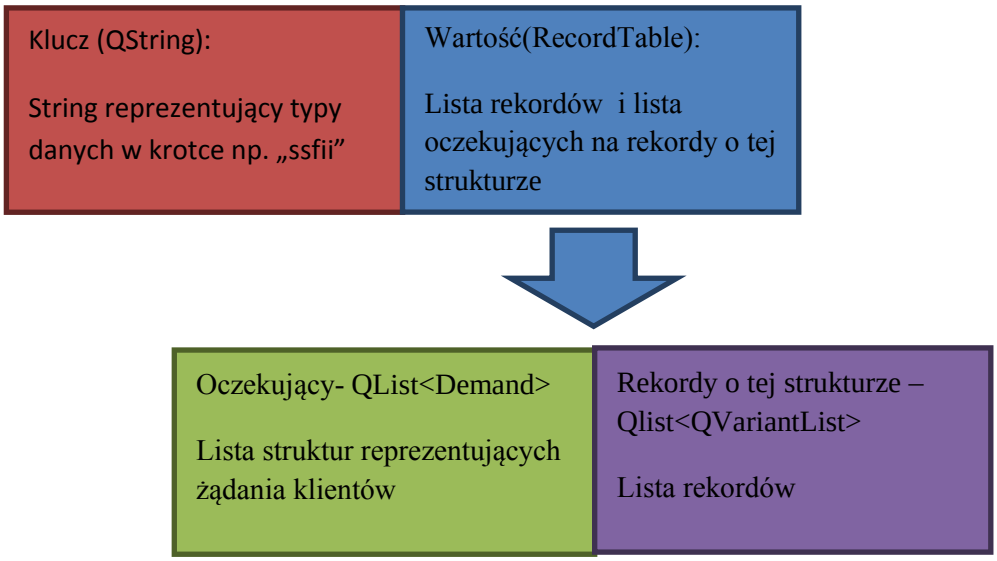
Moduł ten dostarcza interfejs do zapisu i odczytu krotek. Dane dostarczane przez klienta są sprawdzane pod kątem poprawności, a następnie formowana jest wiadomość, która jest wpisywana do potoku serwera. Biblioteka może zgłosić wyjątek spowodowany niepoprawnym formatem danych lub brakiem uruchomionego procesu serwera. W przypadku wiadomości pull i preview klient po wysłaniu komunikatu do serwera oczekuje na utworzonym przez siebie potoku na odpowiedz od serwera. Uproszczony diagram sekwencji dla klienta wygląda następująco:



b) Moduł serwera

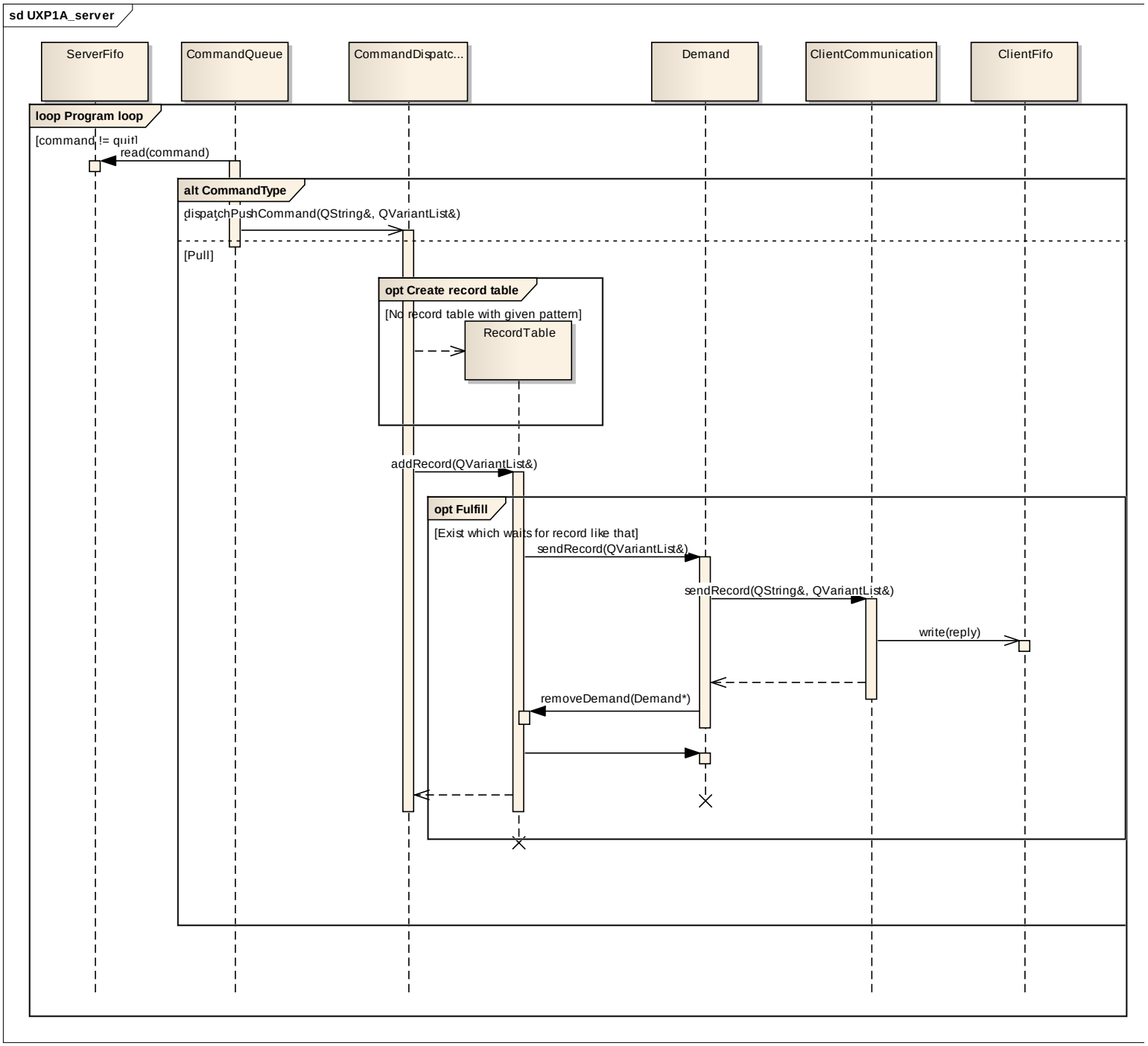
Moduł jest to program składający się z dwóch wątków.
-Jeden wątek (wątek w którym działa CommandQueue) jest to wątek pomocniczy, który odczytuje z potoku komendy od klientów i przekazaniu jej do wykonania, w związku z czym przez większość czasu wątek ten jest zawieszony na czytaniu z potoku.
-Drugi wątek natomiast zajmuje się faktycznym wykonywaniem zadań zleconych przez klientów oraz pilnuje aby klienci, którym upłynął czas oczekiwania zostali wznowieni i o tym powiadomieni.
Komendy wczytane przez CommandQueue są gromadzone w kolejce zdarzeń zapewnionej przez bibliotekę Qt. W momencie wywołania metod CommandDispatcher'a komendy te są gromadzone w kolejce zdarzeń, a ich wykonanie przenoszone jest do drugiego wątku.

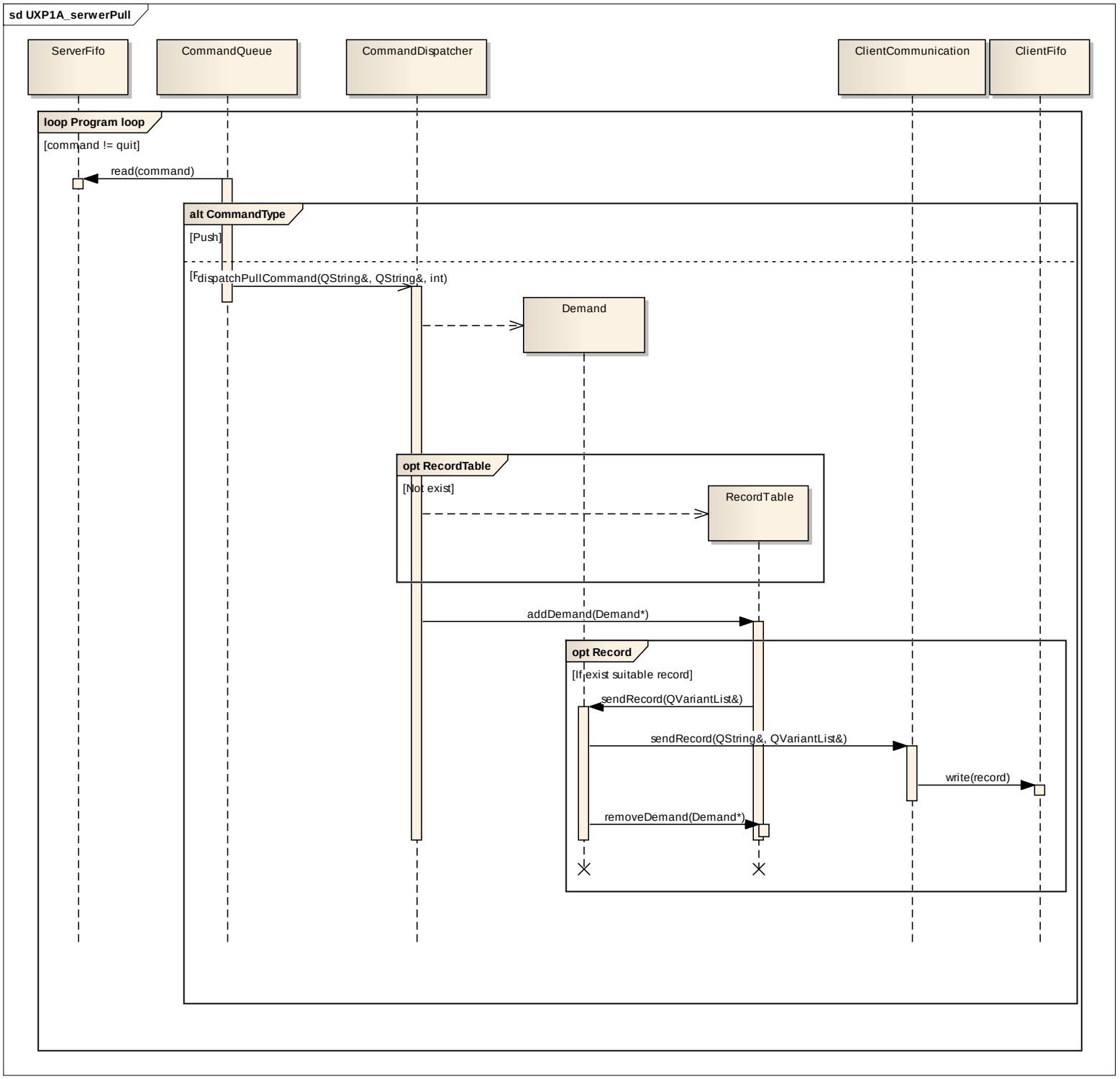
Klasa CommandDispatcher stanowi szkielet całego programu serwera. Przechowuje ona strukturę danych w której przechowywane są krotki oraz żądania klientów. Struktura ta ma budowę słownika, gdzie pojedynczym elementem jest:



Każda krotka trafia do odpowiadającej jej wzorcowi tabeli. Przy dodawaniu krotki, sprawdzane jest czy spełnia ona warunki podane przez któregokolwiek z oczekujących na krotkę o takiej strukturze danych. Jeśli tak to jest ona mu przesyłana i w zależności od rodzaju żądania klienta dodawana do tabeli lub nie.
W przypadku dodania żądania na krotkę o zadanych warunkach wyszukiwana jest pozycja według podanego wzorca typów, po czym sprawdzane jest czy w liście rekordów znajduje się jakiś, który spełnia warunki wyspecyfikowane przez klienta, jeśli tak to wysyłany jest rekord, jeśli nie to żądanie dodawane jest do listy żądań oczekujących i uruchamiany jest tiemer, jeśli czas wyspecyfikowany przez klienta dobiegnie końca to żądanie usuwane jest z listy, a do potoku klienta zostaje wpisana specjalna wiadomość informująca o upłynięciu czasu.

Uroszczone działanie serwera przedstawiają poniższe diagramy sekwencji:



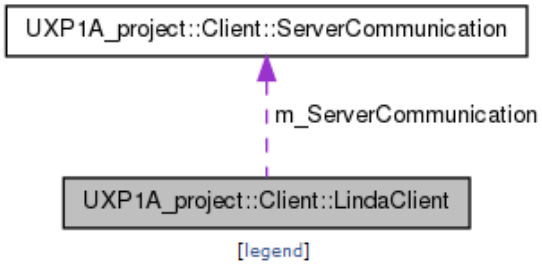

sd UXP1A_serwerPull
ServerFifo
CommandQueue
CommandDispatcher
ClientCommunication
ClientFifo
loop Program loop
[command != quit]
read(command)
alt CommandType
[Push]
[Pdispatch
dispatchPullCommand(QString&, QString&, int)
Demand
opt RecordTable
[Not exist]
RecordTable
addDemand(Demand*)
opt Record
[If exist suitable record]
sendRecord(QVariantList&)
sendRecord(QString&, QVariantList&)
write(record)
removeDemand(Demand*)

7) Szczegółowy opis interfejsu użytkownika

Szczegółowy opis interfejsu został wygenerowany z użyciem programu doxygen. Fragment jego dotyczący został przedstawiony poniżej:

`#include <LindaClient.h>`

Collaboration diagram for UXP1A_project::Client::LindaClient:



Public Member Functions

	LindaClient () Constructor.
virtual	**~LindaClient** () Destructor.

Static Public Member Functions

static QVariantList	**preview** (const QString &pattern, long timeout=-1) Execute preview command in Linda **Client** application.
static QVariantList	**pull** (const QString &pattern, long timeout=-1) Execute pull command in Linda **Client** application.
static void	**push** (const QVariantList &record) Execute push command in Linda **Client** application.

Static Private Attributes

static **ServerCommunication**	**m_ServerCommunication** Object used to communicate with Linda **Server**.

Constructor & Destructor Documentation

UXP1A_project::Client::LindaClient::LindaClient ( )

Constructor.

You may create object of this class for more convenient usage of static methods of this class.

Definition at line **25** of file **LindaClient.cpp**.

UXP1A_project::Client::LindaClient::~LindaClient ( ) [virtual]

Destructor.

Definition at line **30** of file **LindaClient.cpp**.

QVariantList UXP1A_project::Client::LindaClient::preview (const QString & pattern, long timeout = -1) [static]

Execute preview command in Linda **Client** application.

This method doesn't cause tuple deletion from server.

Parameters:

[in] **pattern** which returned tuple should fulfill.

[in] **timeout** is max time in milliseconds which server will wait for tuple since begin of serving this command, before it sends timeout message

Returns:

QVariantList which contains required tuple or empty QVariantList which means that there is no such tuple and timeout occurs.

This method send preview request to server and wait for answer (until timeout). First calling of this method opens client's FIFO which is use to receive tuples from server in this method and **pull()** also. FIFO is opening only one per client - only if it is necessary (after first use of **preview()** or **pull()** method).

Exceptions:

Shared::ParserException	if pattern is incorrect
Shared::NumericLimitException	if some numeric values exceeds integer or float limits
ServerFifoException	if function was unable to open server's FIFO
ClientFifoException	if function was unable to create or open client's FIFO

Definition at line **35** of file **LindaClient.cpp**.

References **UXP1A_project::Shared::Parser::checkCondition()**, **m_ServerCommunication**, and **UXP1A_project::Client::ServerCommunication::sendPreview()**.

Referenced by **UXP1A_project::Tests::ManualLinda::previewTuple()**, and **testServerFifo()**.

void UXP1A_project::Client::LindaClient::push (const QVariantList & record) [static]

Execute push command in Linda **Client** application.

This method will send record to the Linda **Server** via its FIFO.

Parameters:

[in] **record** to be sent. This record will be send and store in **Server** only if types of each QVariant will be one from the list below: string, int or float

To push record **Client** application should prepare tuple - QList of QVariant (QVariantList typedef) and call this method which should return immediately after write this tuple to **Server** FIFO.

Exceptions:

Shared::ParserException	if record consist of from types diferen than integer, float, string
ServerFifoException	if function was unable to open server's FIFO
ClientFifoException	if function was unable to create or open client's FIFO

Definition at line **63** of file **LindaClient.cpp**.

References **m_ServerCommunication**, and **UXP1A_project::Client::ServerCommunication::sendPushData()**.

Referenced by **UXP1A_project::Tests::ManualLinda::pushTuple()**, and **testServerFifo()**.

QVariantList UXP1A_project::Client::LindaClient::pull (const QString & pattern, long timeout = -1) [static]

Execute pull command in Linda **Client** application.

This method DELETES tuple from server!

Parameters:

[in] **pattern** which returned tuple should fulfill. This is not parsed pattern.

[in] **timeout** is max time in milliseconds which server will wait for tuple since begin of serving this command, before it sends timeout message

Returns:

QVariantList which contains required tuple or empty QVariantList which means that there is no such tuple and timeout occurs.

This method send pull request to server and wait for answer (until timeout). First calling of this method opens client's FIFO which is use to receive tuples from server in this method and **preview()** also. FIFO is opening only one per client - only if it is necessary (after first use of **preview()** or **pull()** method).

Note:

Returned tuple won't be available on the server any more.

Exceptions:

Shared::ParserException	if pattern is incorrect
Shared::NumericLimitException	if some numeric values exceeds integer or float limits
ServerFifoException	if function was unable to open server's FIFO
ClientFifoException	if function was unable to create or open client's FIFO

Definition at line **49** of file **LindaClient.cpp**.

References **UXP1A_project::Shared::Parser::checkCondition()**, **m_ServerCommunication**, and **UXP1A_project::Client::ServerCommunication::sendPullData()**.

Referenced by **UXP1A_project::Tests::ManualLinda::pullTuple()**, and **testServerFifo()**.

8) Wykorzystane narzędzia

W trakcie prac nad projektem wykorzystano następujące narzędzia:

- CMake – automatyczne budowanie
- Doxygen – generacja dokumentacji
- Git – repozytorium
- Eclipse – IDE wykorzystywane do implementacji projektu
- Enterprise Architect – środowisko do projektowania projektów software
- Biblioteka Qt
- Biblioteka Boost
- Google Test Framework – testy jednostkowe
- potoki nazwane – komunikacja miedzy procoesowa
- mechanizm sygnałów – kończenie pracy serwera

9) Metodyka testowania

Testy do wykonanego projektu zostały podzielone na kilka etapów:

- pierwszy etap stanowią testy jednostkowe, które sprawdzają podstawową funkcjonalność wybranych klas. W testach tych uczestniczą przede wszystkim klasy odpowiedzialne za sprawdzania zgodności wzorca czy porównywanie warunków wyspecyfikowanych przez użytkownika z danymi faktycznymi.

- drugi etap stanowi program, z prostym interfejsem tekstowym, który wykorzystuje zaimplementowana bibliotekę. Umożliwia on dodanie lub pobranie krotki o zadanym wzorcu, przez co pozwala na przetestowanie w prosty sposób wszystkich scenariuszy, jakie uda się wymyślić.

-trzeci etap są to skrypty wykorzystujące program do ręcznej analizy poprawności z etapu drugiego. Dzięki ich użyciu otrzymujemy powtarzalne testy scenariuszowe. Program testowy został przygotowany do podawania na wyjście informacji mających na celu weryfikację poprawności działania poprzez analizę tego tzw. logu. Dla odpowiednich parametrów wejściowych program uruchamiany jest odpowiednio do użycia razem z napisanymi skryptami. Parametry programu (klasy *ManualLinda* stworzonej na potrzeby testowania) pozwalające na losowy test współbieżnych procesów:

testOutput	flaga blokująca komunikaty wychodzące na standardowe wyjście
testSleep	brak lub losowe opóźnienie w *ms* z podanego zakresu po każdej operacji
testTimeout	losowy *timeout* w *ms* z podanego zakresu przy operacji *pull*/*preview*
testAfterPulledSleep	losowe opóźnienie po sukcesie operacji *pull*

Program na standardowe wyjście podaje wartość zwróconą przez klasę testującą w postaci ciągu cyfr: 0 i 1.

1 oznacza że podana operacja zakończyła się sukcesem, np. krotka została wypchnięta (operacja *push*) z sukecem, lub krotka została pobrana (operacja *pull*/*preview*).
0 oznacza że podana operacja osiągnęła podany *timeout*, tj. nie została zrealizowana.

W trakcie działania programu w celu utworzenia logu zdarzeń, używając klasy degub'ującej biblioteki Qt, wypisywane są informacje o tym co dany proces (identyfikowany numerem PID) aktualnie robi. Są to i nformacje o tym iż wykonał operację *push* dla podanych danych, że oczekuje jakiś czas na kontynuację programu, bądź że wykonuje operację *pull*, poprzedzając informację o sukcesie lub osiągnięciu limitu czasu, podaje jej argumenty: *timeout*i wzorzec.

Plikiem konfiguracyjnym dany przypadek testowania programu jest sekwencja tekstowych komend używanych do obsługi programu testowego.

Scenariusze testowe zakładają podanie *timeout'u* o zadanym czasie. Do celów testowania niedopuszczalnym by było nieskończone czekanie w przypadku braku krotki.
Obecnie testowane są następujące scenariusze:

1. *1_test.sh*
   Proces wykonuje operację *push* z podaniem określonej sekwencji danych typu *int* na podstawie pliku konfiguracyjnego dane operacje.
   Uruchamiane zostają 3 współbieżnie działające procesy, których celem jest odbiór (operacja *pull*) tej samej sekwencji krotek które zostały umieszczone w przestrzeni przez pierwszy proces. Wszystkie trzy procesy dostają ten sam plik konfiguracyjny określający co ma zostać pobrane.
   Procesy uruchomione zostały z *timeout'*em równym *500ms* w oczekiwaniu na każdą z krotek. Po odebraniu krotki przez któryś z procesów, zostaje on uśpiony na losowy czas z przedziału od *0* do *1000ms*. Takie parametry dają ciekawe rezultaty konkurujących ze sobą procesów 'konsumenckich'.
   W wyniku uruchomienia testu dostajemy ciągi znaków generowane przez uruchomione procesy, np:
   ```
   Pushed: 11111
   Pulled: 01011
   Pulled: 10000
   Pulled: 00111
   ```
   Skrypt sprawdza operacją XOR wyniki procesów odbierających krotki. Wynikiem pozytywnym powinien być ciąg znaków składający się z samych 1 o długości pierwszego ciągu opisującego umieszczanie krotek przez pierwszy proces. Pokazuje to iż każda krotka została pobrana tylko jeden raz.
2. *2_test.sh*
   Test przebiega w podobny sposób do testu nr. 1, z tym że po umieszczeniu krotek w przestrzeni, kolejny proces wykonuje operację *preview* na każdej z nich. Następnie uruchamiane są procesy pobierające krotki oraz następuje weryfikacja wg opisu jak w

teście nr. 1. Test ma na celu pokazanie iż krotki 'przejrzane' operacją *preview* nie zostają usunięte z przestrzeni i są ciągle dostępne.

3. *3_test.sh*

   Proces wykonuje operację *push* z podaniem określonej sekwencji danych różnych typów złączonych w krotki. Uruchamiane zostają 3 współbieżne procesy, a każdy z nich otrzymuje odpowiedni plik konfiguracyjny z wzorcami krotek, sprawdzającymi jednocześnie działanie operatorów. Dane zostały tak skonstruowane, żeby procesy uruchomione jako 2. 3. 4. kolejno odbierały co trzecią krotkę, a na innych blokowały się w oczekiwaniu na *timeout*. Wyjątkiem jest ostatnia krotka którą otrzyma proces który pierwszy się po nią 'zgłosi'. Wynikiem pozytywnym tak samo jak w testach nr. 1 i 2, jest operacja XOR na wynikach. Oczekiwany wynik z dokładnością do ostatniej krotki jest następujący:

   ```
   Pushed: 1111111111
   Pulled: 1001001001
   Pulled: 0100100100
   Pulled: 0010010010
   ```

4. *4_test.sh*

   Proces wykonuje operację *push* z podaniem konkretnych pojedyńczych krotek z wartością *int*. Uruchamiane zostają 3 współbieżne procesy mające odbierać kolejno krotkę z dowolną wartością typu *int*. Wynikiem pozytywnym testu jest równa ilość krotek umieszczonych w przestrzeni z ilością pobranych krotek przez wszystkie procesy. Przykładowy poprawny wynik działania testu:

   ```
   Pushed: 11111
   Pulled: 10000
   Pulled: 11000
   Pulled: 10000
   ```

5. *5_test.sh*

   Przestrzeń krotek jest pusta - tj. nie została w niej umieszczona żadna krotka. Uruchamiany jest proces pobierający krotki wg podanego pliku konfiguracyjnego. Wynikiem działania programu powinien być ciąg zer, reprezentujący otrzymanie *timeout'u*.

6. *6_test.sh*

   Przestrzeń krotek jest pusta - tj. nie została w niej umieszczona żadna krotka. Uruchamiany jest proces kolejno wstawiający krotkę i pobierający ją wg podanego pliku konfiguracyjnego. Wynikiem działania programu powinien być ciąg jedynek, reprezentujący umieszczenie krotek w przestrzeni i pobranie ich z przestrzeni z sukcesem.

10) Obsługa błędów i sytuacji wyjątkowych:

Wszystkie operacje na obiekcie LindaClient mogą zakończyć się błędem sygnalizowanym za pomocą mechanizmu wyjątków. Samo tworzenie instancji obiektu tej klasy powinno zawsze zakończyć się powodzeniem.

Klient korzystający z systemu Linda powinien obsługiwać następujące sytuacje:

1. Niepowodzenie otwarcia pliku FIFO serwera krotek.
   Wyjątek typu: ServerFifoException z wiadomością: "Error while opening server FIFO."
   Znaczenie: Najczęściej oznacza brak działającej instancji serwera.
2. Nieudany zapis do pliku FIFO serwera.
   Wyjątek typu: ServerFifoException z wiadomością "Server doesn't response."
   Znaczenie: Praca serwera została zakończona w sposób nieoczekiwany.
3. Niepowodzenie tworzenia odbiorczego FIFO po stronie klienta.
   Wyjątek typu: ClientFifoException z opisem doprecyzowującym zaistniały problem.
   Znaczenie: Podane razem z obiektem wyjątku.
4. Niepoprawny wzorzec krotki (dla operacji pull oraz preview).
   Wyjątek typu: Shared::ParserException
   Znaczenie: Błąd składniowy wzorca. Np. wybranie nieobsługiwanego typu danych.
5. Niepoprawna lista danych (dla operacji push)
   Wyjątek typu: Shared::ParserException
   Znaczenie: Lista danych zawiera element o typie spoza listy obsługiwanych.
6. Przekroczenie wartości liczbowych dla pól krotki.
   Wyjątek typu: Shared::NumericLimitException
   Znaczenie: Zapytanie zawiera warunki, które przekraczają limity lokalnego systemu operacyjnego dla zmiennych o typach obsługiwanych przez system Linda.

Działanie serwera jest zabezpieczone przed sytuacjami wyjątkowymi. Przykładowo przed błędami dostępu do pliku FIFO klienta. Niemniej jednak uruchomienie procesu serwera może zakończyć się niepowodzeniem w przypadku kiedy w danym systemie operacyjnym pracuje już inna instancja serwera Linda. Podczas normalnego startu procesu serwera tworzony jest plik FIFO. Ta operacja z punktu widzenia systemu operacyjnego również może zakończyć się niepowodzeniem. Serwer reaguje na taką sytuację, ale przyjęte przez nas i opisane wyżej założenia sprawiają iż prawdopodobieństwo jej wystąpienia jest bliskie zeru.