Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Абрамов Иван Александрович

Реализация агрегатора книг заявок валютных рынков для торговой платформы Tbricks

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Романов К. Н.

Рецензент:
к. т. н., доцент Литвинов Ю. В.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Ivan Abramov

Implementation of the FX orderbook aggregator for the Tbricks trading platform

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
Konstantin Romanov

Reviewer:
assistant professor Yuri Litvinov

Saint-Petersburg
2017

Оглавление

Введение **5**

1. Постановка задачи **7**

2. Обзор **8**

2.1. Платформа Tbricks 8

2.2. Существующие решения 9

2.2.1. FX aggregator 11

2.2.2. Soft-FX 11

2.3. Выводы 12

3. Работа с потоками данных платформы Tbricks **13**

3.1. Использование потоков книг заявок 14

4. Реализация агрегатора книг заявок **17**

4.1. Описание типов книг заявок 17

4.2. Формирование агрегированной книги заявок 19

4.3. Обработка ограничений торговых площадок и пользовательских настроек 22

4.3.1. Обработка пользовательских настроек 22

4.3.2. Обработка ограничений торговых площадок 23

5. Реализация приложения для автоматической торговли на валютном рынке **25**

5.1. Описание входных параметров 27

5.2. Разработка обработчика заявок 28

5.3. Разработка торговых сценариев 30

5.3.1. Сценарий Iceberg 30

5.3.2. Сценарий VWAP 31

5.3.3. Сценарий Aggressive Watch 32

6. Тестирование **35**

6.1. Тестирование торговых сценариев с помощью фреймворка TestEngine . 35

6.1.1. Описание фреймворка TestEngine 35

6.1.2. Создание тестового сценария 37

6.2. Модульное тестирование агрегатора с помощью фреймворка Google Test 41

6.3. Тестирование производительности агрегатора книг заявок 42

Заключение **44**

Приложения **45**

Список литературы **48**

Введение

В современном мире эффективное функционирование экономики государства невозможно без развитой финансовой системы. Основу этой системы составляют финансовые рынки, которые принято классифицировать на денежный, фондовый, валютный рынки и рынок капитала. Отдельного внимания заслуживает рынок торговли валютой, который обеспечивает взаимодействие остальных финансовых рынков. По данным на 2016 год, ежедневный оборот на валютном рынке составляет 5,1 триллиона долларов [1].

Торговля валютой значительно отличается от торговли на фондовом рынке. Вместо единой централизованной биржи, характерной для фондового рынка, для валютного рынка существует фиксированное количество торговых площадок, предоставляющих возможность совершать сделки по одному и тому же валютному инструменту. При взаимодействии с различными торговыми площадками необходимо обрабатывать несколько потоков рыночных данных для выбранного инструмента. Полученные данные требуется представить в унифицированном виде, который позволит использовать доступную информацию для анализа и совершения сделок.

Данные с торговых площадок поступают в виде книг заявок (так же называется «стакан заявок» или orderbook), которые можно представить как таблицы. Строка такой таблицы характеризует уровень, где содержится информация об объеме валютного инструмента, его цене на покупку и продажу. Таблицы формируются из заявок, которые отправили на торговую площадку участники рынка. В однин уровень объединяются данные о заявках, имеюших совпадающие цены. Каждая торговая площадка формирует собственную книгу заявок, учитывая имеющиеся ограничения. Используя информацию из книг заявок, а также данные о совершенных сделках, инвесторы вычисляют достоверные технические индикаторы, которые помогают выявлять и отслеживать возможные валютные риски [6].

Для того, чтобы вычислять значения технических индикаторов или эффективно отправлять новые заявки, анализируя ситуацию на рынке, необходимо объединять предоставляемые торговыми площадками книги заявок, принимая во внимание специфику и правила каждой из них. Известным приемом в данной области является создание агрегированной книги из отдельных книг заявок, которая становится базисом для решения представленных ранее задач [7].

Разработчики современного программного обеспечения для автоматической торговли на валютном рынке предоставляют своим клиентам функциональность, позволяющую одновременно работать с несколькими торговыми площадками. Такие решения дают возможность получать информацию, характеризующую состояние рынка, а также отправлять новые заявки на рынок, используя готовые сценарии [3, 8]. Зачастую такие решения являются непосредственной частью конкретной платформы и не

могут быть интегрированы с приложениями, разрабатываемыми на клиентской стороне. По этой причине процесс настройки и расширения программного обеспечения для работы на валютном рынке под нужды конкретной компании или пользователя становится затруднительным.

В компании *Itiviti* [4], которая занимается разработкой финансового программного обеспечения для автоматической торговли на электронных биржах, было принято решение исправить сложившуюся ситуацию и предоставить гибкое решение для работы с валютными рынками. Для этого была поставлена задача разработать модуль для агрегации книг заявок валютных рынков, который можно использовать при реализации пользовательских приложений для разрабатываемой в компании платформы *Tbricks*. Таким образом компания-клиент получит возможность самостоятельно создавать необходимую бизнес-логику, используя и дополняя уже написанный модуль по агрегации книг заявок валютных рынков.

В данной работе описывается создание модуля на основе платформы *Tbricks*, реализующего агрегацию книг заявок. Представленный модуль учитывает требования и особенности существующих валютных рынков. Для того, чтобы показать, что предложенный модуль может успешно применяться при разработке пользовательских приложений, в данной работе описан процесс создания и тестирования комплексного приложения для автоматической торговли, использующего в своей основе разработанный модуль.

1. Постановка задачи

Целью данной работы является реализация модуля для агрегации книг заявок валютных рынков для приложений торговой платформы Tbricks. Для достижения поставленной цели были сформулированы следующие задачи.

1. Реализовать модуль для обработки и агрегации книг заявок:
   (a) поддержать обработку нескольких потоков рыночных данных;
   (b) реализовать корректную работу с книгами заявок разных типов;
   (c) реализовать обработку ограничений торговых площадок и пользовательских настроек;
   (d) реализовать вычисление индикатора VWAP и сбор информации для создания заявок.
2. Создать приложение для автоматической торговли на валютном рынке, которое использует модуль для агрегации книг заявок:
   (a) реализовать сценарии, использующие несколько рынков для торговли;
   (b) реализовать сценарии, анализирующие данные и торгующие на разных рынках.
3. Протестировать модуль для агрегации книг заявок и торговое приложение:
   (a) создать набор тестов-сценариев для проверки поведения торгового приложения;
   (b) создать набор модульных тестов для проверки вычисления индикатора VWAP;
   (c) провести тестирование производительности.

2. Обзор

Для того, чтобы реализовать модуль для агрегации книг заявок валютных рынков, который будет использоваться и расширяться при разработке приложений для торговой платформы *Tbricks*, необходимо четко понимать особенности используемых технологий и структуру сервисов, которые обеспечивают функционирование системы. Во-первых, требуется убедиться, что поставленная задача выполнима и может быть решена в рамках доступного инструментария, предоставляемого платформой. Во-вторых, необходимо описать, какие задачи и в каком контексте должны быть решены. Четкое понимание границ ответственности реализуемого модуля поможет избежать дублирования функциональности в различных программных компонентах. Кроме этого, на данном этапе логично рассмотреть уже готовые решения, которые присутствуют на рынке. На основе найденной открытой информации можно будет проанализировать, какие минимальные требования предъявляются к реализуемому модулю, а также какие удачные решения были приняты компаниями, разрабатывающими схожие продукты. В завершение хочется отметить, что у программистов, разрабатывающих приложения для платформы *Tbricks*, есть сформировавшееся понимание того, как создавать бизнес-логику. Нужно сделать так, чтобы модуль для агрегации книг заявок был не только понятен и прост в использовании, но и гармонично вписывался в существующий процесс разработки.

2.1. Платформа Tbricks

Tbricks – это современная платформа для автоматической торговли на электронных биржах, написанная на C++. Ключевой особенностью платформы является сочетание производительности, масштабируемости, гибкости в настройке и расширяемости. Во многом этого удалось достичь благодаря компонентно-сервисной структуре платформы. Понятие сервиса в данном контексте можно трактовать как услугу, направленную на решение определенных задач. Примером является сервис, который доставляет рыночные данные в систему. В свою очередь, компонента — это процесс, запущенный на физическом сервере, реализующий один или несколько сервисов. Доступные сервисы можно сгруппировать и представить следующим образом:

- сервисы для администрирования;
- сервисы для взаимодействия с графическим интерфейсом;
- сервисы приложений;
- интеграционные сервисы;
- сервисы хранения;

- торговые сервисы;
- сервисы рыночных данных.

Так как модуль для агрегации книг заявок валютных рынков будет использоваться приложениями, то данная группа сервисов заслуживает отдельного внимания. *Tbricks* приложение — это плагин, который загружается в подходящий по типу сервис. Далее плагин запускается и ожидает события от других сервисов. В зависимости от своей задачи приложения принято делить на 3 типа:

- стратегии — приложения, которые занимаются торговлей;
- вычисления — приложения, вычисляющие технические индикаторы;
- вспомогательные — приложения, используемые для импорта, экспорта данных и генерации отчетов.

Так как одной из поставленных задач является создание приложения для автоматической торговли на валютном рынке, то рассмотрим сервисы, которые могут быть задействованы при разработке стратегии помимо сервиса приложений. Во-первых, торговое приложение должно собирать информацию о валютных инструментах, представленных на торговых площадках. Для этого оно будет взаимодействовать с сервисом инструментов, который является представителем группы сервисов рыночных данных. Во-вторых, для найденных инструментов потребуется получить рыночную информацию — книги заявок. С этой задачей помогут справиться потоки данных *Tbricks*, являющиеся базовым механизмом получения данных. На основе книг заявок возможно будет реализовать торговые алгоритмы, формирующие новые заявки для валютных рынков. Ответственность за размещение заявок возьмут на себя торговые сервисы.

Подробное описание взаимодействия с представленными сервисами будет дано в основной части работы. На данном этапе хочется отметить, что было принято решение, что функциональность, предоставляемая платформой, является достаточной для того, чтобы реализовывать агрегатор книг заявок. По этой причине технические детали реализации будут тесно связаны с инструментарием платформы *Tbricks*.

2.2. Существующие решения

Прежде чем перейти к описанию решений, представленных на рынке, рассмотрим некоторые публикации, связанные с агрегацией книг заявок. Одной из немногих работ, которые написаны по этой теме, является [7]. В данной публикации автор отмечает необходимость агрегации рыночных данных, предлагая формировать единую агрегированную книгу заявок на основе отдельных книг (рис. 1).

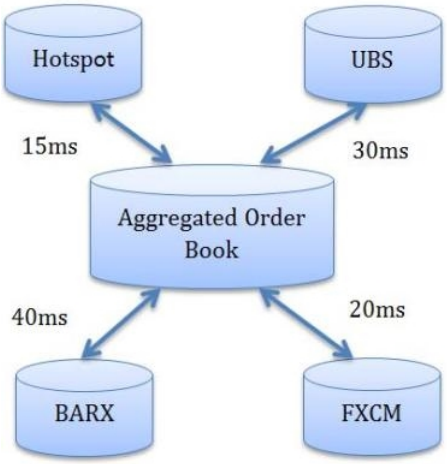

Рис. 1: Агрегированная книга заявок (рис. из [7])

Кроме этого, особое внимание автор уделяет событийно-ориентированному подходу, который, по его мнению, должен использоваться при разработке агрегатора. Отметим, что механизм потоков данных, используемый в платформе Tbricks, успешно подстраивается под данный критерий, из чего можно сделать вывод, что решение о таком способе взаимодействия с сервисом рыночных данных было сделано верно.

Однако, в представленной работе не описаны детали агрегации. Автор не рассматривает особенности, связанные с разными типами книг заявок и вычислением технических индикаторов, а больше уделяет внимание описанию архитектуры торговой системы.

Для того, чтобы иметь достаточно информации о природе рыночных данных, были рассмотрены статьи [5] и [2]. В данных статьях рассматривается популярный технический индикатор VWAP, который используется для мониторинга рыночных данных. Было отмечено, что в агрегаторе книг заявок индикатор VWAP будет считаться в других условиях, нежели чем в представленных работах. Разница заключается в том, что в статьях анализируются данные за торговый день или более длительный отрезок времени. Для агрегатора же потребуется вычислять конкретное значение с опорой на текущее состояние рынка без использования исторических данных. Информация о математических моделях, представленных в статьях, не учитывалась, так как эта обширная тема выходит за рамки работы.

Для понимания функциональности приложения для автоматической торговли, разработанного на основе агрегатора книг заявок, были рассмотрены популярные торговые решения. Программное обеспечение, используемое в данной сфере, является проприетарным, поэтому выводы о рассмотренных решениях пришлось делать на основе открытой информации.

2.2.1. FX aggregator

Компания *FX aggregator* предоставляет своим клиентам программу по агрегации потоков биржевых котировок, которая работает с более чем пятьюдесятью источниками ликвидности. Данное решение используется в крупных российский банках и финансовых организациях. *FX aggregator* — это комплексный продукт, включающий в себя:

- клиентский терминал;
- приложения для управления счетами клиентов;
- систему передачи сделок в бэк-офисные приложения;
- набор торговых алгоритмических стратегий.

Компания не раскрывает детали реализации агрегатора, но предоставляет пользовательскую информацию о торговых сценариях, которые могут иметь тип:

- limit;
- stop loss;
- market;
- immediate or cancel.

Тип limit гарантирует, что сделка будет исполнена по цене не хуже той, что определил пользователь. Сценарий типа market исполняет заданный объем по рыночным ценам. Тип stop loss говорит о том, что стратегия срабатывает в момент, когда цены достигают определенного уровня, и исполняет заданный объем по рыночным ценам. Тип immediate or cancel гарантирует, что сделка либо будет исполнена в данный момент времени, либо будет отменена.

Отметим, что представленные типы сценариев дают возможность проводить разнообразную торговую политику, но не являются достаточными. Например, нельзя указать торговые площадки, которые будут использоваться в сценарии stop loss для принятия решения, но не для торговли.

2.2.2. Soft-FX

Компания *Soft-FX* занимается разработкой программного решения для агрегации ликвидности от различных поставщиков. Агрегатор получает и обрабатывает информацию о спросе и предложении и гарантирует исполнение заявок клиентов по наилучшей цене, доступной на рынке. Аналогично предыдущему рассмотренному решению разработка *Soft-FX* поддерживает торговые сценарии типа: limit, market, stop loss.

Наличие специального сервиса, который отвечает за обработку котировок, выгодно отличает продукт *Soft-FX* от аналогов. Данный сервис фильтрует нерыночные и дублирующие котировки, защищая систему от резких скачков цен.

2.3. Выводы

После рассмотрения статей и программных продуктов можно сделать ряд выводов относительно проектируемого агрегатора. В основе разрабатываемого модуля для агрегации книг заявок будет лежать идея создания единой агрегированной книги заявок, которая будет получать данные через потоки платформы *Tbricks*. Рыночные данные должны будут пройти проверку на корректность, а проблемные источники данных вовремя исключены. Созданный агрегатор должен использоваться в разрабатываемом торговом приложении, которое реализует популярные торговые сценарии, добавив возможность раздельно использовать рынки для анализа и торговли.

3. Работа с потоками данных платформы Tbricks

Для Tbricks-приложений основным средством получения информации от активных сервисов являются потоки данных. Приложение, которое заинтересовано в данных определенного типа, открывает соответствующий поток, используя фильтр, чтобы специфицировать требуемые критерии. Данные, удовлетворяющие критериям, частями поступают в приложение и могут использоваться для реализации бизнес-логики.

Потоки данных работают в двух режимах:

- режим снимка;
- режим обновлений.

Режим снимка позволяет потоку данных предоставить приложению информацию, которая удовлетворяла критериям фильтра и была доступна в системе в момент, когда поток данных был открыт. По сути, данный режим описывает данные в момент запроса, но не отслеживает последующие изменения. Зачастую режим снимка используют, когда требуется импортировать или экспортировать данные в систему.

Чтобы получать уведомления об изменениях интересующих данных, необходимо открывать поток в режиме обновлений. Сперва поток данных, открытый в режиме обновлений, так же как и поток, открытый в режиме снимка, предоставит информацию обо всех доступных в системе данных, удовлетворяющим критериям выбранного фильтра. После этого приложение получит уведомление о том, что снимок завершен. Далее приложение, которое открыло поток в режиме обновлений, будет получать уведомления о том, что отслеживаемые данные изменились. Потоки данных в режиме обновлений открывают приложения, которые должны анализировать информацию и реагировать на ее изменение. Большинство торговых приложений устроены именно так.

Для того, чтобы использовать концепцию потоков данных в логике приложения, необходимо реализовать интерфейс, состоящий из 4-х методов (листинг 1).

Листинг 1: Интерфейс потока данных

```
virtual void HandleStreamOpen(const tbricks::StreamIdentifier& stream);
virtual void HandleStreamFailed(const tbricks::StreamIdentifier& stream);
virtual void HandleSnapshotEnd(const tbricks::StreamIdentifier& stream);
virtual void HandleStreamStale(const tbricks::StreamIdentifier& stream);
```

Метод *HandleStreamOpen* оповещает приложение о том, что выбранный поток данных, чей идентификатор передается в качестве аргумента, был успешно открыт. Кроме этого, данный метод вызывается, когда данные снова стали доступны после временной задержки.

Метод *HandleStreamFailed* уведомляет приложение о том, что возникли серьезные проблемы с потоком данных. Чаще всего такие ошибки происходят, когда в поток данных был передан некорректный фильтр и поток не смог запуститься.

Метод *HandleSnapshotEnd* оповещает приложение о том, что снимок данных завершен. В случае, если поток был открыт в режиме обновлений, то стоит ожидать поступление данных и после вызова данного метода.

Метод *HandleStreamStale* уведомляет приложение о том, что поток данных временно не может предоставлять данные. В большинстве случаев проблема вызвана тем, что приложение потеряло связь с сервисом, который являлся поставщиком данных. В случае если соединение удается восстановить, в приложении будет вызван метод *HandleStreamOpen*, за которым последуют обновления с актуальными данными.

Стоит отметить, что представленный интерфейс является общим для всех потоков данных. Но каждый отдельно взятый поток дополнительно требует от приложения реализации метода, который будет являться обработчиком данных, специфичных для данного потока. В качестве примера рассмотрим поток, который предоставляет лучшие цены с рынка для конкретного инструмента. Это значит, что в приложение с рынка поступит информация о самых низких ценах, по которым можно осуществить покупку, и информация о самых высоких ценах, по которым можно продать рассматриваемый инструмент. Информация о лучших ценах содержится в объекте *price* и будет передана в реализацию метода после того, как поток данных будет открыт (листинг 2).

Листинг 2: Дополнительный метод потока лучших цен

```
virtual void HandleBestPrice(const tbricks::StreamIdentifier& stream,
                             const tbricks::BestPrice& price);
```

Так как в данной работе поставлена задача реализовать агрегатор книг заявок, то далее подробно будут рассматриваться потоки данных, предоставляющие книги заявок.

3.1. Использование потоков книг заявок

Операции, связанные с валютными парами, осуществляются на торговых площадках. Стороны, заинтересованные в покупке или продаже определенной валюты, оставляют на торговой площадке заявки, где указывают информацию об объеме заявки и цене, по которой они готовы заключить сделку. Заявки, поступившие на определенную валютную пару, например *EUR/USD*, объединяются в единую таблицу, которая называется книгой заявок (стаканом заявок) или orderbook. Пример такой книги представлен на таблице 1.

Количество строк в такой таблице называют глубиной книги заявок, а отдельную строку — уровнем книги заявок. Уровень книги заявок содержит информацию об

Таблица 1: Пример книги заявок

Покупка	Продажа	Объем
1.3505	1.3506	1000000
1.3504	1.3507	3000000
1.3503	1.3508	5000000
1.3502	1.3509	10000000

объеме и цене. Торговая площадка объединяет в один уровень заявки, имеющие совпадающую цену. Для того, чтобы подписаться на получение книг заявок, приложение должно использовать поток *OrderBookStream*. Приложение реализует специфичный для данного потока данных интерфейс (листинг 3).

Листинг 3: Дополнительный метод потока книг заявок

```
virtual void HandleOrderBook(const tbricks::StreamIdentifier &stream,
                             const tbricks::OrderBook::Update &update);
```

После этого приложение создает объект *OrderBook*, который представляет книгу заявок. Для того, чтобы поддерживать книгу заявок в актуальном состоянии, приложение использует объект *OrderBook::Update*. В объекте *OrderBook::Update* находится информация об изменениях книги заявок, которые произошли на торговой площадке. Чтобы добавить полученные с рынка изменения в локальную копию книги, необходимо применить *OrderBook::Update* к *OrderBook*, используя функцию *Merge*. Если применение обновлений прошло удачно, то из объекта *OrderBook* можно получить информацию о текущих уровнях и использовать ее при реализации бизнес логики.

Описанный сценарий работы с потоками книг заявок использовался при реализации агрегатора книг заявок. На вход поступал набор идентификаторов финансовых инструментов, для которых открывались потоки книг заявок. Для каждого потока данных поддерживалась в актуальном состоянии соответствующая книга заявок. В определенный момент времени из книг заявок извлекались уровни и передавались в класс, который занимался формированием агрегированной книги заявок на основе существующих. Особенности класса-агрегатора будут подробно рассмотрены в следующей главе.

При реализации функциональности, использующей потоки книг заявок, возник ряд проблем, которые изначально не были учтены. Для каждой проблемы было предложено решение, которое помогло исправить сложившуюся ситуацию

Во-первых, оказалось, что для отслеживания состояния потока книг данных не достаточно обрабатывать уведомления, поступающие через *HandleStreamStale* и *HandleStreamFail*. Торговая площадка, поставляющая данные через открытый поток, может информировать пользователя о возникших проблемах, передавая сообщения со статусом. Для того, чтобы получать статус-сообщения, разработчики используют

поток StatusStream. Данный поток требует реализации интерфейса, представленного на листинге 4.

Листинг 4: Метод потока уведомлений

```
virtual void HandleInstrumentStatus(const tbricks::StreamIdentifier& stream,
                                    const tbricks::InstrumentStatus& status);
```

Из полученного статусного сообщения извлекается информация о том, в каком состоянии находится поток данных. Данная информация позволяет сделать вывод о том, насколько локальная информация о книге заявок соответствует реальной информации, которая находится на торговой площадке. Было принято решение, что необходимо исключать из агрегации полученные данные о книге заявок в случае, если для данной книги пришло статус-сообщение с флагом:

- stale;
- fail.

Важно отметить, что хранимый объект *OrderBook* не должен удаляться в ситуации, когда было получено проблемное статус-сообщение. Так как *OrderBook* модифицируется путем получения обновлений, то после того, как будет обработано сообщение о том, что поток данных восстановился, последует *OrderBook::Update*, который переведет хранимую книгу заявок в актуальное состояние.

Так как сообщение из *StatusStream*, о том, что поток книг заявок испытывает проблемы, может поступить после того, как ошибочные данные попали в агрегатор, необходимо добавить дополнительные проверки. Стоит учитывать, что сервис-поставщик данных контролирует, что уровни книг заявок не будут пустыми, поэтому приложение должно проверить, что записанным в книге заявок данным можно доверять. Опытным путем было обнаружено, что в агрегатор может поступить книга заявок, для которой уровни на продажу и покупку пересекаются по цене. Такой ситуации быть не должно, так как удовлетворяющие друг друга по цене заявки на покупку и продажу должны быть одобрены и обработаны торговой площадкой. Поэтому была реализована проверка пересечений уровней, и в случае, если такая ситуация произойдет, данные из проблемной книги заявок не будут учитываться при агрегации.

В заключении раздела хочется отметить, что объекты, описывающие уровни в книге заявок *OrderBook*, напрямую не использовались в коде агрегатора. Это было связано с тем, что в каждом таком объекте хранилась дополнительная метаинформация, которая была нужна только на этапе применения обновлений. Поэтому было принято решение использовать собственный легковесный объект *LightDepth*, который бы хранил только данные о торгуемом на конкретной площадке инструменте, объем, цену и тип уровня. Именно контейнер с объектами *LightDepth* поступал на вход вычислительной логике агрегатора.

4. Реализация агрегатора книг заявок

Агрегатор книг заявок валютных рынков задумывался как модуль, который поможет разработчикам приложений торговой платформы *Tbricks* комфортно реализовывать бизнес-логику, связанную с валютными рынками. Так как подход, подразумевающий подписку на определенные потоки данных, является основным в концепции разработки *Tbricks* приложений, то он в значительной степени повлиял на архитектуру реализованного агрегатора, основные классы которого представлены на рисунке 2. Было принято решение, что приложение будет подписываться на агрегированные обновления с рынков. Для этого приложению потребуется реализовать интерфейс *ConsolidatedOrderbook::Stream::IHandler* и открыть поток *ConsolidatedOrderbook::Stream*, которому на вход будут переданы настройки *ConsolidatedOrderbook::Stream::Options*. В объекте настроек указываются идентификаторы валютных инструментов, представленных на торговых площадках. Для каждого идентификатора будут открываться *OrderBookStream* и *StatusStream*, которые были описаны ранее. Вся логика по обработке сообщений от открытых потоков данных будет сокрыта в объекте *ConsolidatedOrderbook::Stream*. Клиентское приложение, реализовавшее интерфейс *ConsolidatedOrderbook::Stream::IHandler*, сможет отслеживать цены, указав потоку определенный объем. Если для выбранного объема изменится цена, то клиентское приложение получит соответствующее уведомление через метод *HandleConsolidatedOrderBook*. На основе полученного уведомления приложение сможет принять решение о том, что необходимо выполнить определенныей действия.

После краткого обзора предполагаемого сценария использования агрегатора необходимо детально описать последовательность действий, которые связаны с работой с рыночными данными. На этом этапе можно считать, что в агрегатор поступил набор контейнеров с объектами *LightDepth*, которые описывают уровни книги заявок для выбранных валютных инструментов.

4.1. Описание типов книг заявок

Поступающие с торговых площадок книги заявок делятся на два типа:

- лимитные (limit);
- с ценовыми диапазонами (price banded).

Тип книги заявок влияет на то, какие правила должен соблюдать клиент, если он собирается совершать сделки на конкретной торговой площадке. Для того, чтобы наглядно продемонстрировать существующие различия между книгами заявок разных типов, будем использовать в качестве примера таблицу, которая может представлять собой как лимитную книгу, так и книгу с ценовыми диапазонами. Для удобства будем считать, что клиент заинтересован в том, чтобы купить объем 2000000.

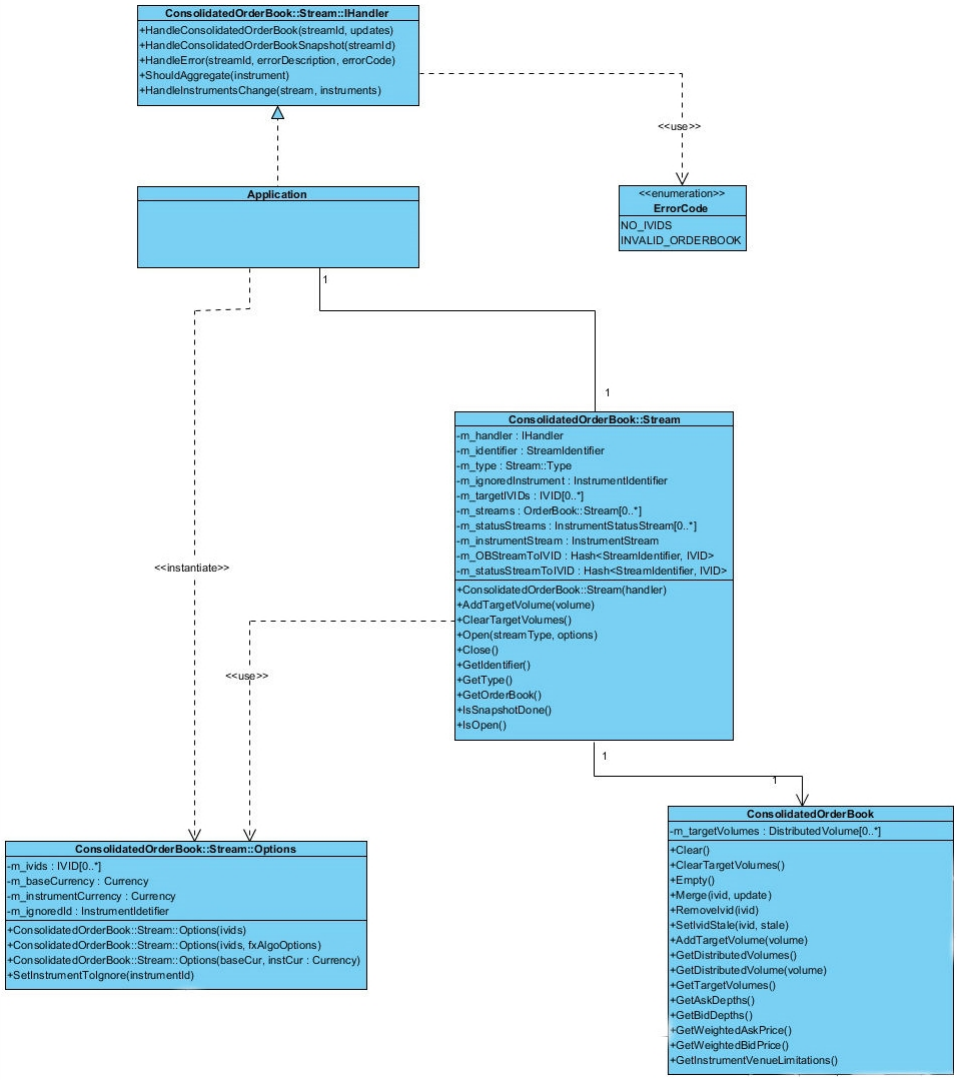

Рис. 2: Диаграмма классов, описывающая сценарий использования агрегатора

В случае, если представленная книга заявок является лимитной, то подразумевается, что клиент имеет возможность выкупать все доступные уровни. Это значит, что при совершении сделки на покупку сначала будут обработаны уровни с наименьшей ценой. В рассматриваемом примере со сделкой объемом 2000000 будут обработаны следующие уровни, представленные в таблице 3.

Если вычислять значение цены средневзвешенной объемом по формуле (VWAP, Volume Weighted Average Price):

$$VWAP = \frac{\sum_{i=1}^{n} price_i \cdot volume_i}{\sum_{i=1}^{n} volume_i} \tag{1}$$

то получается значение индикатора VWAP равное 1.35065.

В случае, если книга заявок является книгой с ценовым диапазоном, то рассматриваемая заявка объемом 2000000 будет исполнена по другому сценарию. Для запрашиваемого объема будет выбран диапазон (в представленном случае для 2000000 это

Таблица 2: Пример книги заявок

Покупка	Продажа	Объем
1.3505	1.3506	1000000
1.3504	1.3507	3000000
1.3503	1.3508	5000000
1.3502	1.3509	10000000

Таблица 3: Описание сделки для лимитной книги заявок

Цена	Объем
1.3506	1000000
1.3507	1000000

диапазон от 1000000 до 3000000) и соответствующая диапазону цена. Поэтому заявка будет обработана по сценарию представленному на таблице 4 Значение VWAP ин-

Таблица 4: Описание сделки для книги с ценовым диапазоном

Цена	Объем
1.3507	2000000

дикатора в данном случае равно 1.3507. Логично предположить, что в случае книги заявок с ценовым диапазоном удобно разбить исходную заявку на несколько, например, на две заявки по 1000000, и выполнить их независимо по меньшей цене. Такая стратегия идет в противоречие с правилами, которые требуется соблюдать клиентам торговой площадки, использующей книги заявок с ценовым диапазоном, поэтому дробление исходной заявки на набор меньших по объему запрещено.

4.2. Формирование агрегированной книги заявок

Так как агрегатор книг заявок работает с несколькими торговыми площадками, то ему на вход поступают данные о нескольких книгах. Для того, чтобы агрегатор смог вычислить значение технического индикатора VWAP для заданного объема и сформировать вспомогательные данные для новых заявок, необходимо объединить поступившие книги в единую агрегированную книгу. Хочется отметить, что большинство площадок используют лимитные книги заявок, поэтому первоначально будет рассмотрен простой случай, когда объединяются только лимитные книги заявок.

Допустим, что представленные таблицы 5 и 6 описывают уровни с ценами на продажу для лимитных книг заявок. Объединенная таблица 7 является результатом сли-

Таблица 5: Книга заявок A

Покупка	Объем
1.3505	500000
1.3506	3000000

Таблица 6: Книга заявок B

Покупка	Объем
1.3506	1000000
1.3507	3000000
1.3508	5000000

яния представленных таблиц, при условии что сохранятся упорядоченность по цене. Отметим, что на данном этапе не учитываются приоритеты торговых площадок, поэтому относительный порядок уровней с одинаковыми ценами не важен. Вычисление индикатора VWAP для агрегированной книги становится простой задачей — достаточно просматривать книгу с первого уровня до тех пор, пока требуемый объем не будет найден. Важно при просмотре книги сохранять информацию о том, какой уровень и на какой площадке планируется выкупить. Собранные статистические данные будут использоваться, когда возникнет необходимость отправлять новые заявки на рынок.

Интересный случай возникает тогда, когда необходимо агрегировать книги заявок разных типов. Так как хотелось работать с данными единообразно, то было принято решение конвертировать книги заявок с ценовым диапазоном в лимитные книги заявок. Для того, чтобы конвертированные книги удовлетворяли требованиям площадок, использующих книги с ценовым диапазоном, были введены дополнительные ограничения.

Рассмотрим ситуацию, когда таблица 6 представляет книгу заявок с ценовым диапазоном. В случае книги заявок с ценовым диапазоном мы можем быть уверены в том, что при переходе от одного уровня к другому объем будет возрастать. Чтобы книга заявок с ценовым диапазоном стала похожа на лимитную книгу заявок, необходимо преобразовать уровни так, чтобы иметь возможность выкупать их последовательно.

Обратим внимание на таблицу 8, которая будет соответствовать сконвертированной таблице 6. Отметим, что, совершив преобразованием, мы получили возможность последовательно обрабатывать уровни. Например, для объема 3000000 будут выкуплены уровни, представленные в таблице 9

Получается, что $\frac{1.3506 \cdot 1000000 + 1.35075 \cdot 2000000}{3000000} = 1.3507$ – цена, соответствующая объему 3000000 в агрегированной книге заявок.

Таблица 7: Объединенние книг заявок 5 и 6

Покупка	Объем
1.3505	500000
1.3506	3000000
1.3506	1000000
1.3507	3000000
1.3508	5000000

Таблица 8: Книга заявок B'

Покупка	Объем
1.3506	1000000
1.35075	2000000
1.35095	2000000

Заметим, что сконвертированная книга заявок отличается от обычной лимитной книги. Во-первых, если клиент захочет купить объем, не указанный на уровне книги заявок с ценовым диапазоном (например, 2000000), то для такого объема цена будет посчитана неверно. Будут выкуплены уровни, представленные на таблице 10. Полученная цена $\frac{1.3506 \cdot 1000000 + 1.35075 \cdot 1000000}{3000000} = 1.350675$ не является допустимой ценой для книги заявок с ценовым диапазоном, так как в книге нет уровня с такой ценой.

Во-вторых, есть книги заявок, для которых сконвертированная книга имеет не упорядоченные по цене уровни. Если мы специально отсортируем уровни, то мы получим возможность выкупить объем по ценам, которые не представлены в исходной книге заявок.

Чтобы избежать описанных выше проблем, было принято решение добавить зависимости между уровнями конвертированной книги заявок. Уровень конвертированной книги заявок будет указывать на уровень, который должен быть выкуплен перед ним. Такой подход гарантирует то, что уровни будут выкуплены в правильном порядке. Кроме этого, при покупке объема, не указанного в исходной книге заявок с ценовыми диапазонами, будет учитываться штраф за выкуп нового уровня конвертированной книги. Такое решение обеспечит то, что цены, которые будут получены из конвертированной книги, всегда будут соответствовать ценам из исходной книги заявок с ценовыми диапазонами.

Таблица 9: Описание сделки для сконвертированной книги заявок

Цена	Объем
1.3506	1000000
1.35075	2000000

Таблица 10: Описание сделки для сконвертированной книги заявок

Цена	Объем
1.3506	1000000
1.35075	1000000

4.3. Обработка ограничений торговых площадок и пользовательских настроек

Прежде чем агрегатор приступит к вычислению значений технических индикаторов и формированию вспомогательных данных для отправки новых заявок на основе агрегированной книги заявок, ему необходимо учесть дополнительные ограничения, которые можно разделить на два вида:

- пользовательские настройки;
- ограничения торговых площадок.

4.3.1. Обработка пользовательских настроек

Пользовательские настройки задаются клиентом через специальную древовидную структуру *TreeNode*. Такие настройки описывают конфигурацию, выставленную на конкретной торговой системе. Для агрегатора книг заявок был создан отдельный тип *TreeNode* структуры под названием *FX algo configuration*, где указывалась таблица с настройками. Для каждой валютной пары на выбранной торговой площадке разрешалось указать:

- коэффициент масштабирования объема;
- приоритет торговой площадки.

Коэффициент масштабирования объема показывает, что у пользователя есть определенная информация о том, что поступающие с торговой площадки данные не отражают действительность. Например, когда пользователь знает, что торговая площадка поэтапно размещает данные о доступных заявках. Поэтому, когда с рынка приходит книга заявок, пользователь хочет видеть значения объемов уровней, умноженные на

определенный коэффициент. Именно этот коэффициент и называется коэффициентом масштабирования объема. Агрегатор может модифицировать полученные с рынка объемы до того, как они попадут в агрегированную книгу заявок. Таким образом коэффициент масштабирования будет учтен.

Приоритет торговой площадки – это неотрицательное целое число, которое определяет, в каком порядке будут выкуплены уровни агрегированной книги заявок в случае, если они имеют одинаковую цену. Например, в объединенной таблице 7 присутствуют два уровня с ценой 1.3506. Первым будет выкуплен уровень торговой площадки, у которой больше приоритет.

4.3.2. Обработка ограничений торговых площадок

Кроме пользовательских настроек агрегатор учитывает и ограничения торговых площадок. Хранением и обработкой таких ограничений занимается отдельный сервис, поэтому желательно, чтобы приложение один раз сформировало информацию об ограничениях торговых площадок и использовало ее на всех последующих этапах исполнения. Далее каждое из ограничений будет рассмотрено в отдельности.

Для того, чтобы корректно взаимодействовать с торговой площадкой, необходимо знать, какие типы заявок она обрабатывает. Агрегатор умеет формировать данные по заявкам следующего типа:

- лимитные;
- со средневзвешенной ценой.

В лимитных заявках указывается цена, не хуже которой должна пройти сделка. Например, если рассмотреть таблицу 2 для объема 2000000, то там выкупались уровни с ценой 1.3506 и 1.3507. Чтобы провести сделку по покупке указанных уровней, необходимо отправить заявку объемом 2000000 и лимитной ценой 1.3507. Заметим, что так как цена 1.3506 меньше лимита, то данный уровень будет выкуплен перед тем, как будет выкуплена часть уровня 1.3507. В заявках со средневзвешенной ценой указывается VWAP цена для выкупаемого объема. Поэтому формируемая заявка для 2000000 имеет цену 1.35065.

Для заявок со средневзвешенной ценой значение цены вычисляется, а не выбирается из уровня книги заявок. По этой причине торговые площадки требуют, чтобы клиенты использовали определенную политику округления цен. Рассматриваемые валютные рынки используют одну из следующих политик округления:

- bank;
- fair;
- market;

- none.

Представленные политики округления были реализованы в стандартном типе Price. Данное решение позволило агрегатору учесть рассмотренное ограничение на этапе сбора вспомогательных данных.

Некоторые торговые площадки определяют ограничения на минимальный объем заявок, которые принимаются в обработку.

5. Реализация приложения для автоматической торговли на валютном рынке

Основным способом реализации бизнес-логики для торговой платформы *Tbricks* является создание приложений. Стандартное *Tbricks* приложение загружается в специальный сервис, где оно сможет получать и обрабатывать внешние события. Примером такого события может быть поступление рыночных данных от торгового сервиса или взаимодействие пользователя с графическим интерфейсом приложения. В зависимости от полученных данных и текущего состояния приложение реализует свой поведенческий сценарий.

Чтобы иметь полноценное представление о том, как происходит разработка *Tbricks* приложений, рассмотрим важнейшие интерфейсы и типы данных, связанные с приложением. Начнем с того, что приложение должно реализовать интерфейс *Strategy* (листинг 5).

Листинг 5: Интерфейс *Strategy*

```
virtual void HandleRunRequest();
virtual void HandlePauseRequest();
virtual void HandleDeleteRequest();
virtual void HandleModifyRequest(const StrategyModifier& modifier);
```

Чтобы объяснить назначение первых трех методов, необходимо упомянуть, что Tbricks приложение может находиться в одном из трех состояний:

- работает (running);
- остановлено (paused);
- удалено (deleted).

Решение о переходе из одного состояния в другое приложение принимает самостоятельно. У сервиса есть возможность уведомить приложение о том, что требуется изменить состояние, вызвав один из представленных методов (*HandleRunRequest*, *HandlePauseRequest* или *HandleDeleteRequest*). Причина, по которой смена состояния организована таким образом, заключается в том, что порой приложению требуется выполнить определенный набор действий, чтобы корректно перейти в другое состояние. Например, торговое приложение обязано убрать все активные заявки с рынка, прежде чем оно перейдет в удаленное состояние. После того, как приложение перешло в удаленное состояние, в него перестают поступать события, и сервис приложений исключает его из планирования.

Кроме состояния у приложения имеются атрибуты и параметры. В атрибутах содержится описательная информация, характерная для всех приложений. Примером

такой информации может быть имя создателя приложения, время последней модификации или тип приложения. В параметрах же содержатся данные, специфичные для конкретного приложения. В большинстве случаев параметры влияют на сценарий исполнения приложения. Параметры могут быть определены в момент запуска, либо могут быть изменены во время исполнения приложения. В каждом из рассмотренных случаев будет вызываться метод *HandleModifyRequest*, где аргумент *modifier* хранит в себе переданные значения параметров. Так как идентификатор параметра известен на этапе запуска приложения, то он будет использоваться для поиска и извлечения необходимого значения параметра из объекта *StrategyModifier*.

Описанные интерфейсы и данные характерны для произвольного Tbricks приложения. Так как одной из задач данной работы является создание приложения для автоматической торговли на валютных рынках, то разумно рассмотреть приложения, специализирующиеся на отправке новых заявок. Для этого определим сервисы, с которыми взаимодействует торговая стратегия (рис. 3), и опишем ее стандартный сценарий работы.

Рис. 3: Описание взаимодействие приложения с сервисами

Изначально торговая стратегия, загруженная в сервис приложений, определяет список инструментов, которые будут являться предметом будущих сделок. Стратегия обращается в сервис инструментов, чтобы получить требуемые идентификаторы. Далее, если необходимо, стратегия получает рыночную информацию по интересующим инструментам, используя потоки данных от сервиса рыночных данных. После того, как информация получена, приложение формирует заявки, которые передаются в торговый сервис. Для каждой заявки приходит подтверждение о том, что она была размещена и одобрена. Если при обработке заявки возникли сложности, торговый сервис пришлет уведомление с описанием проблемы. Торговая стратегия должна быть готова к обработке таких исключительных ситуаций.

Чтобы подробно рассмотреть представленные этапы работы торговой стратегии, нужно понимать природу входных данных и существующие требования. По этой причине далее будут рассматриваться параметры приложения для автоматической торговли на валютном рынке, для которого была реализована торговая логика.

5.1. Описание входных параметров

Основной задачей разрабатываемого приложения является отправка новых заявок на валютные рынки. Но стоит отметить, что бывают различные ситуации, в которых пользователь принимает решение совершить сделку по валютному инструменту. Например, пользователь может открыть торговое приложение и исполнить одну заявку на конкретной торговой площадке, либо поставить условие на отправление заявки в случае, когда рыночные данные на заранее определенном рынке пробьют определенный уровень цен. Получается, что приложение должно реализовывать набор торговых сценариев, которые сильно отличаются друг от друга. Каждый такой торговый сценарий может требовать для корректного исполнения определенный набор входных параметров, который будет отличаться от набора параметров другого приложения. Но есть список параметров, который требуется каждому приложению. В этот список входят:

- базовая валюта;
- котируемая валюта;
- список площадок для торговли;
- тип торгового сценария;
- объем сделки;
- лимитная цена;
- тип сделки.

Анализируя список обязательных параметров, можно выделить функциональность, которая будет использоваться с каждым торговым сценарием. Во-первых, по валютной паре (базовая валюта, котируемая валюта) и списку торговых площадок необходимо определить идентификаторы инструментов, которые будут предметом сделки. Для этого требуется реализовать модуль для взаимодействия с сервисом инструментов. Полученные идентификаторы вместе с объемом, типом сделки и лимитной ценой должны быть переданы на вход выбранному пользователем торговому сценарию. На данном этапе желательно учесть, что торговых сценариев будет несколько, и приложение должно взаимодействовать с ними единообразно. По этой

причине необходимо предложить интерфейс, который будет реализовывать каждый из торговых сценариев. Торговый сценарий будет анализировать рыночные данные, взаимодействуя с соответствующим сервисом. В определенный момент времени сценарий передаст в приложение данные, на основе которых будут созданы новые заявки. Сформированные заявки будут отправлены на рынки, а приложение будет ожидать уведомления о статусе каждой из заявок. В момент, когда заявки будут одобрены и требуемый объем выкуплен, приложение должно перейти в удаленное состояние.

Теперь, когда рассмотрены основные этапы работы торгового приложения, необходимо предложить и реализовать архитектуру, которая обеспечит успешное функционирование приложения. Далее в разделе будут описаны основные решения, принятые при разработке общего кода, который обеспечивает проверку входных параметров и отслеживание состояния созданных заявок. После этого будут описаны реализованные торговые сценарии и их особенности.

5.2. Разработка обработчика заявок

Общий код приложения, который будет заниматься обработкой параметров и отслеживанием созданных заявок, будем называть обработчиком заявок. Как уже было упомянуто ранее, приложение, загружаемое в сервис, должно реализовать интерфейс Strategy. Кроме этого, чтобы взаимодействовать с торговым сервисом, требуется реализовать интерфейсы *OrderManager* и *IRequestReplyHandler* (рис. 4).

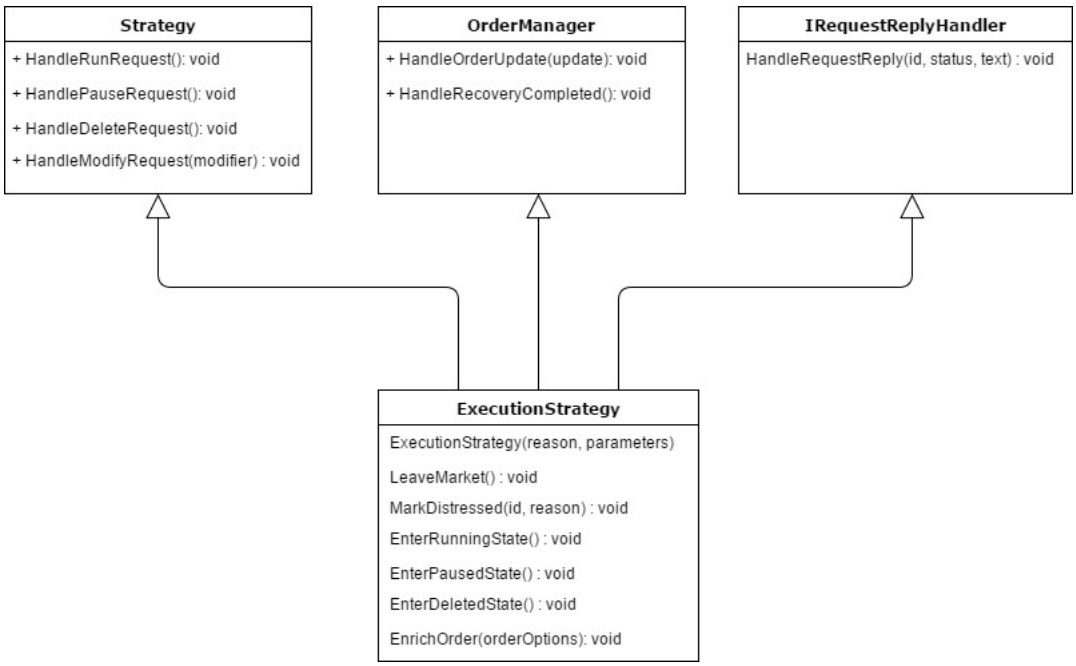

Рис. 4: Диаграмма классов, описывающая реализацию класса ExecutionStrategy

Приложение получает информацию о том, что заявка была создана через метод *HandleOrderUpdate*. Класс *ExecutionStrategy* реализует логику, которая обеспечивает

взаимодействие с торговыми площадками. Именно этот класс отслеживает созданные заявки и ведет статистику о том, какой объем и по какой цене был исполнен, статистику объема и цены исполнения. Отметим, что класс *ExecutionStrategy* принимает решения о смене состояния приложения. При переходе в остановленное или удаленное состояние размещенные на рынке заявки отзываются. Кроме этого, было принято решение о том, что если торговая площадка в ответ на заявку присылает уведомление со статусом *Fail*, то инструмент, размещенный на данной площадке, исключается из рассмотрения. Отметим, что для этого был специально добавлен булев атрибут *Exclude in FX algo*, который всегда проверяется перед тем, как идентификатор инструмента поступит в торговый сценарий.

Чтобы обрабатывать параметры приложения и взаимодействовать с торговым сценарием, был создан класс FXAlgo (рис. 5). FXAlgo проверяет входные параметры и активирует реализованный CurrencyInstrumentManager — класс, взаимодействующий с сервисом инструментов. CurrencyInstrumentManager открывает поток валютных ин-

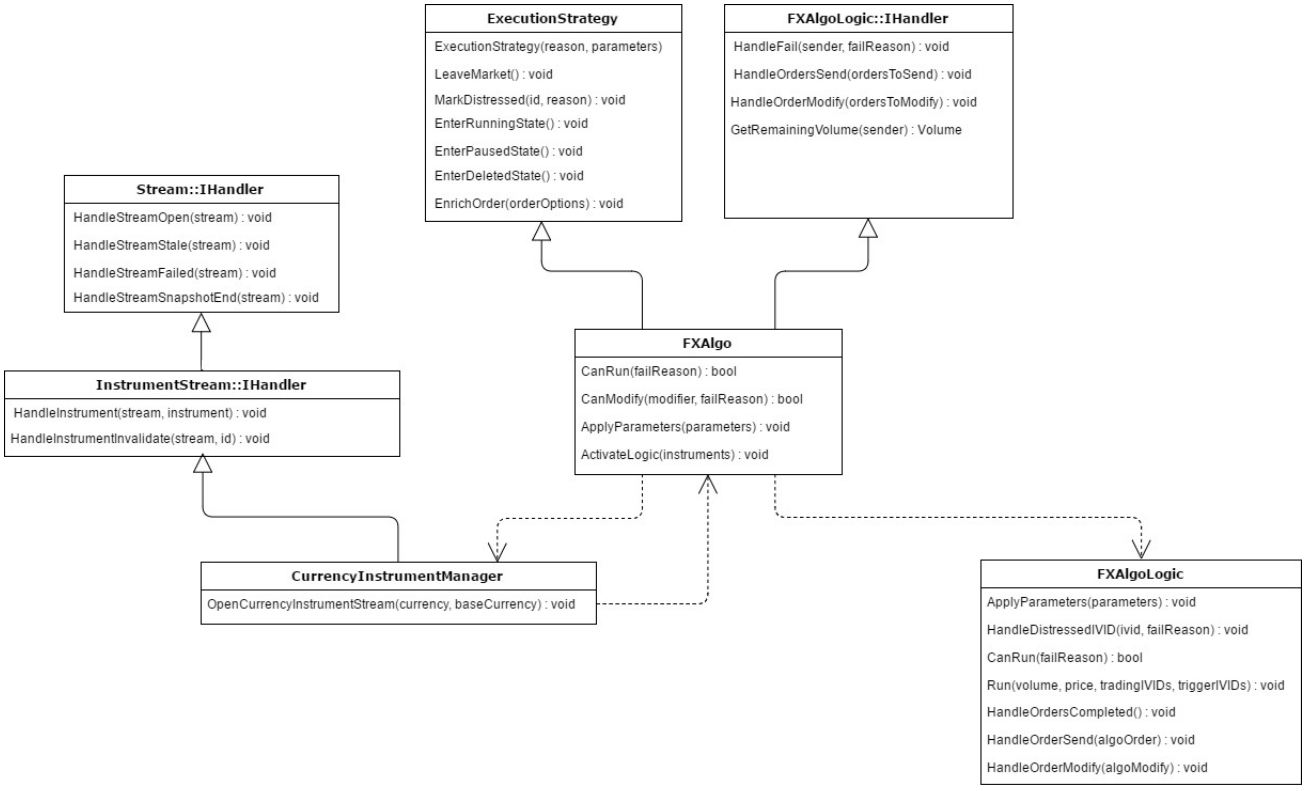

Рис. 5: Диаграмма классов, описывающая реализацию класса FXAlgo

струментов, определив фильтр на указанные базовую и котируемую валюты. После того, как удовлетворяющие критерию фильтра инструменты были получены, они попадают на проверку в FXAlgo. Для каждого инструмента проверяется:

- отсутствие флага “удален” (deleted);
- отсутствие флага Exclude in FX algo;

- наличие доступной торговой площадки, определенной пользователем.

Инструменты, прошедшие проверку, поступают на вход торговому сценарию, который должен реализовывать интерфейс *FXAlgoLogic*. Торговый сценарий начинает работу после вызова метода *Run*. Когда сценарий сформировал данные, достаточные для отправки заявки, он вызывает метод HandlerOrdersSend, передавая в *FXAlgo* необходимую информацию. В случае, если на каком-то этапе работы сценария возникает серьезная проблема, то он экстренно завершается, вызывая метод *HandleFail*. Когда *FXAlgo* исполнил все заказы, указанные торговым сценарием, то у сценария вызывается метод *HandleOrdersCompleted*, в котором принимается решение о том, готово ли торговое приложение завершиться, либо есть необходимость формировать данные для новых заявок.

5.3. Разработка торговых сценариев

Разработанные торговые сценарии, реализующие интерфейс FXAlgoLogic, можно разделить на три группы:

- торгующие на одном рынке;
- торгующие на нескольких рынках;
- торгующие и анализирующие разные рынки.

В первую группу попадают сценарии *Join* и *Iceberg*, которые отправляют заявки на выбранную торговую площадку. В отличие от них сценарии *Take* и *VWAP* анализируют набор торговых площадок и при формировании заявки выбирают рынок, на котором сделка пройдет с наибольшей выгодой. В третьей группе представлены сценарии *Peg*, *Stop Loss* и *Aggressive Watch*. В каждом из этих сценариев торговые площадки делятся на два множества. Первое множество площадок используется для того, чтобы анализировать цены и принимать решение о том, в какой момент необходимо отсылать заявку. Второе множество описывает рынки, на которые заявка может быть отправлена.

Для того, чтобы подробно рассмотреть этапы работы торговых сценариев, из каждой группы будет выбран один представитель и для него будет описана последовательность действий, характеризующая его логику.

5.3.1. Сценарий Iceberg

Торговый сценарий *Iceberg* (рис. 6) используется, когда пользователь хочет на конкретном рынке частями купить или продать определенный объем валютного инструмента. Для этого пользователь определяет не только суммарный объем сделки, но и

частичный объем через дополнительный параметр Disclosed volume. Торговый сценарий Iceberg будет последовательно отправлять заявки с частичным объемом до тех пор, пока весь суммарный объем не будет выкуплен. Отметим, что Iceberg отправляет новую заявку на рынок только после того, как предыдущая заявка была одобрена.

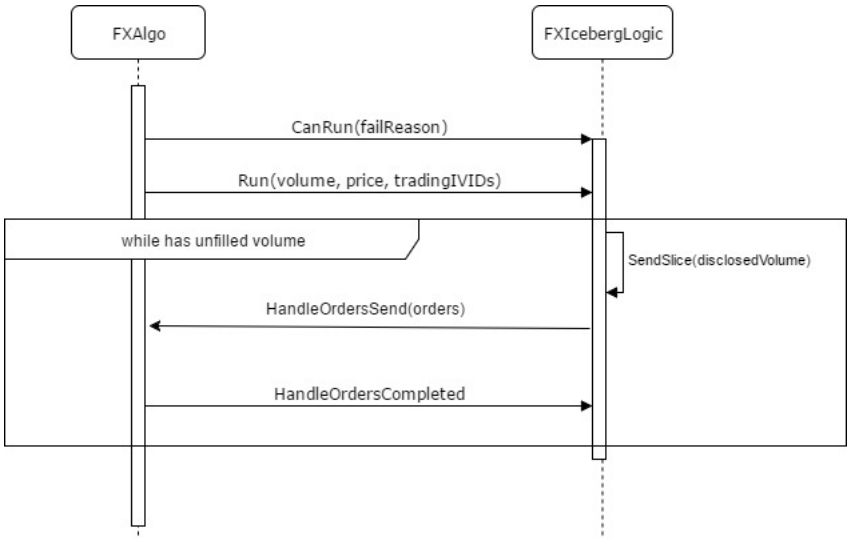

Рис. 6: Диаграмма последовательности для торгового сценария *Iceberg*

5.3.2. Сценарий VWAP

Торговый сценарий *VWAP* (рис. 7) представляет вторую группу и является более сложным. Его используют, когда необходимо провести сделку по валютному инструменту, используя несколько торговых площадок. Сценарий *VWAP* активно взаимодействует с агрегатором книг заявок валютных рынков, так как он предоставляет информацию о том, на какую торговую площадку лучше послать заявку. Сначала сценарий дожидается снимка потока данных, на основе которого формируется первый набор заявок. Данные по этим заявкам передаются в *FXAlgo*, который отправляет заявки на рынок. Если требуемый объем исполнен, то сценарий завершается, а если нет, то сценарий продолжает получать уведомления от агрегатора книг заявок. После получения обновления, которое свидетельствует о том, что можно провести сделку, сценарий заново формирует и отправляет заявки.

Для торгового алгоритма *VWAP* реализована специальная настройка. Данный алгоритм учитывает параметр *Optimistic approach*, который описывает то, как будет учитываться лимитная цена, переданная на вход. Так как алгоритм формирует набор заявок для разных торговых площадок, то вероятно, что эти заявки будут исполнены с разными ценами. Лимитная цена может учитываться для каждой заявки в отдельности, либо для совокупности заявок. В первом случае получается более жесткое условие, так как каждая заявка будет вынуждена удовлетворять лимитному ограничению.

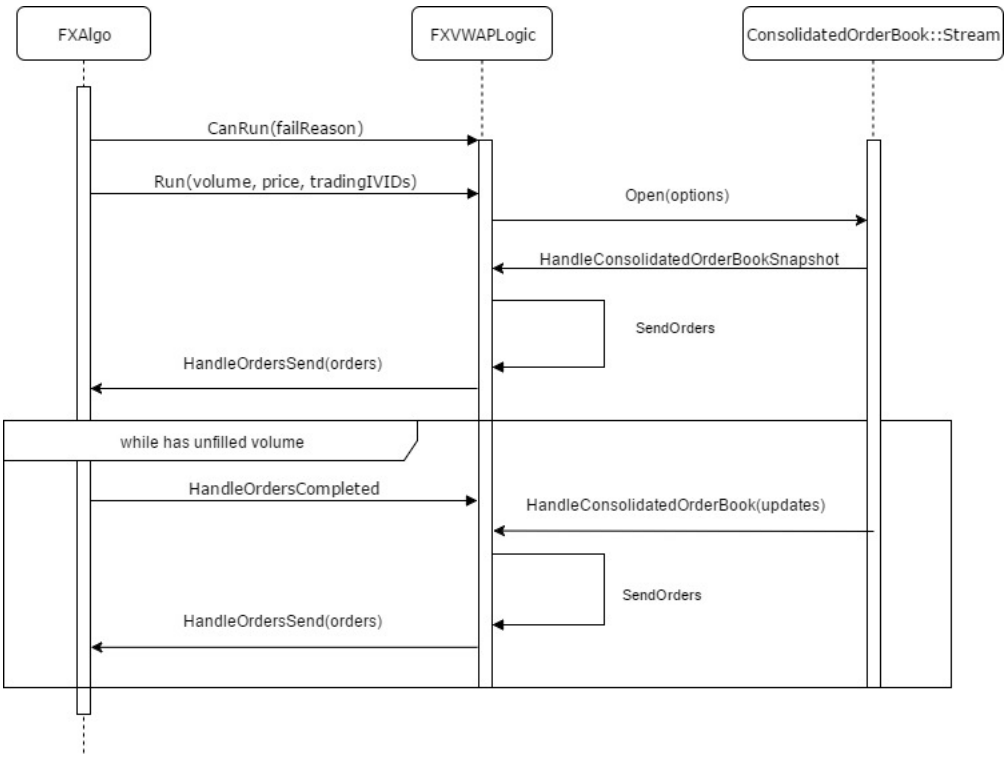

Рис. 7: Диаграмма последовательности для торгового сценария *VWAP*

5.3.3. Сценарий Aggressive Watch

Торговый сценарий Aggressive Watch (рис. 8) значительно отличается от сценариев, рассмотренных ранее. Для данного торгового алгоритма помимо списка рынков, где будет осуществляться торговля, передается торговая площадка, которая используется для анализа данных и принятия решения об отправке заявок. Отметим, что так же, как и в сценарии *VWAP*, данный алгоритм взаимодействует с агрегатором для формирования информации для будущих заявок. Но так как для принятия решения об отправке в *Aggressive Watch* используется отдельная торговая площадка, то стоит рассмотреть подробно ситуацию, когда сценарий принимает решение отправлять заявки. Для данного сценария был добавлен параметр *Trigger price*, который описывает цену для срабатывания. Это значит, что при достижении на анализируемом рынке этой цены, торговый сценарий отправляет заявки на площадки, подготовленные для торговли. Важно понимать, что от анализируемого рынка торговому алгоритму требуется только верхний уровень книги заявок. По этой причине сценарий *Aggressive Watch* использует более быстрый поток данных *BestPriceStream*, который поставляет только лучшие цены на покупку и продажу, представленные в книге заявок. Для сценария *Aggressive Watch* был добавлен альтернативный вариант срабатывания. Если пользователь указал, что торговый алгоритм должен отслеживать не лучшие цены, а поток открытых сделок, то сценарий исполнит заявки в момент, когда будет совершена сделка, цена которой достигает *Trigger price*. Для отслеживания открытых сделок

использовался поток *PublicTradeStream*.

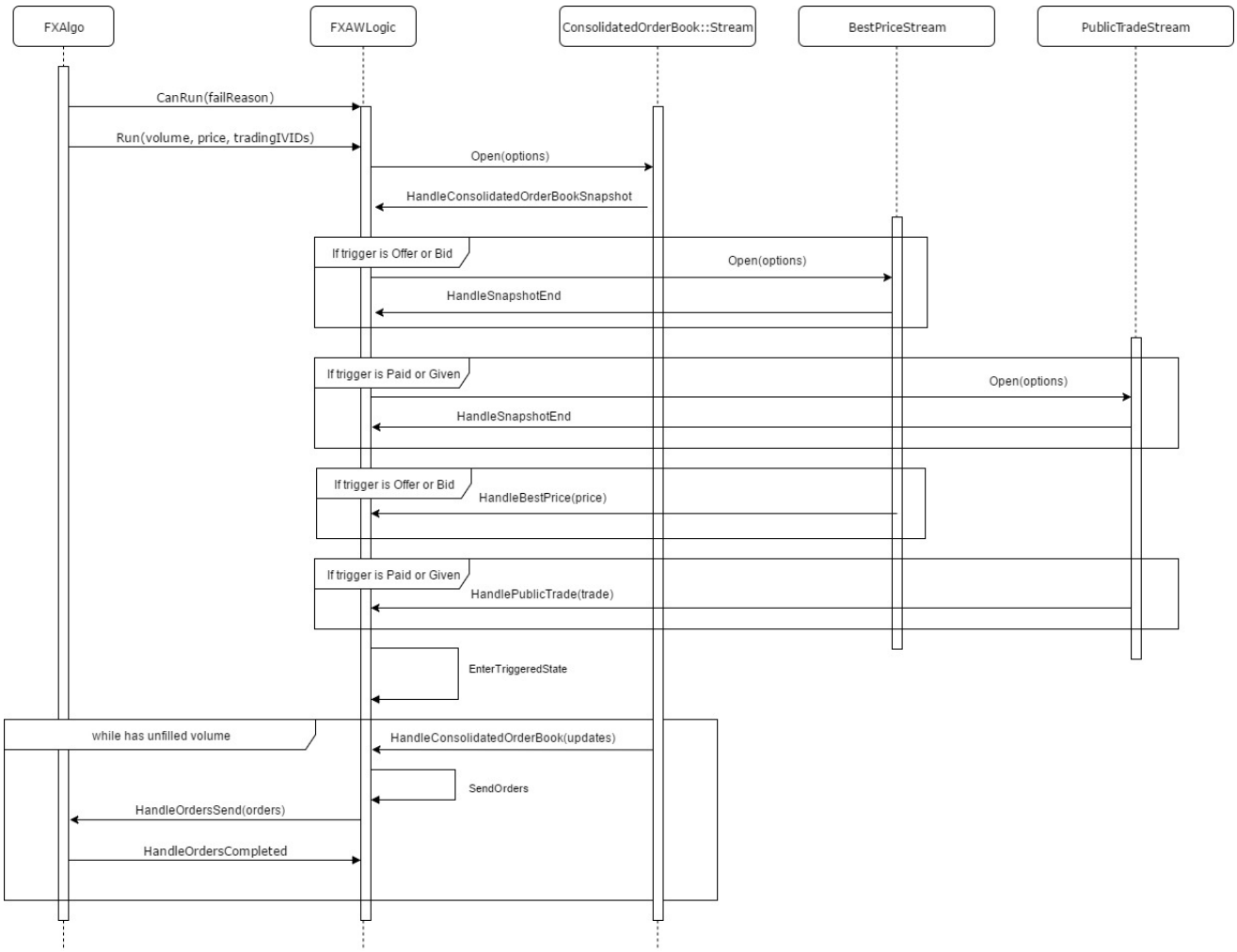

Рис. 8: Диаграмма последовательности для торгового сценария *Aggressive Watch*

В данном разделе были описаны три торговых сценария. Без подробного рассмотрения остались алгоритмы:

- Join;
- Take;
- Stop Loss;
- Peg.

Алгоритм *Join* (рис. 9) похож на алгоритм *Iceberg*, только там отсутствует параметр *Disclosed volume*. Это значит, что сценарий *Join* отправляет только одну заявку на рынок и дожидается ее исполнения.

Алгоритм *Take* (рис. 10) остался без внимания, так как он является, по сути, реализацией алгоритма *VWAP*, которая отправляет заявки на рынок один раз, если

они удовлетворяют указанной лимитной цене. Важно отметить, что именно в сценарии *Take* был реализован метод, который анализирует данные, полученные из агрегатора, и заполняет параметры будущих заявок. Данный метод переиспользуется в алгоритмах *VWAP*, *Aggressive Watch*, *Stop Loss* и *Peg*.

Сценарий *Stop Loss* (рис. 11) отличается от *Aggressive Watch* только условием срабатывания, поэтому диаграмма последовательности для него схожа с диаграммой для *Aggressive Watch*.

Алгоритм *Peg* (рис. 12) создает два агрегированных потока данных, один из которых использует для анализа, а другой для торговли.

В заключение, хотелось бы отметить, что реализованное приложение с представленным списком сценариев является полноценным решением, которое можно использовать для успешной торговли на валютном рынке. Созданные сценарии являются расширением стандартных торговых алгоритмов, представленных в системах [3] и [8], так как дают возможность отделять рынки, используемые для анализа, от рынков, используемых для торговли. Такое разделение позволило использовать подходящие по логике потоки данных (*BestPriceStream*, *PublicTradeStream*), что положительно отразилось на производительности решения. Разработанное приложение было размещено в основной ветке приложений *Tbricks* и в будущем будет использоваться как часть продукта.

6. Тестирование

Процесс тестирования программного продукта является неотъемлемой частью жизненного цикла современного программного обеспечения. Особенно тщательно должны быть протестированы решения, используемые для автоматической торговли на электронных биржах, так как цена ошибки программного продукта в данной сфере очень высока. Описываемый в данной работе модуль для агрегации книг заявок валютных рынков предполагается использовать при разработке пользовательских торговых приложений. Это значит, что наряду с корректно функционирующим агрегатором клиент должен получить возможность самостоятельно тестировать разработанную бизнес-логику. Для удовлетворения данного требования в компании *Itiviti* был разработан фреймворк *TestEngine*, предназначенный для тестирования *Tbricks* приложений. В данном разделе будет рассматриваться использование фреймворка *TestEngine* для тестирования сценариев исполнения приложения для автоматической отправки заявок на рынок. Кроме этого, будут рассмотрены модульные тесты для агрегатора книг заявок, реализованные при помощи [10]. Отдельное внимание будет уделено тестированию производительности агрегатора, основанное на размещении меток для сбора статистики и последующей визуализации данных при помощи платформы [9].

6.1. Тестирование торговых сценариев с помощью фреймворка TestEngine

6.1.1. Описание фреймворка TestEngine

Фреймворк TestEngine разрабатывался для того, чтобы дать возможность разработчикам торговой платформы Tbricks средство для проверки состояния приложения на разных этапах исполнения. Так как на поведение приложения оказывают влияние действия со стороны внешних сервисов, которые, например, уведомляют об изменении состояния рынка, возникла необходимость создать набор виртуальных сервисов, эмулирующих поведение основных сервисов. С помощью виртуальных сервисов создается полноценное окружение, в рамках которого тестируемое приложение может функционировать так же, как и в развернутой торговой системе. Это значит, что виртуальные сервисы дают возможность описать:

- торговые площадки;
- ограничения торговых площадок;
- рыночные данные;
- рыночные инструменты;

- пользовательские настройки.

На этапе тестирования после того, как тестовое окружение было успешно сконфигурировано, виртуальные сервисы начинают обрабатывать запросы, которые посылает проверяемое приложение. Для каждого запроса формируется ответ, использующий информацию, внесенную в виртуальные сервисы на начальном этапе, и передается в тестируемое приложение. Подразумевается, что в момент получения сообщения от виртуального сервиса сценарий тестируемого приложения проверит, что содержимое ответа совпадает с ожиданиями, а в случае расхождения данных оповестит об ошибке.

Для работы с приложениями виртуальные сервисы предоставляют API, где основными являются следующие методы:

- LoadPlugin — загружает скомпилированное приложение в сервис;
- LoadMetadata — загружает в сервис метаданные, которые используются независимо от тестируемого приложения;
- CreatePlugin — запускает загруженное приложение, инициализируя требуемые параметры;
- RunPlugin, PausePlugin, DeletePlugin — переводит приложение в одно из состояний (Running, Paused, Deleted);
- WaitPluginCondition — ожидает действие приложения, которое задается через фильтр;
- GetPluginHistory — возвращает протокол событий, который описывает действия приложения после запуска.

Представленные методы используются при написании тестового сценария.

Для того, чтобы описать требуемое тестовое окружение, необходимо создать два конфигурационных файла:

- конфигурация системы;
- конфигурация сервиса.

Конфигурация системы — это XML файл, в котором описано, какие сервисы, торговые площадки и инструменты будут использоваться приложением. Данная информация будет проанализирована, и требуемая функциональность будет загружена в виртуальный сервис на этапе запуска тестового сценария. Пользователю остается проинициализировать связанные с загружаемыми элементами параметры и атрибуты при запуске своего теста.

Конфигурация сервиса — это XML файл, в котором указано, с какими настройками будет запускаться виртуальный сервис. В данном файле указывается то, какое

название будет присвоено сервису, где будет размещаться связанный с сервисом журнал и какой уровень логирования будет использовать данный виртуальный сервис.

Достаточно создать конфигурационные файлы под одно приложение, чтобы потом их использовать для каждого связанного с приложением тестового сценария.

6.1.2. Создание тестового сценария

Создание тестового сценария для приложения торговой платформы *Tbricks* с использованием фреймворка *TestEngine* начинается с указания директории, где размещается скомпилированное приложение. Кроме этого, требуется указать путь до конфигурационных файлов, необходимых для описания тестового окружения. После этого приложение может быть запущено и протестировано в тестовом окружении, созданном при помощи виртуальных сервисов.

Важно понимать, что элементы, загруженные в сервис согласно конфигурационному файлу сервиса, должны быть проинициализированы. Так, для тестируемого приложения для автоматической торговли требуется загрузить несколько валютных инструментов, размещенных на разных торговых площадках, но имеющих одинаковые базовые и котируемые валюты.

Получается, что для различных тестовых сценариев можно выделить блоки кода, которые будут всегда использоваться. Зачастую такие блоки описывают:

- создание пользовательских настроек;
- инициализацию атрибутов инструментов;
- импорт рыночных данных;
- импорт требований торговых площадок;
- инициализацию параметров по умолчанию.

Переиспользуемые блоки кода выносят в общий код, который специфичен для конкретного тестируемого приложения. Например, для рассматриваемого торгового приложения в общий код был вынесен метод, инициализирующий входные параметры торгового приложения (листинг 6).

Листинг 6: Инициализация входных параметров приложения

```
StrategyParameters PluginParameters(const Volume& volume, const Price& price,
                                    const Side& side, const Integer& algoType)
{
    StrategyParameters parameters;
    parameters.SetParameter(fx_algo::Side(), side);
    parameters.SetParameter(fx_algo::Currency(), GetCurrency());
    parameters.SetParameter(fx_algo::BaseCurrency(), GetBaseCurrency());
```

```
    parameters.SetParameter(fx_algo::Instrument(), ID::m_instrument);
    parameters.SetParameter(fx_algo::TotalVolume(), volume);
    parameters.SetParameter(fx_algo::FX_AlgoVenues(), GetTradingTable());
    parameters.SetParameter(fx_algo::FX_AlgoTriggerVenues(), GetTriggerTable());
    parameters.SetParameter(fx_algo::Price(), price);
    parameters.SetParameter(fx_algo::FX_AlgoType(), algoType);

    return parameters;
}
```

Подразумевается, что для формируемых тестовых сценариев основное внимание будет уделяться таким параметрам как цена, объем, тип действия и тип торгового сценария. По этой причине валютную пару и список допустимых площадок можно зафиксировать.

Тестовый сценарий начинается с вызова общего кода. Когда приложение успешно загружено в сервис и переведено в состояние Running, стоит ожидать события, которые можно проверить. У приложения для автоматической торговли логично проверять параметры отправляемых на рынок заявок. Кроме этого, виртуальные сервисы дают возможность проверить поведение приложения при возникновении исключительных ситуаций. Далее на примере торгового алгоритма *Take* будут рассмотрены три базовых сценария поведения торговой стратегии.

Первый рассматриваемый сценарий проверяет ситуацию, когда на рынке представлены торговые площадки *XSTO* и *CHIX*. Для них сформированы следующие книги заявок, представленные на таблицах 11 и 12.

Таблица 11: Тестовая книга заявок для площадки *XSTO*

Покупка	Продажа	Объем
10.01	10.00	1000
10.05	9.98	2000
10.06		3000

Таблица 12: Тестовая книга заявок для площадки *CHIX*

Покупка	Продажа	Объем
10.01	9.99	1000
10.04875	9.96	2000
10.068125		4000

Приложение запускает торговый сценарий *Take* с требуемым на покупку объемом 2000 и лимитной ценой 10.01. Проверяется, что алгоритм создал две заявки, каждая из которых имеет цену 10.01 и объем 1000 (листинг 7).

Листинг 7: Проверка отправки заявок на торговые площадки

```
//  Verify that orders 1000@10.01 are sent to XSTO and CHIX
auto orderXSTO = XSTOVenue.WaitOrderCreateRequest();
auto orderCHIX = CHIXVenue.WaitOrderCreateRequest();

ASSERT_ORDER_VENUE(orderXSTO, ID::m_XSTOVenue);
ASSERT_ACTIVE_VOLUME(orderXSTO, Volume(1000));
ASSERT_ORDER_PRICE(orderXSTO, Price(10.01));

ASSERT_ORDER_VENUE(orderCHIX, ID::m_CHIXVenue);
ASSERT_ACTIVE_VOLUME(orderCHIX, Volume(1000));
ASSERT_ORDER_PRICE(orderCHIX, Price(10.01));
```

Данные заявки одобряются, и виртуальный сервис отсылает ответ в приложение (листинг 8).

Листинг 8: Формирование ответа на заявки

```
// Fill orders
FillOrder(orderXSTO, Volume(1000));
XSTOVenue.UpdateOrder(orderXSTO);
FillOrder(orderCHIX, Volume(1000));
CHIXVenue.UpdateOrder(orderCHIX);
```

После этого проверяется, что исполненный объем равен требуемому объему, и то, что стратегия успешно завершается (листинг 9).

Листинг 9: Проверка параметров приложения

```
// Check that filled volume = 2000
te.WaitPluginCondition(pluginUuid, StrategyByParameterFilter(
    fx_algo::FilledVolume(), Volume(2000)));

// Verify that plugin entered "DELETED" state
te.WaitPluginCondition(pluginUuid,
    StrategyByStateFilter(StrategyState::DELETED));
```

Второй сценарий в аналогичных условиях проверяет, что в случае, если одна заявка была одобрена, а вторая нет, то алгоритм *Take* все равно успешно завершится. В данном листинге представлен код, который оповещает приложение о том, что заявка на торговую площадку *XSTO* не была одобрена, а заявка на площадке *CHIX* была успешно обработана (листинг 10).

Листинг 10: Отклонение заявки виртуальным сервисом

```
// Reject order 1000@10.01 to XSTO
orderXSTO.SetOrderTransactionState(TransactionState::FAIL);
orderXSTO.SetStatusText("Rejects order!");
orderXSTO.SetDeleted(true);
XSTOVenue.UpdateOrder(orderXSTO);
```

```
// Fill order on CHIX
FillOrder(orderCHIX, Volume(1000));
CHIXVenue.UpdateOrder(orderCHIX);
```

В данноме тесте проверяется, что итоговый одобренный объем равен 1000 и что приложение успешно завершилось (листинг 11).

Листинг 11: Проверка параметров приложения

```
// Check that filled volume = 1000
te.WaitPluginCondition(pluginUuid, StrategyByParameterFilter(
    fx_algo::FilledVolume(), Volume(1000)));

// Verify that plugin entered "DELETED" state
te.WaitPluginCondition(pluginUuid,
    StrategyByStateFilter(StrategyState::DELETED));
```

Третий сценарией вводит ограничения на минимальный объем для торговой площадки *XSTO*, который равен 500. Получается, что данная торговая площадка не будет обрабатывать заявки, где указан объем меньше 500. Алгоритм *Take* должен корректно обработать данное ограничение (листинг 12).

Листинг 12: Установка ограничения на минимальный объем

```
// Sets minimum volume for XSTO venue (minimum volume = 500)
test::InstrumentTradingInformation tradingInfo;
tradingInfo.SetMinimumVolume(Volume(500));
te.GetTRD(FXAlgoConfiguration::ID::m_XSTOVenue)
  .CreateInstrumentTradingInformation(VIID::m_XSTO, tradingInfo);
```

Чтобы проверить описанный сценарий, приложение запускается на продажу с объемом 100 и лимитной ценой 9.98. Получается, что оптимально данный объем должен исполниться на площадке *XSTO* по цене 10.00, но данная площадка не обрабатывает заявки с объемом менее 500. Поэтому код теста проверяет, что торговое приложение послало заявку на площадку *CHIX* с ценой 9.99.

Листинг 13: Проверка заявки для площадки *CHIX*

```
// Verify that order 100@9.99 is sent to CHIX
auto orderCHIX = CHIXVenue.WaitOrderCreateRequest();

ASSERT_ORDER_VENUE(orderCHIX, ID::m_CHIXVenue);
ASSERT_ACTIVE_VOLUME(orderCHIX, Volume(100));
ASSERT_ORDER_PRICE(orderCHIX, Price(9.99));
```

Заявка исполняется, и тест завершается после успешной проверки на обработанный объем.

Аналогичные тестовые сценарии были созданые для каждого реализованного торгового сценария. Созданные тестовые сценарии отличались друг от друга, так как

учитывали и проверяли особенности, характерные только для отдельно взятого торгового сценария. Всего с помощью фреймворка *TestEngine* было реализовано 33 торговых сценария:

- 11 для *Take*;
- 5 для *Join*;
- 4 для *Peg*;
- 4 для *VWAP*;
- 3 для *Iceberg*;
- 3 для *Aggressive Watch*;
- 3 для *Stop Loss*.

Алгоритму *Take* было уделено особое внимание, так как логика, реализованная в нем, используется в сценариях *VWAP*, *Peg*, *Aggressive Watch*, *Stop Loss*.

6.2. Модульное тестирование агрегатора с помощью фреймворка Google Test

Агрегатор книг заявок валютных рынков является основой реализованного приложения по автоматической отправке заявок. На основе реализованного агрегатора пользователи будут разрабатывать новую бизнес-логику. Сложность тестирования агрегатора заключается в том, что для того, чтобы полностью проверить возможности агрегатора, необходимо использовать виртуальные сервисы. Сервисы дают возможность эмулировать потоки данных и получать требования торговых площадок, которые учитываются агрегатором. Текущая реализация фреймворка *TestEngine* заточена под приложение, поэтому для тестирования агрегатора пришлось бы создавать отдельный плагин. Поэтому было принято решение проверить часть функциональности агрегатора на этапе тестирования сценариев торгового приложения. По этой причине проверки параметров заявок, создаваемых торговым приложением, можно считать тестированием функциональности агрегатора.

Отдельного внимания заслуживает логика, реализующая вычисление технического индикатора VWAP в агрегаторе книг заявок. Данная логика может быть протестирована без использования виртуальных сервисов. Для нее необходимо предоставить книги заявок, которые можно задать непосредственно в тесте. Далее для запрашиваемого объема можно проверять результат вычисления индикатора VWAP.

Можно отметить, что при вычислении индикатора VWAP (без учета требований торговых площадок) используются только входные данные и код агрегатора. Поэтому было принято решение реализовать набор модульных тестов с использованием популярного фреймворка [10].

Для тестирования были сформированы 4 варианты книг заявок, которые могут быть интерпретированы как лимитные книги, так и книги с ценовым диапазоном. Далее в тестовом сценарии несколько книг заявок передавались в агрегатор, где вычислялись значения индикатора VWAP для разных объемов. Всего было реализовано 18 тестов, в каждом из которых проверялись вычисленные VWAP значения на покупку и продажу.

6.3. Тестирование производительности агрегатора книг заявок

Прежде чем переходить к тестированию производительности агрегатора книг заявок, необходимо обратить внимание на то, с какими объемами данных будет работать агрегатор. Основными параметрами, которые стоит учитывать, являются:

- глубина книги заявок;
- количество агрегируемых инструментов;
- частота обновлений книги заявок.

Глубина книги заявок описывает количество уровней в книге заявок. Можно считать, что глубина произвольной книги, полученной от торговой площадки, не будет превышать 50. Параметр, характеризующий количество агрегируемых инструментов, зависит от того, какое количество торговых площадок поддерживается в системе. В системе *Tbricks* планируется поддержать около 40 торговых площадок. Отметим, что чаще всего клиенты агрегируют данные не более чем по 10 валютным инструментам. Частота обновлений книги заявок зависит непосредственно от торговой площадки. Нужно быть готовым к тому, что для одной торговой площадки обновления будут поступать чаще, чем 100 раз в секунду.

Чтобы протестировать производительность реализованного агрегатора, было принято решение расширить уже существующий нагрузочный сценарий, который работал с валютной парой *EUR/USD*. Данный сценарий использовал 4 валютных инструмента, которые теперь отслеживаются через реализованный модуль для агрегации. На фиксированную торговую площадку отправлялись новые заявки с частотой 50 раз в секунду, которые являлись причиной генерации обновлений для рассматриваемой книги заявок. На этапе получения рыночного обновления и на этапе, когда агрегатор завершал формирование данных по сделкам, были поставлены временный метки. Статистика по временным меткам была собрана при помощи платформы [9]. Были получены результаты, представленные в таблице 13.

Таблица 13: Результаты измерений производительности агрегатора

Перцентиль	Время (мкс)
99%	65
95%	62
80%	54
50%	47

Стандартное отклонение можно считать равным 2 микросекундам.

Заключение

В рамках данной работы были достигнуты следующие результаты.

1. Реализован модуль для агрегации книг заявок, поддерживающий возможность одновременной обработки нескольких потоков рыночных данных. В модуле была реализована корректная работа с книгами заявок разных типов, обработка ограничений торговых площадок и пользовательских настроек, а также вычисление технического индикатора VWAP.

2. Разработано приложение для автоматической торговли на валютном рынке, использующее модуль для агрегации книг заявок. Для приложения было создано 7 различных торговых сценариев, использующих торговые площадки для анализа данных и совершения сделок.

3. Выполнено тестирование модуля для агрегации книг заявок и приложения для автоматической торговли:

   - реализовано 33 тестовых сценарий с использование фреймворка TestEngine;
   - реализовано 18 модульных тестов с использованием фреймворка Google Test;
   - произведено тестирование производительности агрегатора.

Приложения

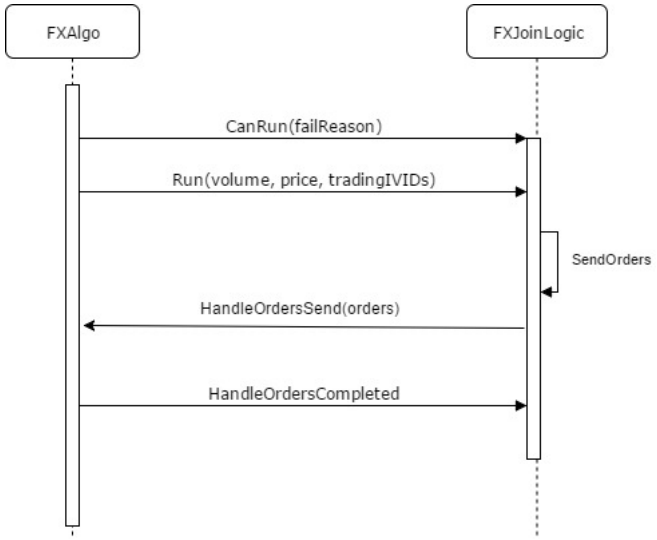

Рис. 9: Диаграмма последовательности для торгового сценария *Join*

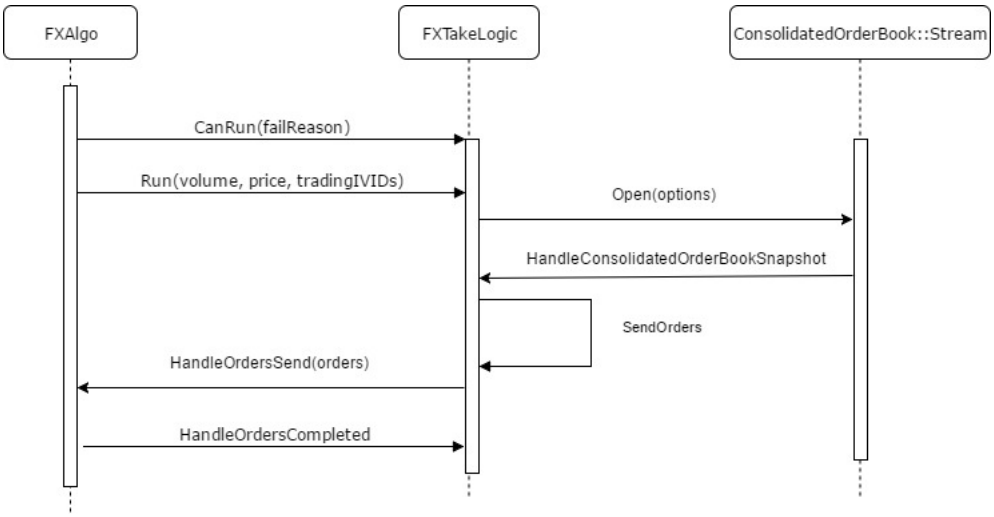

Рис. 10: Диаграмма последовательности для торгового сценария *Take*

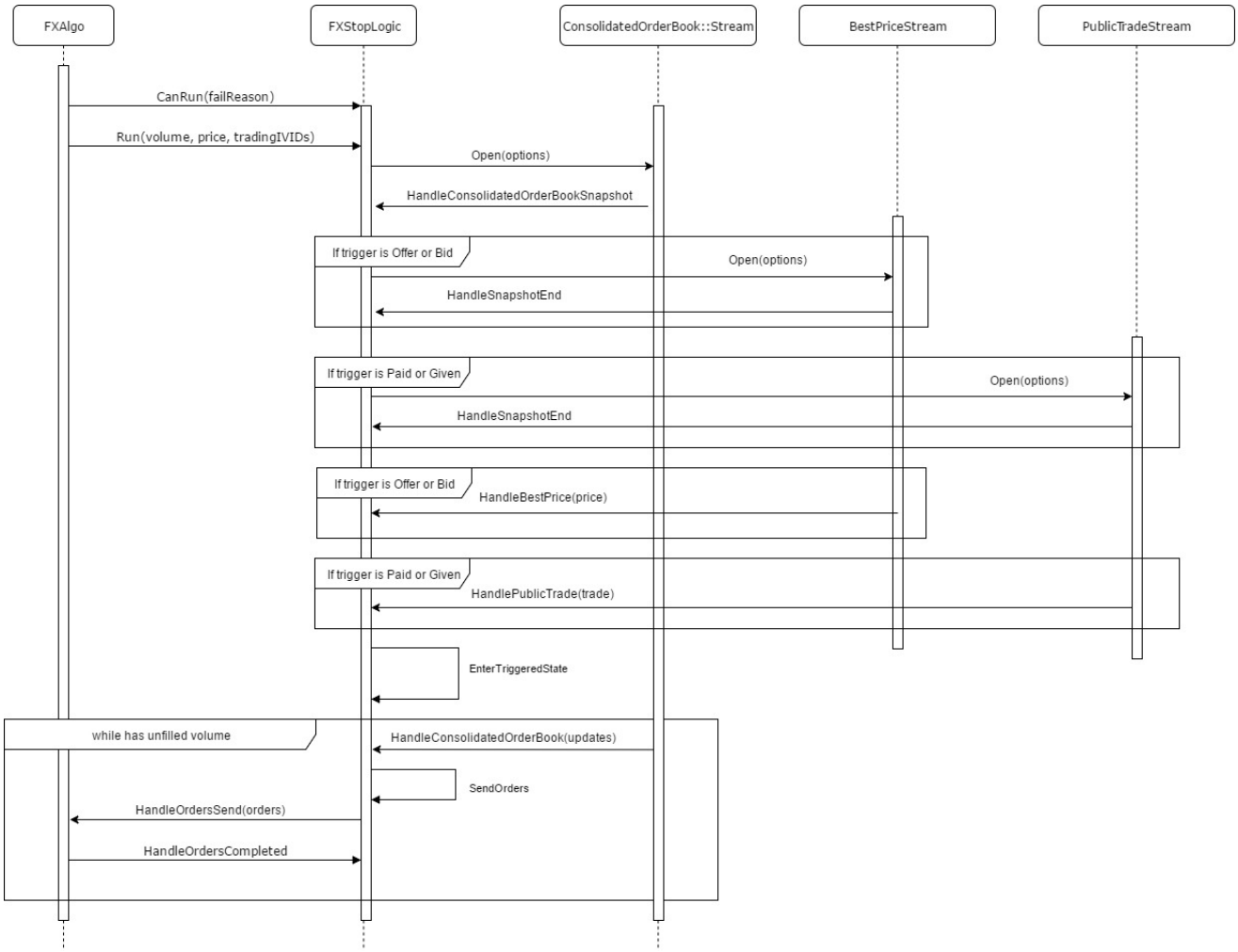

Рис. 11: Диаграмма последовательности для торгового сценария *Stop Loss*

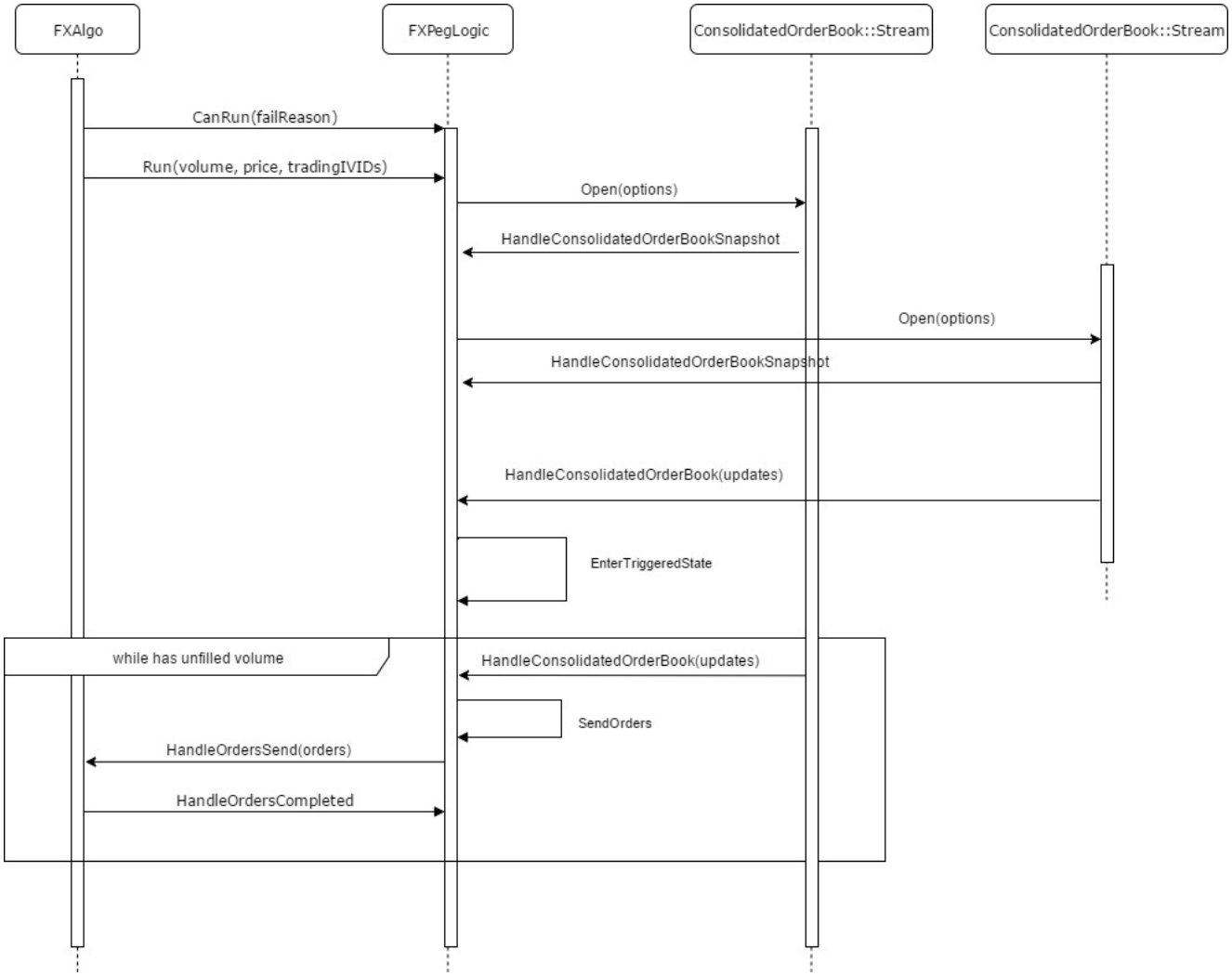

Рис. 12: Диаграмма последовательности для торгового сценария *Peg*

Список литературы

[1] Bank for International Settlements. Foreign exchange turnover in April 2016. — 2016. — URL: http://www.bis.org/publ/rpfx16fx.pdf (дата обращения: 12.04.2017).

[2] Competitive Algorithms for VWAP and Limit Order Trading / Sham M. Kakade, Michael Kearns, Yishay Mansour, Luis E. Ortiz. — ACM, 2004. — С. 189–198. — URL: http://doi.acm.org/10.1145/988772.988801 (дата обращения: 10.03.2017).

[3] FX aggregator. Создание систем для профессиональной торговли на валютном рынке. — URL: http://www.fx-aggregator.ru (дата обращения: 25.03.2017).

[4] Itiviti. Разработка финансового программного обеспечения и решений для работы на электронных биржах. — URL: http://www.itiviti.com (дата обращения: 15.02.2017).

[5] Madhavan Ananth. VWAP Strategies. — Spring 2002, Vol. 2002, 2002. — URL: http://www.smallake.kr/wp-content/uploads/2016/03/TP_Spring_2002_Madhavan.pdf (дата обращения: 10.03.2017).

[6] Mancini Loriano, Ranaldo Angelo, Wrampelmeyer Jan. Liquidity in the Foreign Exchange Market: Measurement, Commonality, and Risk Premiums. — Journal of Finance, 68 (5). 1805-1841, 2013. — URL: https://www.snb.ch/n/mmr/reference/working_paper_2010_03/source/working_paper_2010_03.n.pdf (дата обращения: 17.03.2017).

[7] Pengyuan Shao Barret, Frank Greg. Aggregation of an FX order book based on complex event processing. — Investment management and financial innovations, 9 (1), 2013. — URL: ftp://ftp.ams.sunysb.edu/papers/2011/susb_qf_11_01.pdf (дата обращения: 10.02.2017).

[8] Soft FX. Разработка решений для форекс бизнеса, позволяющих эффективно управлять ликвидностью и исполнением. — URL: https://www.soft-fx.com (дата обращения: 26.03.2017).

[9] Tableau. Программное обеспечение для визуализации данных. — URL: https://www.tableau.com (дата обращения: 10.05.2017).

[10] Test Google. Библиотека для модульного тестирования на языке C++. — URL: https://github.com/google/googletest (дата обращения: 15.03.2017).