Санкт-Петербургский Государственный Университет

Кафедра компьютерных технологий и систем

Крупышев Иван Сергеевич

Выпускная квалификационная работа бакалавра

Разработка системы автоматической проверки правильности кода

Направление 010300

Фундаментальные информатика и информационные технологии

Научный руководитель,
доцент
Лепихин Т. А.

Санкт-Петербург
2016

Оглавление

Введение

Ежегодно проводятся олимпиады по программированию среди школьников и студентов. Организация такого мероприятия неизбежно связана с обработкой большого количества информации. Большая часть действий является общей для всех подобных олимпиад: регистрация участников, тестирование их программ и подведение итогов. Все это можно осуществлять вручную, однако, это требует больших временных затрат, а также не исключает ошибки, вызванные человеческим фактором.

Ввиду описанных трудностей предложено использовать систему автоматической проверки правильности кода, систему управления олимпиадой. Такая система должна существенно облегчить обеспечение ряда функциональных возможностей при проведении подобных мероприятий, а именно:

- хранить базу участников, информацию о ходе соревнования;
- автоматически проверять корректность исходных кодов решения на наборе тестов;
- информировать участника о результате проверки его работы;
- вычислять и сохранять статистику соревнования.

Целью данной работы является разработка системы автоматической проверки правильности кода на примере проведения олимпиад по программированию.

Обзор литературы

Основная часть требуемых теоретических знания для реализации разрабатываемой системы была получена из следующих двух книг:

1. Троелсен Э. Язык программирования *C#* 5.0 и платформа .NET 4.5. В данной книге представлены общие сведения, требующиеся для разработки приложений на языке *C#*. Материал начинается с самых азов, таких как описание переменных, циклов, основных конструкций программирования, плавно переходя в более сложные темы. Автор рассматривает большое количество различных технологий и подходов, таких как *LINQ*, *WPF*, *WCF*, различные способы организации взаимодействия с базами данных, даже затрагивает тему веб-разработки используя *ASP.NET*. Ввиду такого большого объема аспектов программирования, представленных в рамках одной книги, не все они описаны достаточно подробно для эффективного их применения. Данная работа является хорошей отправной точкой, чтобы составить представление о языке *C#*, платформе *.NET Framework*, с дальнейшим углублением знаний по интересующей тематике, используя другие ресурсы.
2. Lowy J., Montgomery M. Programming WCF Services. 4th Edition. Данная книга специализируется исключительно на технологии *Windows Communication Foundation*, на основе которой реализовано сетевое взаимодействие в разрабатываемой системе. Один из авторов данной книги входит в число создателей *WCF*, ввиду чего он приводит не только практическое описание того, как использовать данную технологию, но и рассказывает о принципах, заложенных в тот или иной ее аспект. Эта книга представляет собой исчерпывающее описание платформы *WCF*, во многом избыточное для решения задачи сетевого взаимодействия в системе проведения олимпиад по программированию.

В качестве электронного ресурса необходимо выделить *Microsoft Developer Network* (https://msdn.microsoft.com). Данный веб-ресурс содержит огромное количество статей об использовании технологий, разработанных компанией *Microsoft*, представленных вместе с соответствующим описанием и примерами кода на различных языках программирования, включая *C#*. Большая часть статей находится в открытом доступе и предоставляется на безвозмездной основе.

Глава 1. Постановка задачи и обзор аналогов

1.1 Постановка задачи

Для достижения поставленной цели необходимо реализовать информационную систему автоматической проверки программного кода, которая удовлетворяет ряду требований и позволяет, в частности, осуществлять проведение мероприятий, связанных с функциональными возможностями разрабатываемой системы. Одним из таких мероприятий может быть олимпиада по программированию.

Разрабатываемая система должна иметь следующие функциональные возможности:

- создание сессии турнира;
- контроль времени сдачи задач в рамках сессии;
- подготовка компьютера участника к проведению олимпиады;
- регистрация участников;
- компилирование исходных кодов;
- тестирование скомпилированных решений;
- уведомление участника о результате тестирования его программы;
- подведение итогов и вычисление статистики.

1.2 Обзор аналогов

На рынке уже существует ряд решений для организации олимпиад по программированию. Существующие системы можно условно разделить на 3 категории:

1. Веб-ориентированные системы. Подавляющее большинство продуктов относится именно к этой категории. К ней относятся такие решения как *Google Code Jam*, *Ejudge*, *Contester*.
2. Клиент-серверные решения. На момент написания данной работы найти рабочего представителя системы данного типа в интернете не удалось.

3. Система, предполагающая ручное внесение кода программ. Данные решения являются устаревшими и неконкурентоспособными.

Представленные решения различаются списком доступных языков программирования, платформой, на которой они функционируют, поддержкой распределенного тестирования.

Многие из существующих продуктов являются закрытыми, применяются компаниями-владельцами для организации состязаний и не предоставляют возможности использовать их для организации собственной олимпиады.

Для реализации данной системы выбрана именно клиент-серверная модель, ввиду чего можно сказать, что у системы нет прямых аналогов, причины выбора данной концепции описываются далее в тексте работы.

Глава 2. Описание разрабатываемой системы

В данном разделе приводится общее описание разрабатываемой системы и ее внутренней организации. Детали практической реализации рассматриваются в следующем разделе.

2.1 Формат олимпиад

Реализуемая система рассчитана на очные и заочные практические мероприятия, связанные с программированием, в частности, проводимые в учебных заведениях. В единый момент времени могут проходить несколько сессий, ориентированные на различные категории участников. В тексте далее под мероприятием для простоты и удобства восприятия материала примем олимпиаду по программированию.

Возможны различные способы организации подобного рода мероприятий. В рассматриваемом случае предполагается следующий формат:

- Участнику выдается код, идентифицирующий олимпиаду, в которой он собирается принять участие. С помощью полученного кода происходит регистрация в системе и привязка человека к определенной сессии.
- Система содержит в себе список заданий для каждой сессии, которые передаются пользователю. Каждая задача имеет определенные параметры, помимо словесного описания: максимальный балл, максимальное время решения одного теста, максимальное количество попыток ее сдачи.
- По мере решения задач участник отправляет свои решения (исходные коды программ) на проверку, система информирует его о количестве набранных баллов.
- По истечении времени олимпиады сессия закрывается, подводятся итоги, и выбирается победитель на основании суммы набранных баллов. В случае если несколько участников набрали одинаковые

баллы, анализируется количество попыток сдачи, время отправления решений.

2.2 Общая концепция

Возможны многие варианты устройства информационной системы для автоматического проведения олимпиад, из них можно выделить три основные концепции:

1. веб-ресурс, принимающий исходные коды программ пользователей и информирующий их о результатах тестирования;
2. веб-среда разработки, позволяющая непосредственно писать код и отправлять его на проверку;
3. клиент-серверное приложение.

Первый способ ориентирован на самостоятельность участника в подготовке его ПК и на заочные состязания, при проведении очного тура олимпиады требуется предоставить участникам полностью готовое рабочее место.

Второй вариант является чрезвычайно трудоемким. Создание полноценной среды веб-разработки, поддерживающей различные языки программирования – задача, которую могут решить крупные компании, например, *Intel*, но не один человек. Кроме того, выбор такого подхода потребует гораздо более мощные ресурсы серверной части.

Наиболее подходящей концепцией в данном случае является последний вариант, клиент-серверная модель. Клиентское приложение, в отличие от веб-ресурса позволяет произвести автоматическую предварительную подготовку рабочего места участника перед началом олимпиады. Например, ограничить доступ участников к веб-ресурсам, что в рамках подобного мероприятия является важной особенностью.

2.3 Структура программного продукта

Разрабатываемая система представляет собой набор взаимосвязанных функциональных компонентов. Такое разделение повышает читабельность кода, формирует логичную структуру программы, в которой каждая группа связанных функций представляет собой обособленный элемент. Взаимодействие между большей частью этих элементов основано на интерфейсах, что позволяет гибко вносить изменения в код программы, так как нет привязки к конкретной реализации функционала.

Общая схема устройства продукта приведена на рис. 1.

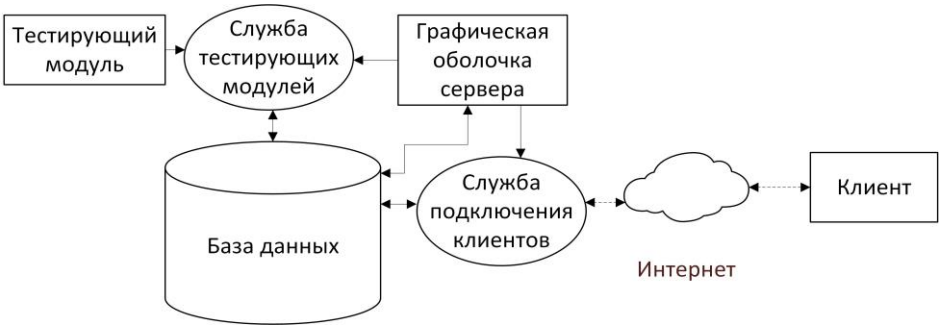

Рис. 1. Общая схема система.

Рассмотрим функциональность, за которую отвечает каждый из представленных модулей.

2.3.1 Клиент пользователя

В рамках реализуемого проекта клиент пользователя устанавливает на компьютер участника среды разработки программного обеспечения, перед началом олимпиады. Предоставляет графический интерфейс, с помощью которого участник:

- регистрируется в системе;
- получает список задач для решения;
- отправляет свои исходные коды на проверку;
- получает уведомления о результате тестирования его программы.

2.3.2 Графическая оболочка сервера

Посредством графической оболочки сервера производится вся необходимая конфигурация, запуск служб работы с клиентом и подключения тестирующих компонентов.

В начале администратор создает сессию олимпиады, для которой задается время начала и конца, код доступа, список задач, которые будут отправлены пользователям и список проверочных тестов.

Во время проведения олимпиады возможно наблюдать процесс сдачи программ и их тестирования.

2.3.3 Служба подключения клиентов

Сервис подключения клиентов отвечает за взаимодействие с пользователем. Функциональность службы сводится к получению запроса, выборке или загрузке информации из базы данных и передаче ее пользователю. Клиенты не имеют возможности напрямую обратиться к базе данных. Вся передаваемая информация проверяется в случае, если значения недопустимы или не удовлетворяют логике программы, служба передает клиенту соответствующее сообщение о возникшей исключительной ситуации.

2.3.4 Служба тестирующих модулей

Этот пункт является промежуточным звеном между тестирующим модулем и базой данных. Такой подход обусловлен тем, что он позволяет сделать систему распределенной, подключив тестирующие компоненты, расположенные на других узлах. Роль этой службы заключается в распределении исходных кодов программ пользователей, требующих проверки, между тестирующими компонентами. А также в получении информации о результате проверки с занесением ее в базу данных. При реализации такого подхода возможна проблема в случае, если один из тестирующих компонентов перестал отвечать (проблемы с оборудованием или сетью). Для корректного выхода из такой ситуации служба хранит информацию о том, какие задачи были переданы какому из компонентов. В

том случае, если тестирующий модуль не вернул результат проверки программ в течение определенного промежутка времени, задачи, которые он проверял, передаются другому тестирующему модулю.

2.3.5 Тестирующий модуль

Реализуемая система допускает наличие более чем одного экземпляра тестирующего модуля. Проверяющий компонент может быть размещен непосредственно на сервере или на другой машине, используя сеть для взаимодействия со службой подключения. Данный модуль получает список контрольных тестов для всех активных сессий олимпиад. Затем запрашивает исходные коды программ участников, которые необходимо проверить. Когда список программ получен, происходит поочередная их проверка, которая состоит из следующих этапов:

1. Создать файл, содержащий исходный код программы пользователя и вызвать компилятор, соответствующий языку данной программы.
2. Проверить результат компиляции, если программа скомпилирована успешно, то перейти к пункту 3. Иначе, записать ошибку и перейти к следующей программе в очереди.
3. Создать соответствующий входной файл (input), содержащий входные данные для тестируемой программы и запустить исполняемый файл, полученный на предыдущем этапе.
4. Если программа не завершилась в течение заданного времени, то принудительно завершить процесс и записать ошибку. Иначе, проверить результат, выданный программой.

Данный цикл повторяется до тех пор, пока все полученные коды не будут проверены, затем тестирующий модуль сообщает полученные результаты службе тестирующих компонентов.

Глава 3. Практическая реализация

В этом разделе описываются технические детали и особенности практической реализации системы.

3.1 Средства разработки

Для написания системы был выбран язык *C#* и платформа *.NET Framework 4.5* [1]. Выбор аргументирован тем, что ОС *Windows* широко распространена на территории РФ, а платформа *.NET Framework* является неотъемлемой частью современных операционных систем семейства *Windows* [2]. Среда разработки полностью удовлетворяет всем необходимым требованиям для решения поставленной задачи.

Основным средством разработки является *Microsoft Visual Studio 2015*. Указанная среда разработки обладает широкими возможностями как для написания кода, так и для работы с базами данных, разработки и тестирования служб. В пакет *Visual Studio* входит технология *IntelliSense* [3], которая помогает сильно ускорить процесс кодирования за счет предоставления быстрой всплывающей информации о параметрах метода, списке доступных членов и за счет автоматического завершения ввода слов.

3.2 Важные конструкции программирования

Ниже приводятся некоторые конструкции программирования, которые используются в данном проекте и могут быть не знакомы читателю.

3.2.1 Обобщённые коллекции

Начиная с версии платформы *.NET 2.0* язык *C#* был расширен функциональностью *обобщений*, содержащихся в пространстве имен *System.Collections.Generic*.

При создании программного продукта неизбежно требуется хранить и представлять данные в памяти. В случае, если данные имеют фиксированный размер допустимо использовать массив. Если количество хранимых элементов может меняться, то требуется контейнер, способный динамически изменять свой размер.

В ранних версиях платформы *.NET* для этих целей использовались типы из пространства имен *System.Collection*, например, *ArrayList*. Использование этих контейнеров приводит к генерации низко производительного кода, ввиду того, что данные типы прототипированы для работы с типами *System.Object*. Следовательно, при добавлении нового элемента в коллекцию требуется произвести *упаковку* (*boxing*), которая включает в себя размещения нового объекта в управляемой куче и передачу данных из стека в выделенное место. При извлечении данных необходима *распаковка* (*unboxing*), которая требует обратной последовательности действий. Кроме того, во время распаковки необходимо отслеживать исключения, которые могут возникнуть при приведении типов.

Для решения описанных выше проблем необходимо использовать обобщенные типы, например, *List<T>*, где *T – заполнитель*, указывающий с какими типами должна работать данная коллекция. После создания такой структуры данных, она будет прототипирована для работы с указанным типом, следовательно, операции упаковки/распаковки не нужны и совместимость типов будет проверена на этапе компиляции.

3.2.2 Лямбда-выражения

Использование лямбда-выражений сильно сокращает нотацию взаимодействия с делегатами.

Делегаты – объекты, указывающие метод или списки методов, описывающие аргументы и возвращаемые значения этих методов.

Лямбда-выражения представляют собой конструкции вида:

((Список аргументов) => (Обрабатывающие операторы))

Компилятор *C#* преобразует подобные выражение в анонимные методы, использующие соответствующий делегат.

Допустим, имеется коллекция объектов типа *Session: List<Session> listSession*. У объекта *Session* есть открытое свойство *AccessCode*, требуется найти объект, у которого *AccessCode* равен «2319». Для решения этой задачи, используя делегаты, потребуется написать следующий код:

Метод, который будет выполнять отбор

```
private static bool Compare(Session session)
{
	return session.AccessCode == "2319";
}
```

Затем, когда необходимо найти данный элемент

```
Predicate<Session> searchFilter = new Predicate<Session>(Compare);
Session findSession = listSession.Find(searchFilter);
```

Используя лямбда-выражения можно уменьшить объем данного кода:

```
Session findSession = listSession.Find((m)=>(m.AccessCode=="2319"));
```

3.2.3 Неявная типизация и анонимные типы

Помимо классического способа задания типа язык *C#* предлагает возможность неявной типизации, используя ключевое слово *var*. Полученная таким образом переменная является строго типизированной, компилятор определяет ее автоматически за счет присвоенных ей значений.

Например:

```
var intType = 5;
```

Другим сходным средством является возможность компилятора генерировать анонимные типы, для быстрого моделирования данных определенной структуры, на основе набора пар имя/значение, без ручного создания класса, представляющего данный тип.

Например:

```
var SomeSession = new
{ AccessCode = "2319",
  Title = "Session #1"};
```

Класс, описывающий данную структуру данных, будет сгенерирован автоматически компилятором. Кроме того, если инициализировать другой объект, описав такую же сигнатуру, то они будут принадлежать к одному

классу. Данная функциональность исключительно необходима при работе с *LINQ*.

3.2.4 LINQ

LINQ (*Language Integrated Query*) – язык интегрированных запросов, является современным способом оперирования данными, которые могут содержаться в различных источниках. *LINQ* применим к коллекциям *.NET Framework*, базам данных, XML-документам. Выражения *LINQ* являются строго типизированными, компилятор проверяет корректность выражений на этапе компиляции.

В качестве результата выполнения запроса, может выступать огромное множество различных типов данных, возможно создание нового типа, представляющего результат запроса на этапе компиляции. Ввиду чего для оперирования запросами *LINQ* требуется неявная типизация и анонимные типы.

Существуют две формы записи выражений *LINQ*, в виде синтаксиса запросов и синтаксиса методов.

- Синтаксис запросов использует ключевые слова *select*, *from*, *where*, и так далее. Что делает этот способ похожим на выражения SQL. Пример выражения *LINQ* в форме запроса для выборки элементов из коллекции, имеющих свойство *AccessCode* равное 2319.

```
var findSession =  from m in listSession where m.AccessCode=="2319"
select m;
```

- Синтаксис методов — это использование стандартной точечной нотации *C#* для вызова методов. Реализуем такую же выборку, что и в примере выше, используя синтаксис методов и лямбда выражение:

```
var findSession = listSession.Where((m) => (m.AccessCode == "2319"));
```

Использование обоих форм записи приводит к генерации одинакового кода после компиляции. Однако синтаксис запросов обладает меньшими

возможностями относительно подхода с использованием вызова методов. Ввиду чего в данной работе используется второй вариант оформления выражений *LINQ*.

3.3 Хранилище данных

Основным способом хранения информации в разрабатываемой системе является база данных. Она относится к серверной части и напрямую с ней взаимодействуют лишь служба подключения пользователей, служба тестирующих компонентов и графический интерфейс серверной части. В качестве СУБД выбрана *Microsoft MSSQL*.

Помимо информации, содержащейся в базе данных, система работает с XML файлами, содержащими описание сессии олимпиады. Для создания и редактирования данных файлов написано специальное приложение с графическим интерфейсом. Вынос данной функциональности в отдельную программу сделан для того, чтобы можно было работать со списками задач и тестов без развертывания всего серверного приложения.

3.3.1 Способы взаимодействия с базами данных

Функциональность взаимодействия с данными в среде *.NET* предоставлена в библиотеках *ADO.NET*. Выделяют три концептуально различных способа взаимодействия с базами данных в *ADO.NET*:

- *Подключенный уровень.* Данный способ подразумевает ручное создание подключения к базе данных с помощью объектов подключения. Создание SQL запросов, выполнение их с помощью объектов команд и чтение данных с помощью поставщика данных. Очевидным недостатком такого подхода является отсутствие строгой типизации, так как компилятор не может проверить правильность построения запроса на этапе компиляции. Чтобы частично решить эту проблему необходимо строить собственную библиотеку доступа к данным, открывающую взаимодействие в строго формализованном виде. Такой подход требует написания

большого количества шаблонного кода, в котором велика вероятность возникновения ошибок. Данный способ хорошо подходит для очень простых приложений.

- *Автономный уровень.* Суть этого подхода заключается в моделировании копии базы данных в памяти программы. Характерной особенностью является возможность работы без подключения к базе данных. Основная функциональность открывается посредством объектов *DataSet*, которые являются представлениями реляционной базы данных в памяти, и *DataTable*, являющимися преставлениями непосредственных отношений. Для взаимодействия с базой данных используется адаптер данных. *Visual Studio* позволяет в автоматическом режиме сгенерировать строго типизированный объект *DataSet* по заданной базе данных. Автономный способ взаимодействия с базами данных является более удобным, чем уровень подключения, однако он требует выделения памяти для представления базы данных.
- *Entity Framework.* Является наиболее современным методом взаимодействия с базами данных в *.NET*. В отличие от двух других, описанных выше способов, *Entity Framework* позволяет оперировать коллекциями объектов, которые называются *сущностями* и абстрагироваться от данных, представленных в виде строк и столбцов в реляционной базе данных. Важным преимуществом *Entity Framework* является возможность выполнения запросов *LINQ*, которые отправляются на обработку базе данных и возвращают строго типизированные объекты. Используя *Entity Framework*, возможно построение приложения, взаимодействующего с базой данных в коде, у которого нет ни одного SQL запроса. Именно технология *Entity Framework* применяется в разрабатываемом проекте.

3.3.2 Теоретические аспекты работы с Entity Framework

Платформа *Entity Framework* [3] является программной моделью, осуществляющей отображение конструкций базы данных на объектно-ориентированную модель.

В центре находится понятие *сущности* – это экземпляр строго типизированного класса, отображающего объект предметной области.

Между сущностями можно устанавливать *связи*. Для взаимодействия со связанными сущностями используются *навигационные свойства*.

Важным элементом *Entity Framework* является *служба объектов*. Она управляет сущностями на стороне клиента, отвечает за отношения между ними. Кроме того, отслеживает изменения, внесенные в сущности, и позволяет сохранять их в базе данных.

Работу, связанную с установлением соединения и взаимодействием с базой данных выполняет *клиент сущности*. Именно он генерирует SQL операторы, которые обрабатываются базой данных.

В рамках *Entity Framework* модель управления данными (*Entity Data Model*) состоит из трех частей:

1. Концептуальная модель, которая задает сущности и отношения между ними.
2. Логическая модель, отвечающая за проекцию сущностей на таблицы базы данных.
3. Физическая модель, представляет непосредственно базу данных.

Взаимодействие информацией в коде осуществляется через контекст объектов. Данный тип является производным от класса *DbContext* и представляет сеанс работы с базой данных, позволяющий запрашивать, изменять, добавлять и сохранять хранимую информацию.

Возможны три различных подхода к началу работы с *Entity Framework*:

1. *Code-First*. Данный метод основан на описании модели данных, которыми предполагается оперировать в коде. На основе этого представления *Entity Framework* генерирует базу данных.
2. *Database-First*. В случае, если база данных уже существует, можно автоматически сгенерировать объектную модель в виде классов *C#*.
3. *Model-First*. Последним способом создания является концептуальное описание требуемой структуры данных, используя графические средства. Такое описание включает в себя описания названий сущностей, их внутренней структуры и установление связей между ними. Затем *Entity Framework* создает соответствующую объектную модель и базу данных.

В разрабатываемой системе был применён подход *Model-First*. Используя его, важно учитывать тот факт, что сгенерированный код объектного представления модели может быть автоматически пересоздан и перезаписан при изменении концептуальной модели. Учитывая этот фактор, необходимо применять принцип построения разделяемых классов для добавления функциональности к полученным типам.

3.3.3 Схема разработанной базы данных

Для реализуемой системы управления олимпиадой по программированию была разработана база данных, представляющая семь типов сущностей. Схема базы данных приведена на **Ошибка! Источник ссылки не найден.**.

Сущность *Session* определяет сессию олимпиады. Для каждой сессии задается код доступа (*AccessCode*). Этот код выдается участникам и указывается при их регистрации. Каждая сессии имеет список заданий (*Task*).

Тип *Task* определяет задачу, которую должен решить участник. Свойство *TimeLimit* задает максимальное время решения теста программой. *MaxAttemptsCount* устанавливает максимальное количество попыток сдачи кода для задания. Каждая задача связана с набором контрольных тестов

(*Tests*), для которых задается входной параметр (*Question*) и правильный ответ (*Answer*).

Сущность участника олимпиады представляется типом *Member*. У каждого участника есть уникальный идентификатор (*GUID*), заданный типом *Guid*. Данный тип представляет 128-разрядное целое. Этот идентификатор применяется для авторизации пользователя службой подключения клиентов. Участник имеет коллекцию программ (*Program*).

Сущность *Program* представляет программный код, отправленный участником. Данная сущность имеет ряд свойств, используемых службой тестирующих компонентов для корректного позиционирования этой сущности в очереди на проверку. Данный тип связан с набором результатов проверки тестов. Программа пользователя ассоциирована с языком программирования, который задан в виде сущности *ProgramLanguage*.

Результат прохождения теста программой представлен типом перечисления *ProgramTestResult*, который содержит описание возможных исходов проверки.

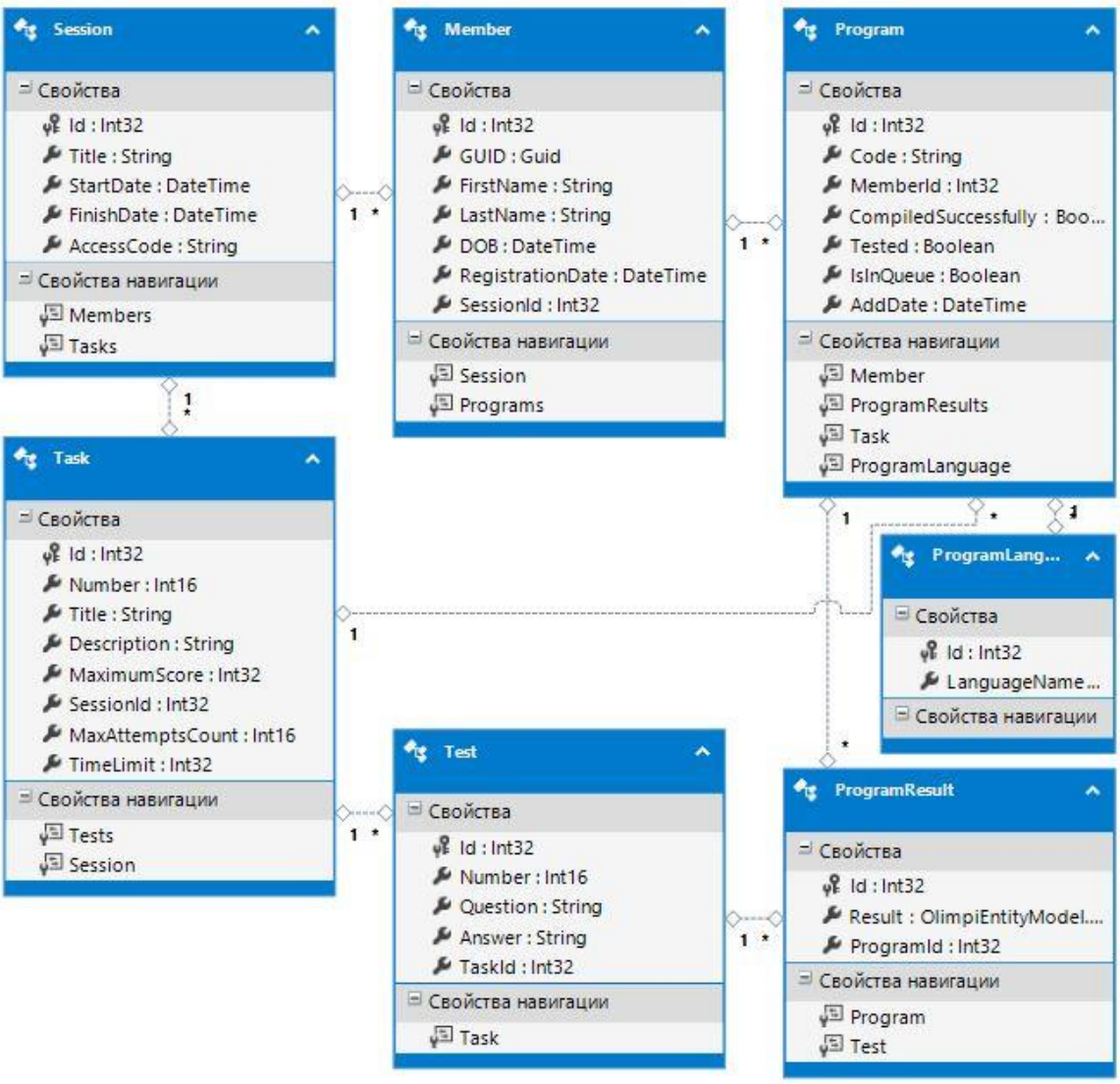

Рис. 2. Схема базы данных.

Разработанная структура данных подразумевает выполнение требований, установленных форматом проведения олимпиад.

3.3.4 Реализация взаимодействия с базой данных

В разрабатываемой системе взаимодействие с базой данных происходит в трех компонентах. Ввиду этого фактора, было принято решение изолировать всю функциональность, необходимую для доступа к базе данных внутри отдельной библиотеки *OlimpiDAL*.

Написанная библиотека предлагает различные варианты получения контекста объектов и задания параметров подключения к базе данных: передавая строку подключения через параметры конструктора, параметры через конфигурационный файл или через вызов статического метода *New()*

класса *OlimpiEntityModelContainer*, который возвращает новый экземпляр контекста, полученного используя строку подключения, заданную непосредственно в свойствах библиотеки.

Для примера приведем код получения списка задач олимпиады, для пользователя с указанным *GUID*:

```
public GetInfoResult GetInfo(Guid guid)
 {
      GetInfoResult infoResult = new GetInfoResult();
      infoResult.Success = false;
      using (OlimpiDAL.OlimpiEntityModelContainer context =
      OlimpiDAL.OlimpiEntityModelContainer.New())
     {
         var member = context.MemberSet.Where(m =>
            m.GUID == guid).FirstOrDefault();
         if (member != null)
         {
             List<Task> tasksList = new List<Task>();
             foreach (var item in member.Session.Tasks)
             {
                   tasksList.Add(new Task()
                   {
                   Number = item.Number,
                   Title   = item.Title,
                   Description = item.Description,
                   MaximumScore = item.MaximumScore,
                   MaxAttemptsCount = item.MaxAttemptsCount});
             }
             infoResult.Tasks = tasksList;
             infoResult.FinishDate = member.Session.FinishDate;
             infoResult.StartDate = member.Session.StartDate;
             infoResult.Success = true;
         }
         else {
             infoResult.ProblemDescription = "Неправильный GUID";
         }
     }
     return infoResult;}
```

В данном примере происходит проверка наличия пользователя с указанным *GUID* в базе данных, есть он имеется, то формируется список задач, которые ему необходимо решить, на основании получения информации о сессии, к которой он относится.

3.4 Службы

Как было описано ранее, в разрабатываемой системе есть две службы: подключения клиентов и тестирующих компонентов.

Службой (*service*) называют функциональный модуль, доступный извне.

Клиентом службы называется сущность, использующая функциональность службы. Служба может выступать в роли клиента для другой службы.

Существуют различные подходы для осуществления сетевого взаимодействия в рамках платформы *Microsoft .NET Framework*. Среди всех технологий наиболее ярко выделяется технология *Windows Communication Foundation* [4].

Windows Communication Foundation (*WCF*) — это унифицированная интегрированная среда для создания защищенных, надежных, транзакционных и интероперабельных распределенных приложений [5].

3.4.1 Устройство службы WCF

Службы *WCF* имеют архитектуру, позволяющую упростить процесс написания кода, сделать приложение гибким к изменениям. Службы обмениваются данными за счет передачи сообщений. На **Ошибка! Источник ссылки не найден.** приведена структура сетевого взаимодействия основанного на службах *WCF*.

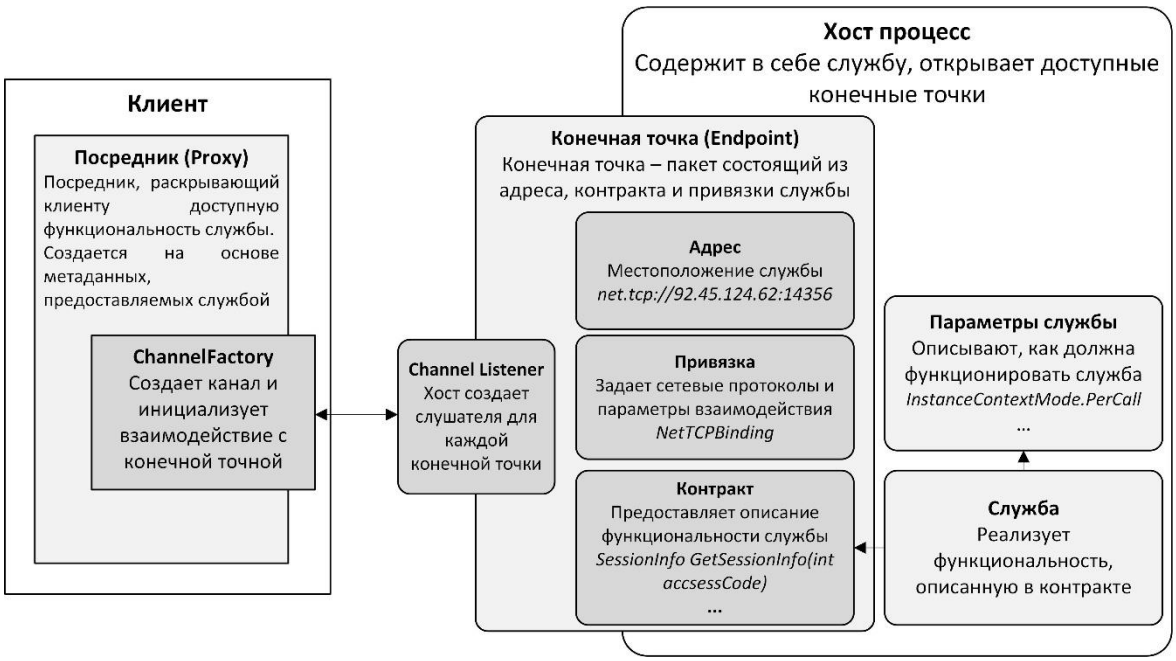

Рис. 3. Сетевое взаимодействие, основанное на службах WCF.

WCF использует принцип проектирования, ориентированный на службы (*service-oriented architecture*), и в приложении *WCF* автоматически реализуются четыре важных принципа данного подхода:

1. *Явное задание границ*. Функциональность службы описывается в интерфейсах, которые не должны меняться в процессе работы. В случае, если необходимо внести в него изменения, стоит создать новый интерфейс, реализующий требуемую функциональность.
2. *Автономность службы*. Данный принцип заключается в построении служб в виде автономных сущностей. Вся внутренняя структура службы скрывается от клиента.
3. *Взаимодействие через контракт*. Внешнее взаимодействие со службой доступно только через заданный интерфейс, называемый контрактом. Информация о реализации службы, языке, на котором она написана, номерах используемых сборок не сообщается клиенту.
4. *Совместимость основана на политике*. Детали функционирования службы, касающиеся семантики службы, задаются отдельно от синтаксического описания функциональности, через политики.

3.4.2 Контракты служб

Рассмотрение темы служб *WCF* невозможно без введения понятия контакта.

Контракт – платформенно независимый способ описания функциональности службы в формате *WSDL* [6], посредством определения интерфейсного типа и специальных атрибутов. *WSDL - Web Services Description Language* язык описания сервисов в виде XML.

В *WCF* существуют четыре различных типов контрактов:

1. *Контракт службы*. Описывают функциональность, предоставляемую службой.
2. *Контракт данных*. Формализуют типы данных, используемые при взаимодействии со службой.
3. *Контракт ошибок*. Определяют типы исключений, которые передаются клиентам в случае возникновения ошибок.
4. *Контракт сообщений*. Позволяют задавать точную структуру передаваемых сообщений, данный тип контракта применяется сравнительно редко.

В разрабатываемой системе используются только контакты служб и данных. В случае возникновения ошибок в коде службы, информация об исключительной ситуации передается посредством специальных типов с суффиксом *Result*, представляющих результат операции. Ручное определение типов сообщений и взаимодействия с ними так же не требуется в данном случае.

Фактически можно использовать определения конкретного класса для описания контракта службы, однако данный подход нарушает принцип проектирования служб. Рекомендуемой манерой является задание интерфейса, помеченного атрибутом *[ServiceContract]*.

К каждой операции, определенной в этом интерфейсе, которая должна представлять аспект разрабатываемой службы, применяется контракт *[OperationContract]*.

Пример кода:

```
[ServiceContract]
 public interface IClientService
 {
      [OperationContract]
RegisterUserResult RegisterUser(string accessCode, UserData userData);
…
 }
```

Если какой-либо элемент интерфейса не будет помечен атрибутом контракта операции, то он не будет включен в контракт службы. За счет такого подхода обеспечивается четкое определение границ служб.

Перегрузка методов недоступна в привычном виде при работе со службами *WCF*. Если требуется реализация перегруженной операции, необходимо определить псевдоним, задав свойство *Name* у атрибута *OperationContractAttribute*.

Если взаимодействие со службой проектируется, используя лишь базовые типы, то возможно обойтись без ручного описания контракта данных. Однако, чтобы служба могла принимать и возвращать сложные типы данных, необходимо определить контракты данных. Для этого тип данных, которым оперирует служба должен быть помечен атрибутом *[DataContract]*, а все его свойства атрибутом *[DataMember]*. В качестве примера приведем описание типа, представляющего регистрационные данные участника олимпиады.

```
public class UserData
{
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    [DataMember]
    public DateTime DOB { get; set; }
}
```

Сформированный тип можно использовать как в реализации службы, так и на стороне клиента.

3.4.3 Управление экземплярами службы

Механизм, с помощью которого устанавливается привязка клиента к конкретному экземпляру службы, называется управлением экземплярами. Существуют три режима:

1. *Одиночная служба*. Все подключения обслуживает один экземпляр службы, который находится в памяти в течение всего выполнения. Данный способ можно применять только в случае небольшого количества клиентов. Характерной особенностью такого подхода является возможность передачи параметров в службу через конструктор при ее инициализации, что недопустимо в других режимах.
2. *Служба уровня вызова*. В данном режиме новый экземпляр службы создается при каждом обращении клиента и уничтожается после завершения обработки. Это обеспечивает хорошую масштабируемость и применимо в случае большого количества клиентов. Смена экземпляров службы происходит прозрачно для клиента.
3. *Служба уровня сеанса*. Новый экземпляр службы создается для каждого нового сеанса и существует до тех пор, пока сеанс не будет завершен.

Тип управления экземплярами задается непосредственно в коде службы за счет присваивания значения свойству *InstanceContextMode* в атрибуте *ServiceBehaviorAttribute*.

В разрабатываемой системе применяется одиночная служба и служба уровня вызова.

3.4.4 Размещение службы

Службы *WCF* не могут существовать сами по себе. Любая служба находится под управлением некоторого процесса, который называется *хост-процессом*. Один хост-процесс может размещать несколько служб, один тип службы может быть размещен различными хост-процессами. Возможны

различные варианты хостинга, среди которых хостинг веб-сервера *Microsoft IIS*, хостинг в облаке *Azure*, хостинг приложения.

В разрабатываемой системе используется *хостинг приложения* (*авто хостинг*), когда службы размещаются программным образом, используя класс *ServiceHost*. Каждый экземпляр данного класса инициализируется с указанием типа службы, которую он будет размещать. Используя выбранный способ размещения, существует возможность явного программного управления состоянием службы, ее остановка и запуск.

У хостинга должна быть как минимум одна *конечная точка*, чтобы существовала возможность обратиться к размещаемой им службе.

Конечной точкой задается набор из трех элементов, описывающих службу и способ взаимодействия с ней:

1. *Адрес*, который задает местоположение службы.
2. *Контракт*, описывающий функциональность службы.
3. *Привязка*, задающая принципы сетевого взаимодействия.

Задание этих параметров возможно как в конфигурационном файле приложения, так и непосредственно в коде программы. Служба обычно не зависит от типа привязки, что позволяет изменить протокол взаимодействия, без перекомпиляции.

Особым видом конечных точек являются точки *MEX*, которые предоставляют *метаданные* службы в формате WSDL. Используя эти данные, можно автоматически сгенерировать клиентский прокси за счет *Visual Studio*.

3.4.5 Служба подключения клиентов

В реализуемой системе служба подключения клиентов обрабатывает все обращения клиентов пользователей. Выбранный алгоритм сетевого взаимодействия предполагает одиночные запросы с большими временными интервалами между ними. Поэтому данная служба работает в режиме службы уровня вызова рис. 4.

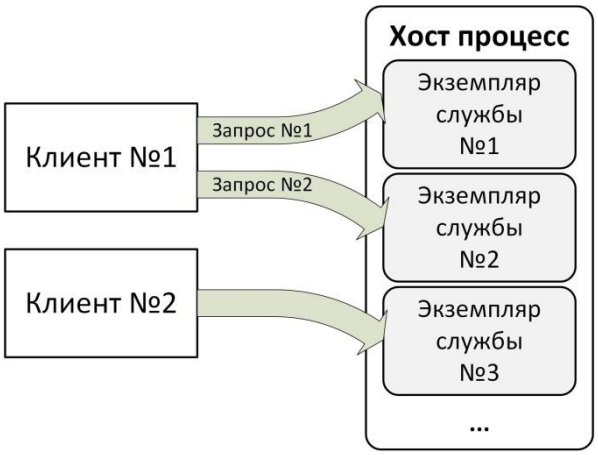

Рис. 4. Модель PerCall.

Для того, чтобы установить данный способ управления экземплярами службы в тексте кода сервиса определен атрибут:

```
[ServiceBehavior(InstanceContextMode =InstanceContextMode.PerCall)]
```

Наглядное представление логики сетевого взаимодействия представлено в виде *UML* диаграммы на рис. 5.

Процесс состоит из четырех этапов:

1. Регистрация участника, на этом этапе участник получает *GUID*, который в дальнейшем используется для его идентификации.
2. Запрос информации об олимпиаде.
3. Передача программы на тестирование.
4. Запрос результатов проверки.

Первая и вторая операции выполняются один раз для каждого участника. Передача кода на проверку происходит каждый раз, когда пользователь решил задание. Запрос результатов происходит спустя 120 сек.

после отправки программы на проверку, затем снова повторяется в случае, если не все коды были проверены.

Результатом любого обращения к службе является тип с суффиксом *Result*. Данная сущность помимо запрошенных данных включает в себя описание ошибки, если таковая была во время выполнения операции. Выбранный подход является альтернативой созданию контрактов исключений.

Приведем определение контракта данной службы:

```
[ServiceContract]
public interface IClientService
{
	[OperationContract]
	RegisterUserResult RegisterUser(string accessCode, UserData userData);
	[OperationContract]
	GetInfoResult GetInfo(Guid guid);
	[OperationContract]
	PassProgramResult PassProgram(Guid guid,UserCode userCode);
	[OperationContract]
	UserResults GetUserResults(Guid guid); }
```

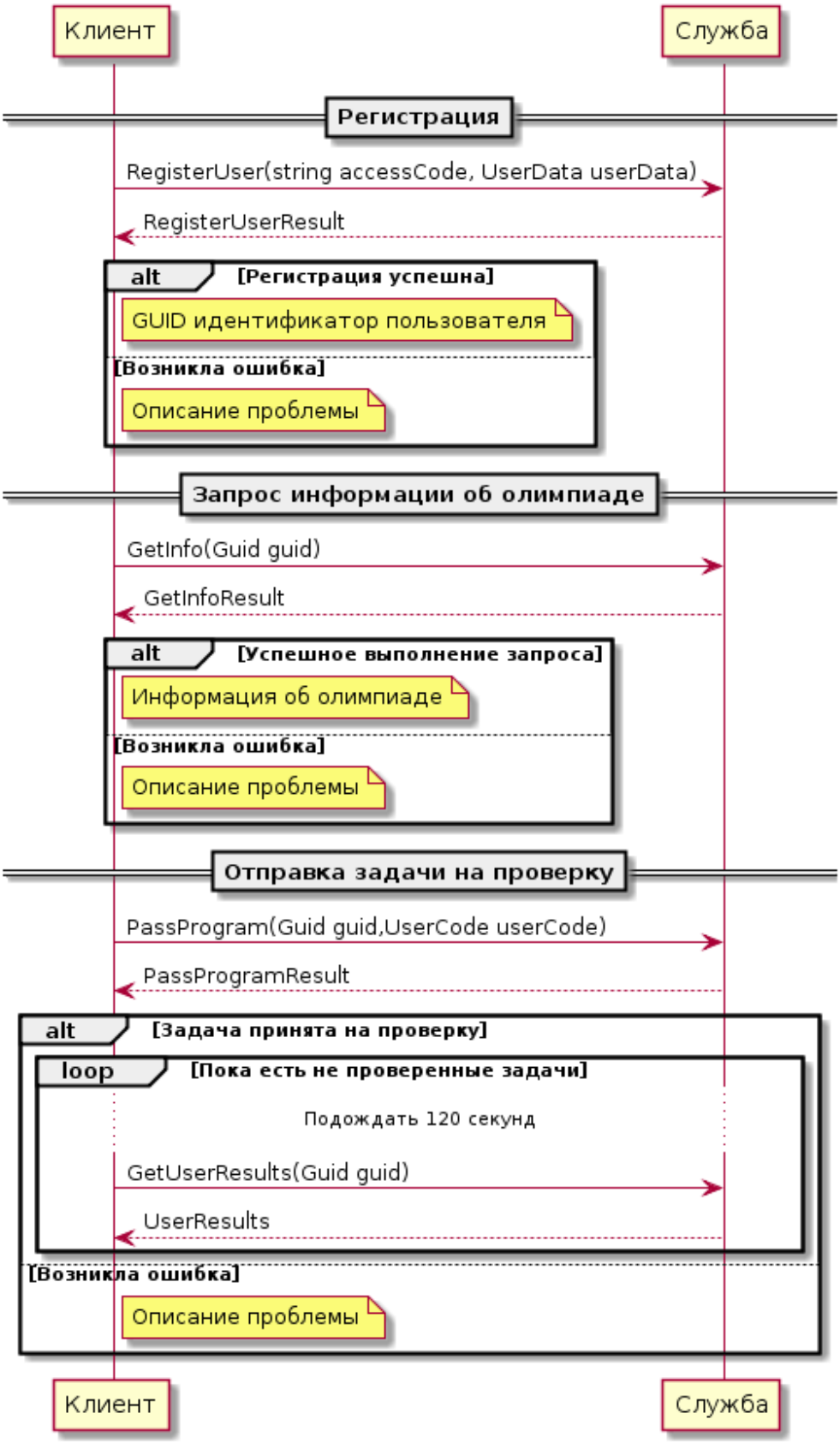

Рис. 5. Диаграмма сетевого взаимодействия клиентов и службы.

Код службы слишком велик, чтобы приводить его целиком. В качестве демонстрации, предложено рассмотреть пример получения информации об олимпиаде.

```
public GetInfoResult GetInfo(Guid guid)
{
    GetInfoResult infoResult = new GetInfoResult();
    infoResult.Success = false;
     using (OlimpiDAL.OlimpiEntityModelContainer context =
     OlimpiDAL.OlimpiEntityModelContainer.New())
    {
        var member = context.MemberSet.Where(m => m.GUID ==
     guid).FirstOrDefault();
        if (member != null)
        {
            List<Task> tasksList = new List<Task>();
            foreach (var item in member.Session.Tasks)
            {
                  tasksList.Add(new Task() {
                  Number = item.Number,
                  Title = item.Title,
                  Description = item.Description,
                  MaximumScore = item.MaximumScore,
                  MaxAttemptsCount = item.MaxAttemptsCount });
            }
            infoResult.Tasks = tasksList;
            infoResult.ServerTime = DateTime.Now;
            infoResult.FinishDate = member.Session.FinishDate;
            infoResult.StartDate = member.Session.StartDate;
            var availableLanguages = new List<ProgramLanguage>();
            foreach (var item in context.ProgramLanguageSet)
            {
                  availableLanguages.Add(new ProgramLanguage() { Name =
                  item.LanguageName });
            }
            infoResult.AvailableLanguages = availableLanguages;
            infoResult.Success = true;
        }
        else {
            infoResult.ProblemDescription = "Неправильный GUID";
        }
```

```
        }
        return infoResult;
    }
```

В данном примере можно увидеть, каким образом служба общается с базой данных. В представленном коде нет ручного закрытия контекста, оно происходит автоматически при выходе из конструкции *using*.

3.4.6 Служба тестирующих компонентов

Сервис, обслуживающий модули проверки кода рассчитан на относительно небольшое количество клиентов. В роли клиента выступает компонент, производящий компиляцию и тестирование исходных кодов программ участников олимпиады. Одной из ключевых функций данной службы является синхронизация доступа к базе данных, чтобы каждый исходный код был проверен лишь один раз. В силу описанных особенностей разумно использовать модель одиночной службы, которая задаётся атрибутом:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
```

Использование такого способа допустимо ввиду того, что время компиляции и тестирования программы на несколько порядков превышает временные затраты, требуемые для выполнения запроса к базе данных и подготовку тестовых данных для проверяющего модуля.

Функциональность данной службы заключается в выполнении трех действий:

1. Регистрация тестирующего компонента.
2. Предоставление коллекции кодов, которые необходимо протестировать.
3. Получение результатов проверки и занесении их в базу данных.

Служба тестирующих компонентов имеет следующий контракт:

```
[ServiceContract]
public interface ITestingService
{
      [OperationContract]
      RegisterTestUnitResult RegisterTestUnit(UnitInfo unitInfo);
```

```
        [OperationContract]
        GetCodesForTestResult GetCodesForTest(Guid UnitGuid);
        [OperationContract]
        ReturnCodesTestResult ReturnCodesTest(Guid UnitGuid,
        IEnumerable<UserProgram> CodesTestsResults);
    }
```

Результатом каждой операции является тип с суффиксом *Result*, который помимо запрошенной информации включает в себя сообщения об ошибках, если они произошли во время выполнения запроса.

При каждом запросе списка задач на проверку, служба помечает отправленные задачи флагом в базе данных и запоминает, какому тестирующему модулю они были предоставлены. Кроме того, происходит вычисление максимального времени, которое может потребоваться тестирующему компоненту на проверку данных задач. Вычисление происходит по формуле:

$$f(n) = d \cdot n + \sum_{i=1}^{n} t_i \cdot m_i,$$

где n – количество программ, m_i количество тестов для i-той программы, t_i – максимальное время решения теста для i-той программы. Величина d характеризует время компиляции программы. Если тестирующий компонент, назовем его *A*, не вернул результаты проверки в течении этого интервала времени, и освободился другой проверяющий модуль (обозначим его *B*), то задачи передаются на решение компоненту *B*, выдвигается предположение, что у тестирующего модуля *A* возникли проблемы с оборудованием или связью. Если через какое-то время модуль *A* снова выйдет на связь, то ему будет передана новая порция данных для обработки.

3.5 Модуль проверки правильности кода

Данный компонент отвечает за компиляцию и проверку правильности исходных кодов программ, представленных участниками олимпиады.

3.5.1 Реализация компонента

Модуль проверки правильности программ предполагает настройку большого числа параметров, таких как способы вызова компиляторов, путь к директории для проведения тестов и так далее. Кроме того, как говорилось ранее, тестирующий модуль может быть запущен на разных машинах для реализации распределенного тестирования. Ввиду этих факторов было принято решение оформить тестирующий компонент в виде отдельного настольного приложения *Windows Forms*. На рис. 6 представлен снимок экрана главного окна приложения.

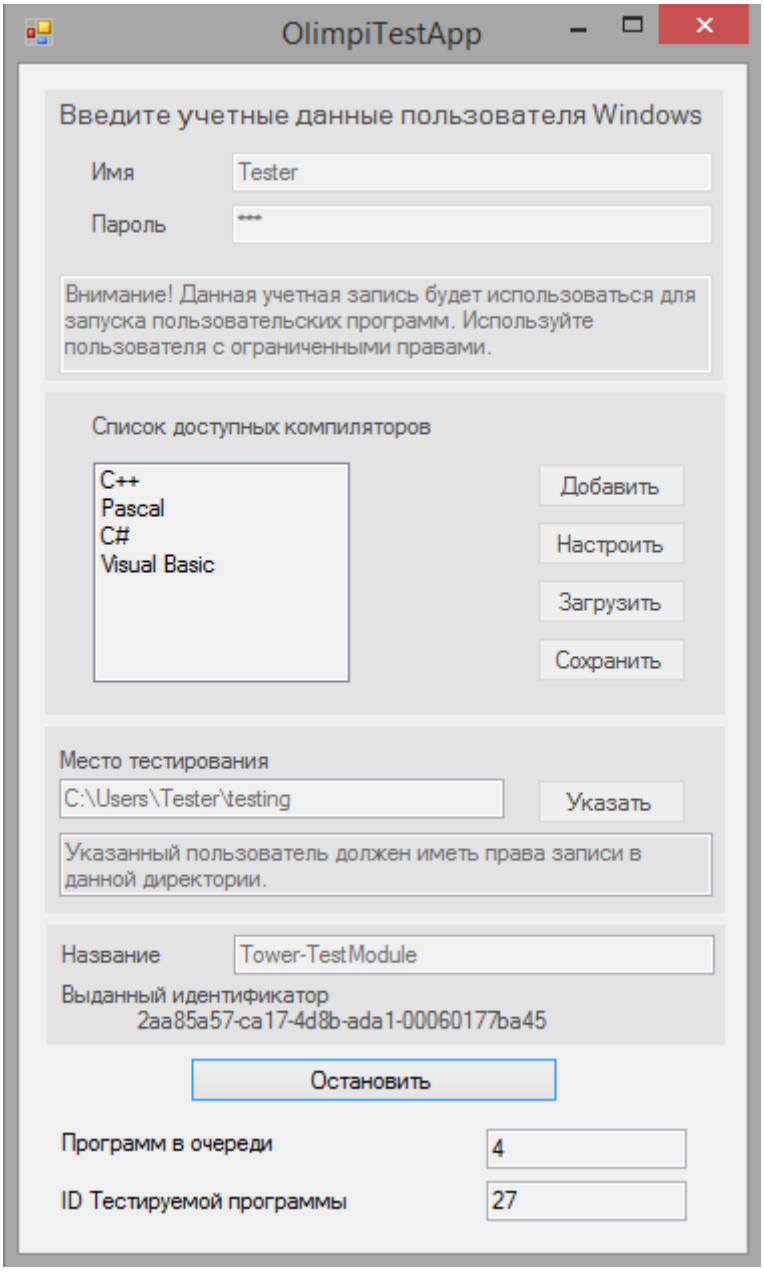

Рис. 6. Интерфейс тестирующего компонента.

На представленном изображении, большая часть элементов управления находится в неактивном состоянии, это обусловлено тем, что вся настройка должна производиться до запуска компонента. После того как модуль запущен, изменение конфигурации не допускается.

3.5.2 Проблема безопасности

Суть тестирующего компонента заключается в проверке правильности программ участников олимпиады на наборе тестов. Модуль получает исходный код программы, вызывает соответствующий компилятор, затем запускает полученную программу.

Однако участник олимпиады может предоставить вредоносный код вместо программного кода, решающего поставленную задачу. Ввиду чего необходимо обеспечить защиту и не допустить нарушения стабильности работы системы. Возможно несколько путей решения данной проблемы:

- *Виртуальная машина*. Если производить тестирование программ пользователей в рамках виртуальной машины, можно обеспечить очень высокий уровень безопасности. Однако данный способ является чрезвычайно ресурсоемким. После каждой проверенной задачи необходимо либо проверять состояние виртуальной машины, либо разворачивать новую. И то и другое требует большого количества времени.
- *Контейнеризация*. Более современным подходом является применение контейнеризации – новой технологии, которая позволяет изолировать приложение в рамках контейнера, в котором поставляется требуемое окружение. Использование данного подхода является самостоятельной и сложной задачей.
- *Песочница*. На рынке существует ряд решений, например, *Sandboxie* или *Evalaze*, которые реализуют песочницу – виртуальную рабочую среду, в рамках которой выполняется приложений. Данные приложения являются коммерческими.

- *Средства .NET framework.* Сама платформа *.NET* имеет средства контроля над исполняемым процессом. Возможно задание списка директорий, к которым может обращаться выполняемый процесс. Однако данное средство работает лишь с управляемым кодом, а разрабатываемая система ориентирована на тестирование, в том числе и неуправляемого программного кода.
- *Средство контроля учетных записей Windows.* Начиная с выпуска *Windows Vista,* компания *Microsoft* начала снабжать операционные системы посредством контроля учетных записей пользователя *UAC* (*User Account Control*) [7]. Данный компонент является средством защиты, требующим прав администратора для выполнения потенциально опасных действий, таких как внесение изменений в системный реестр, управление сетевым экраном, изменение файлов в системных каталогах. Именно этот способ предложено использовать для обеспечения защиты в разрабатываемой системе.

Кроме описанной выше функциональности *UAC* блокирует доступ процесса, выполняемого без прав администратора к директориям других пользователей ПК. Ввиду чего предложено создание нового пользователя на машине, на которой будет запускаться тестирующий компонент. Упомянутая учетная запись должна быть без привилегий администратора, от ее имени будет происходить выполнение программ участников олимпиады.

Приведем код на языке *C#* для запуска процесса от имени конкретного пользователя:

```
System.Diagnostics.Process proc = new System.Diagnostics.Process();
proc.StartInfo.FileName = @"C:\Users\Tester\userProgram.exe";
proc.StartInfo.WorkingDirectory = @"C:\Users\Tester";
proc.StartInfo.Domain = "TestingDomain";
proc.StartInfo.UserName = "Tester";
proc.StartInfo.Password = secureString;
proc.Start();
```

В приведенном коде программа выполняется от имени пользователя *Tester*. Такой подход не требует затрат большого количества ресурсов для обеспечения безопасности.

3.5.3 Процесс запуска

Для подключения дополнительного тестирующего компонента к системе требуется произвести ряд действий:

1. Добавить нового пользователя ПК, на котором будет происходить выполнения программы. Такая учетная запись не должна иметь прав администратора.
2. Ввести данные учетной записи в соответствующие поля в программе.
3. Произвести конфигурацию компиляторов, которые будут использоваться. Задание производится через графический интерфейс. Все пути можно задавать как в виде строки текста, так и используя диалоговые окна. Программа позволяет сохранять и загружать данные о взаимодействии с компиляторами в формате XML.
4. Создать директорию, в которой будет производиться тестирование программ, и указать ее в соответствующем поле компонента.
5. Задать пользовательское имя компонента. Данное имя служит исключительно для удобства администрирования.
6. Осуществить запуск путем нажатия соответствующей кнопки. После этого компонент подключится к серверу и зарегистрируется в системе. Адрес подключения задается в конфигурационном файле. Если запуск прошел успешно, то отобразится идентификатор, выданный сервером.

После осуществления запуска внизу окна программы будет отображаться короткая информация о программе, которая проверяется в данный момент и количестве исходных кодов в очереди.

3.6 Редактор списка задач и тестов

Для функционирования разрабатываемой системы необходим список задания олимпиады и список проверочных тестов. Чтобы предоставить рассмотренную функциональность, было разработано небольшое приложение, названное *OlimpiEditor,* предоставляющее все требуемые возможности для генерации подобного списка. Снимок экрана главного окна программы представлен на рис. 7.

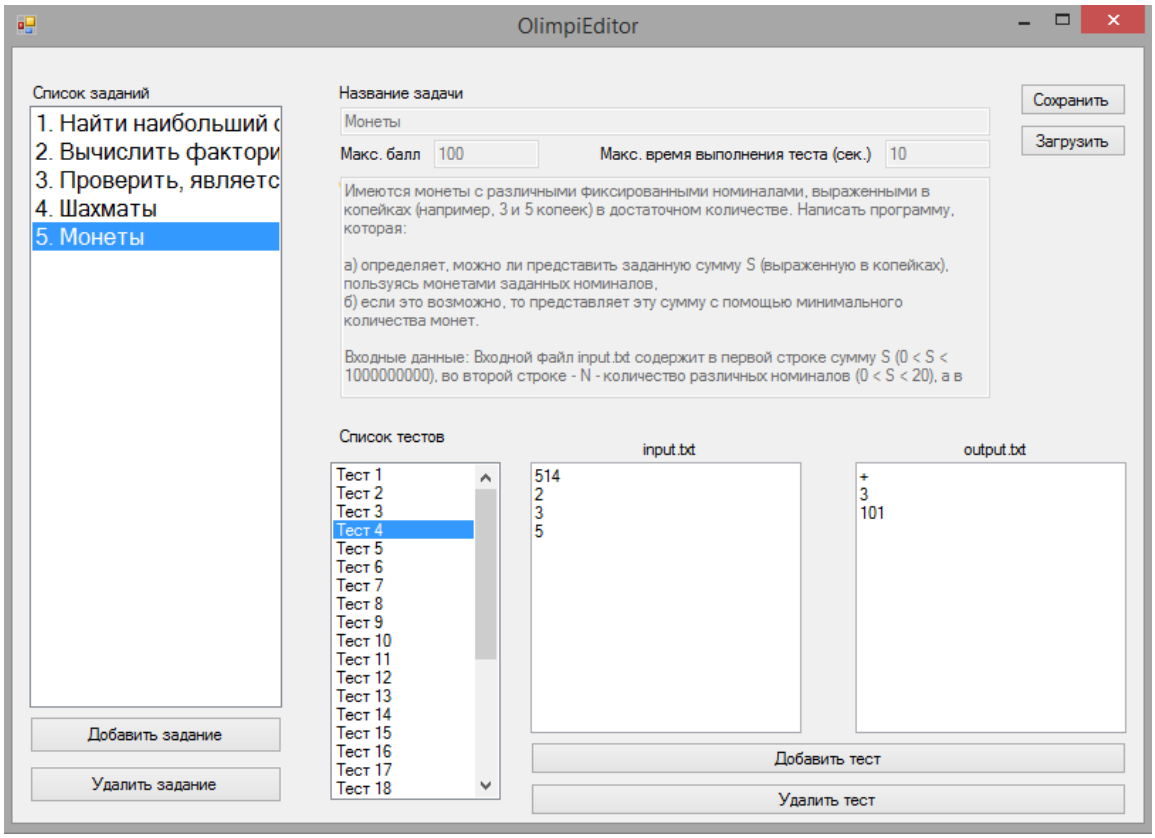

Рис. 7. Редактор списка задач и тестов.

С помощью этой утилиты определяются задачи, включающие в себя описание, которое передается участникам, максимальный балл, ограничение на время прохождения теста. Для каждого задания определяется набор тестовых данных, содержащих входные данные и правильный ответ, который должен быть сформирован программой пользователя, чтобы тест был пройденным.

После занесения определения списка задач и проверочных тестов информация сохраняется в виде XML файла. Сформированные данные используются при создании сессии олимпиады.

3.7 Графическая оболочка сервера

Для повышения удобства администрирования олимпиады серверное приложение было оформлено в виде приложения с графическим пользовательским интерфейсом. Изображение главного окна программы представлено на рис. 8.

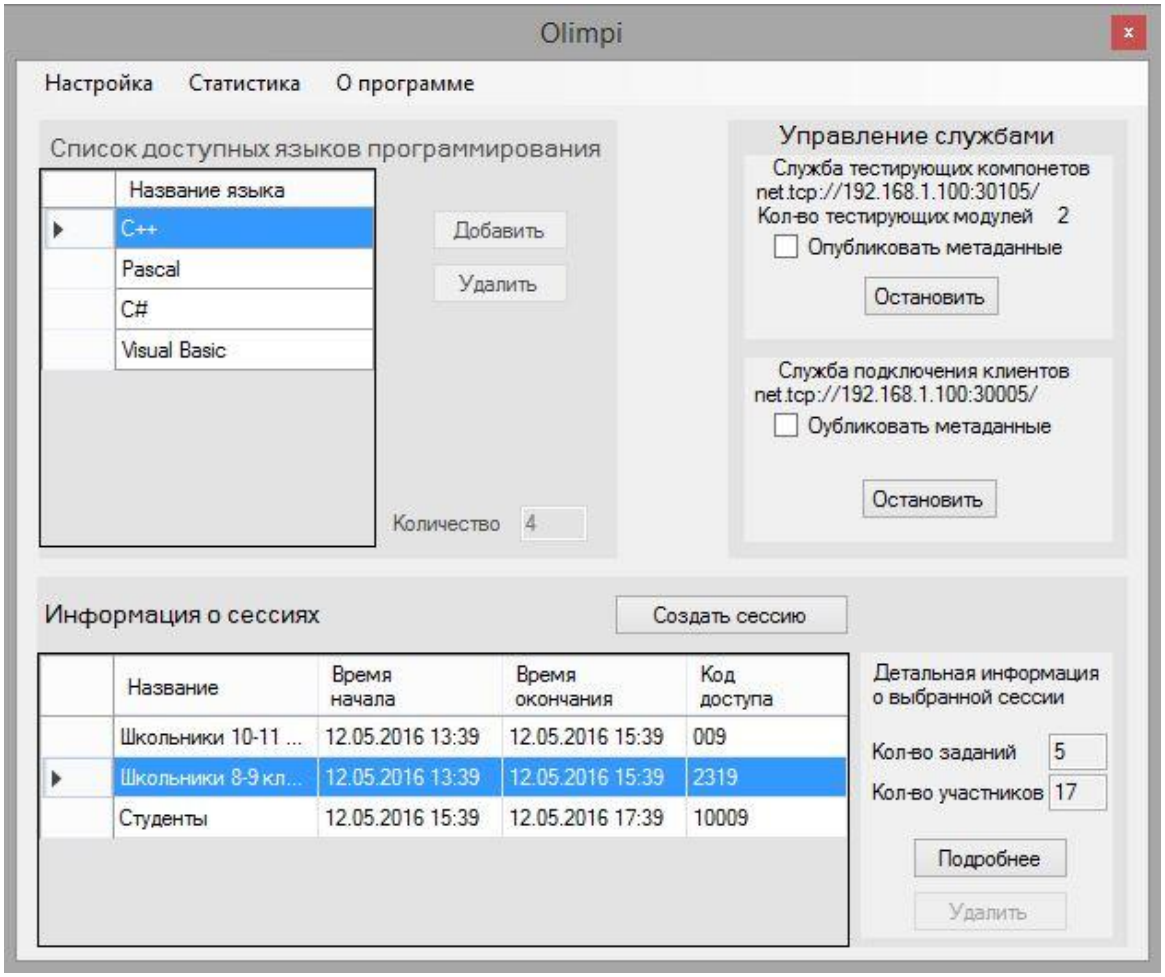

Рис. 8. Графический интерфейс сервера.

С помощью интерфейса можно производить всю необходимую конфигурацию: задание списка языков программирования, доступных для решения задач, добавление сессий олимпиад, настройка и запуск служб подключения.

Используя приложение можно просматривать участников любой из сессии, сданные ими программы и результаты их проверки. Однако внесение изменений в эту информации запрещено в целях обеспечения честности проведения олимпиады.

Функциональность оболочки заключается разрешении выполнения действий только в правильной последовательности, если попытаться ее нарушить, будет выдано соответствующее сообщение. Алгоритмы работы с данной средой состоит из следующих этапов:

1. Задать список доступных языков программирования. Данные языки должны поддерживаться тестирующими компонентами и будут доступны участникам для сдачи программ.
2. Запуск службы тестирующих компонентов. После того как служба была запущена, необходимо подключить тестирующие модули. После размещения данной службы вносить изменения в список доступных языков запрещается.
3. Следующим этапом является запуск службы подключения клиентов.
4. Создание сессии олимпиады является заключительным этапом. Можно производить добавление сессий как до запуска службы подключения клиентов, так и во время ее выполнения. Однако удаление сессий допустимо лишь в случае, если ее время вышло, либо служба подключения клиентов остановлена.

Процесс создания сессии включает в себя указание конфигурационного XML файла, содержащего список заданий и тестов для олимпиады, даты начала и конца сессии, дружественного имени, описывающего олимпиады и кода доступа, который будет передан пользователям.

3.8 Клиент пользователя

Участники олимпиады регистрируются в системе, получают список заданий, отправляют решения на проверку и узнают результаты посредством

клиента пользователя. Данное приложение является настольным приложением *Windows Forms*.

В начале работы участнику необходимо указать регистрационную информацию и код сессии олимпиады. После этого система выдает ему уникальный *GUID*, который используется для авторизации пользователя. Клиентское приложение сохраняет полученный идентификатор в файл на случай сбоя в системе или перезапуска приложения. Снимок экрана главного окна приложения представлен на рис. 9.

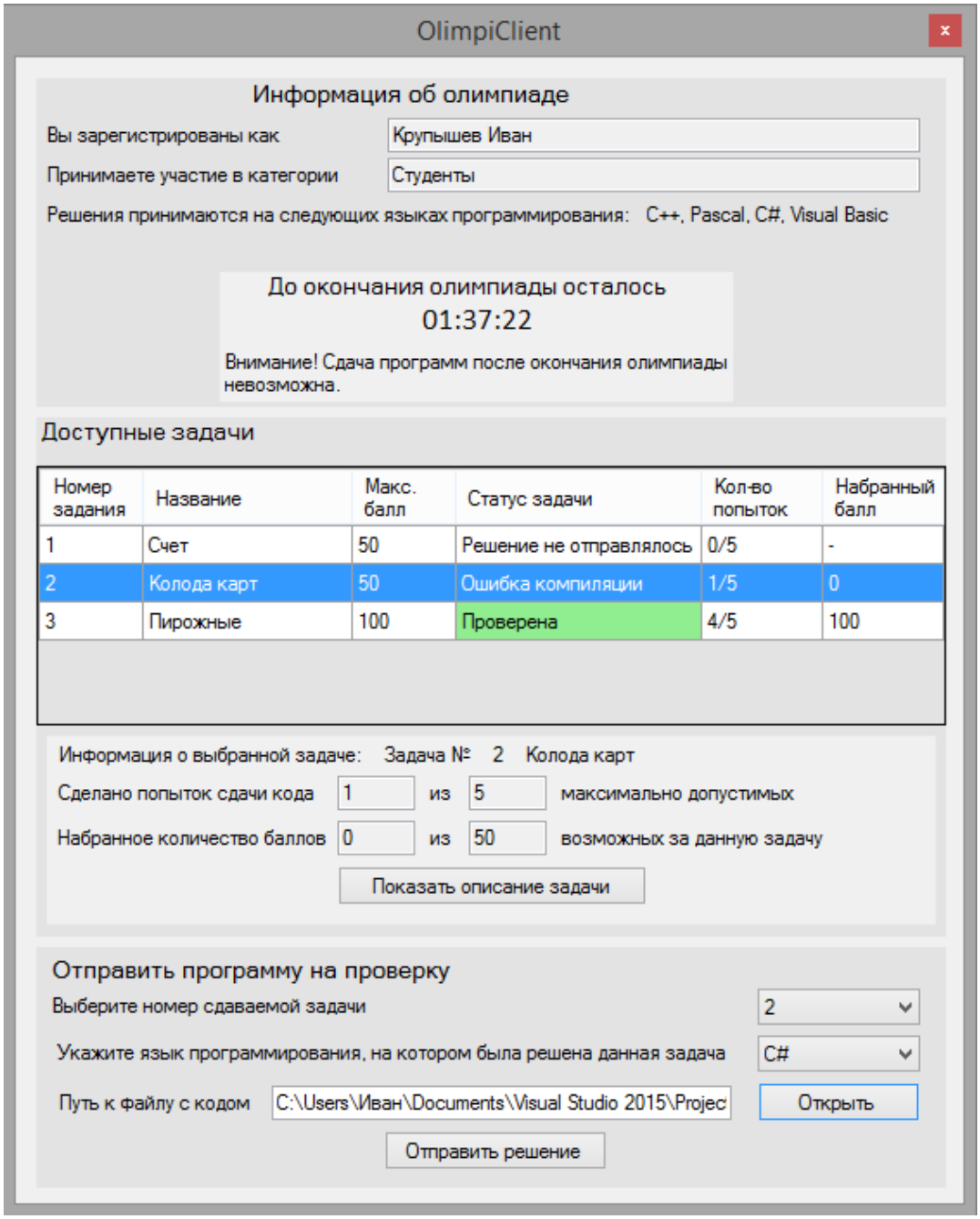

Рис. 9 Клиентское приложение.

Используя интерфейс программы, участник узнает требуемую информацию касательно списка доступных языков, оставшееся время до конца олимпиады.

Выполнение запроса к службе сервера может занять некоторое время, особенно в случае, если в текущий момент сервер загружен обработкой большого количества подключений. Если использовать синхронную модель программирования, то задержка может создать эффект зависания приложения-клиента. Ввиду чего важным аспектом является реализация взаимодействия клиента со службой в неблокирующей манере. Чтобы добиться требуемого результата, необходимо обращаться к службе сервера в отдельном программном потоке. Реализовывать такую модель взаимодействия удобно, используя ключевые слова *await* и *async* [8], доступные начиная с версии *.NET Framework 4.5*. В качестве демонстрации приведем фрагмент кода регистрации пользователя в неблокирующей манере.

```
private async void buttonRegistration_Click(object sender, EventArgs e)
 {
     buttonRegistration.Enabled = false;
     …
     using (ClientServiceClient client = new ClientServiceClient())
     {
       var registrationResult = await client.RegisterUserAsync(accessCode,
       userData);
```

В этом примере метод *client.RegisterUserAsync* возвращает тип *Task<RegisterUserResult>*, ключевое слово *await* отвечает за извлечение внутреннего значения, содержащегося в объекте *Task*. За счет того, что обработчик события нажатия кнопки *buttonRegistration_Click* помечен ключевым словом *async*, компилятор узнает, что код должен выполняться в неблокирующей манере. Использование данной нотации сильно сокращает объем рукописного кода, требуемого для реализации асинхронного выполнения операции за счет автоматической генерации требуемого кода взаимодействия с потоками компилятором.

Подготовка рабочего места участника представляет собой простое развертывание клиентской части программного обеспечения разработанной системы из дистрибутива. Необходимые среды разработки программного кода включены в указанный дистрибутив.

Заключение

В результате проделанной работы был создан программный продукт, обеспечивающий автоматическую проверку правильности кода программ в рамках проведения мероприятия по программированию. Представленная система основана на современных технологиях, обеспечивающих гибкость использования и возможность динамического масштабирования системы.

Указанный программный продукт состоит из набора программных компонентов, таких как клиент пользователя, серверная оболочка, службы подключения клиентов и тестирующих компонентов, библиотека работы с базой данных, модуль тестирования.

В созданном решении реализована структура, позволяющая развивать и добавлять новую функциональность. Например, система позволяет добавить компонент контроля за компьютером участника, который будет отслеживать действия пользователя, анализируя подключенные устройства, поисковые запросы, создавая регулярные снимки экрана пользователя.

Решение касательно необходимости добавления этого функционала может быть принято после апробации представленного программного продукта в организации олимпиад по программированию в вузах, школах и прочих заинтересованных организациях.

Список литературы

1. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5. М.: Вильямс, 2013. 1312 с.
2. Установка .NET Framework // Microsoft Developer Network URL: https://msdn.microsoft.com/ru-ru/library/5a4x27ek(v=vs.110).aspx (дата обращения: 11.05.16).
3. Using IntelliSense // Microsoft Developer Network URL: https://msdn.microsoft.com/en-us/library/hcw1s69b(VS.71).aspx (дата обращения: 11.05.16).
4. Lerman J. Programming Entity Framework. Second Edition. Sebastopol: O'Reilly Medias, 2010. 920 p.
5. Lowy J., Montgomery M. Programming WCF Services. 4th Edition. Sebastopol: O'Reilly Medias, 2015. 1018 p.
6. Windows Communication Foundation Services and WCF Data Services // Microsoft Developer Network URL: https://msdn.microsoft.com/ru-ru/library/bb907578.aspx (дата обращения: 11.05.16).
7. WSDL: взгляд изнутри // CIT forum URL: http://citforum.ru/internet/webservice/wsdl_1/ (дата обращения: 11.05.16).
8. User Account Control // Microsoft TechNet URL: https://technet.microsoft.com/en-us/library/cc731416.aspx (дата обращения: 11.05.16).
9. Асинхронное программирование с использованием ключевых слов Async и Await // Microsoft Developer Network URL: https://msdn.microsoft.com/ru-ru/library/hh191443.aspx (дата обращения: 11.05.16).