**ГОУ ВПО РОССИЙСКО-АРМЯНСКИЙ (СЛАВЯНСКИЙ) УНИВЕРСИТЕТ**

**Институт математики и высоких технологий**

**Кафедра системного программирования**

**ДИПЛОМНАЯ РАБОТА**

***Тема:***

***Разработка и реализация программного обеспечения для автоматизации проверки задач по информатике***

**Выполнил студент 4-его курса: Оганесян Левон Эдгарович**

**Научный руководитель: Цирунян Армен**

**Ереван 2017**

**Оглавление**

[Введение 3](#_Toc494669461)

[Требования к библиотеке и задачи 4](#_Toc494669462)

[Необходимые понятия 7](#_Toc494669463)

[Разработка и концепт 8](#_Toc494669464)

[Архитектура библиотеки 9](#_Toc494669465)

[Классы 10](#_Toc494669466)

[Test 10](#_Toc494669467)

[PrimitiveTest<T> 10](#_Toc494669468)

[CompositeTest 10](#_Toc494669469)

[Range 10](#_Toc494669470)

[ConstPrimitiveTest<T> 10](#_Toc494669471)

[RangePrimitiveTest<T> 10](#_Toc494669472)

[PrimitiveTest<T> 10](#_Toc494669473)

[Delimeter 10](#_Toc494669474)

[Array 10](#_Toc494669475)

[ConstStringSet 10](#_Toc494669476)

[RandomTestSet 10](#_Toc494669477)

[Grammar 10](#_Toc494669478)

[RegEx 10](#_Toc494669479)

[Graph 10](#_Toc494669480)

[Разбор задач 11](#_Toc494669481)

[Решение задач и генерация ответов 12](#_Toc494669482)

[Организация олимпиад и связь с ejudge 13](#_Toc494669483)

[Проверка домашних заданий 14](#_Toc494669484)

[Публикация, распространение и использование библиотеки 15](#_Toc494669485)

[Возможные дальнейшие планы 16](#_Toc494669486)

[Источники 17](#_Toc494669487)

# Введение

Проверка домашних заданий является наиболее долгой и сложной частью преподавательского труда. Из-за огромного количества домашних заданий качество проверки может упасть и преподаватель может упустить из виду какие-то мелкие детали или не проверить какую-то отдельную часть задания. Именно с целью избежания таких проблем было решено создать библиотеку, которая должна позволять с лёгкостью создавать тесты, по корректному решению задач получать правильные ответы и далее автоматически проверять те задания, которые присылают студенты.

Кроме этого, данная библиотека найдет своё место в генерации тестов для олимпиадных задач, намного облегчая весь процесс организации олимпиад.

Данная работа является продолжением курсовой работы, в которой уже были представлены некоторые функции и возможности, в данной же работе будет описано много нововведений, такие как абстракции тестов, усовершенствованные массивы, графы, генерация по грамматике, по регэксу и многое другое.

Вдобавок ко всему – на момент написания данного текста, уже была проведена олимпиада для первого и второго курсов, с использованием данной библиотеки, которая прошла 16-ого сентября. Было представлено 8 задач, некоторые из них будут описаны ниже.

# Требования к библиотеке и задачи

Требования, которые были представлены в курсовой работе, являются на данный момент устаревшими. Для определения новых требований, нам нужно снова определить типичные и не типичные задачи, с которыми должна будет стравиться данная библиотека. Разумеется, требований может очень много, но мы будем затрагивать только те, которые используются в задачах чаще всего. Напомню, что условия задач нас не интересуют, поэтому ниже будут представлены только входные данные задач. Итак, задачи:

**Задача №1  
*<Условие задачи>***

***Исходные данные***  **Дано целое число – длина перестановки.**

***<Описание результата>***

Эта задача является лёгкой, но в ней есть одна особенность: количество тестов всего 16. Конечно, можно сгенерировать много тестов и надеяться, что все варианты будут учтены, но нам не подходит этот вариант.

**Задача №2  
*<Условие задачи>***

***Исходные данные***

**Дано число – количество чисел в массиве. Далее через пробел дано число – количество итераций. На следующих строк записаны пары чисел , каждое из которых может быть по модулю не больше 1000, и .**

***<Описание результата>***

Задача №2 покажет нам, как изменились наши типы, как устроены массивы и некоторое другое.

**Задача №3**

***<Условие задачи>***

***Исходные данные***

**В первой строке задано целое положительное число (), каждая из следующих строк содержит по одному запросу к системе. Каждый запрос - непустая строка длиной не более 32 символов, состоящая только из строчных латинских букв.**

***<Описание результата>***

В данной задаче, нужно заметить, что исходя из условия (которое было слишком большим, чтобы вставить его сюда) следует, что худший случай задачи, это тот, в котором все буквы одинаковы, надо будет учесть при генерации и это.

**Задача №4**

***<Условие задачи>***   
**Задача была следующая: сможете ли вы понять, является ли граф “лесом”?  
Напомним, что дерево — это связный неориентированный граф без циклов.**

**Лесом же называют граф, все связные компоненты которого являются деревьями.**

***Исходные данные***

**В первой строке заданы числа где это количество вершин в графе, а - количество ребер. Далее в строках будут перечислены пары целых чисел это значит, что присутствует ребро между вершинами и . В графе нет петель и кратных ребер.**

***<Описание результата>***

Здесь нужно сгенерировать «лес», разумеется, для этого понадобится класс графов.

**Задача №5**

***<Условие задачи>*** 

***Исходные данные***

**Адрес электронной почты состоит из имени пользователя и домена, разделённых символом «». Имя пользователя и домен — непустые строки, состоящие из строчных английских букв и точек. При этом они не могут начинаться на точку, заканчиваться точкой и содержать две точки подряд.**

**Объём входных данных не превосходит 106 байт.**

***<Описание результата>***

[**http://acm.timus.ru/problem.aspx?space=1&num=1835**](http://acm.timus.ru/problem.aspx?space=1&num=1835)

Итак, требования к библиотеке:

* Примитивные типы
* Зависимые друг от друга переменные
* Массивы
* Графы
* Регэксы
* Грамматики
* Утилита для создания множества тестов
* Удобный интерфейс
* Возможность сгенерировать любой тест
* Возможность изменить любой метод любого класса для конкретной задачи
* И многое другое.

# Необходимые понятия

* Компоновщик (англ. *Composite pattern*) — шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.
* Прототип (англ. *Prototype pattern*) — шаблон проектирования, задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Проще говоря, это паттерн создания объекта через клонирование другого объекта вместо создания через конструктор.
* Строитель (англ. *Builder pattern*) — порождающий шаблон проектирования предоставляет способ создания составного объекта. Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.
* Шаблоны (англ. *templates*) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).
* Библиотека стандартных шаблонов (STL)  (англ. *Standard Template Library*) — набор cсогласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

# Разработка и актуальные проблемы

Имея вышеописанные требования ясно, что библиотека должна быть настолько универсальной, чтобы уметь работать с почти любыми видами задач. В курсовой работе третьего курса были приведены примеры задач и то, как можно создать для них тесты, но некоторые виды задач не было возможно решить теми методами, например задачи в которых переменные зависят друг от друга. Для примера рассмотрим первую часть задачи №2:

**Дано число – количество чисел в массиве. Далее через пробел дано число – количество итераций.**

Прошлой представленной курсовой работой невозможно было получить такой результат в общем случае. Можно было сделать что-то типа:

CompositeTest test;

int start = 0;

int end = 100000;

Range<int> n\_range(start, end);

Int n(n\_range);

Range<int> m\_range(n.Get(), end);

Int m(m\_range);

test.Add(n)

test.Add(new\_line\_delimiter)

test.Add(m);

И это будет работать. Но только для одного теста. Если после этого вызвать Print(), то выведется всё верно, но ниже будет показано, что может быть, если вызвать Generate():

1. устанавливается в
2. устанавливается в
3. устанавливается в
4. устанавливается в
5. вызывая получаем
6. вызываем
7. устанавливается в
8. устанавливается в (потому что его промежуток всё так же остался )
9. вызывая получаем , что противоречит условию .

Однако создать такой тест можно сделав такой «трюк»:

While(1)

{

CompositeTest test;

int start = 0;

int end = 100000;

Range<int> n\_range(start, end);

Int n(n\_range);

Range<int> m\_range(n.Get(), end);

Int m(m\_range);

test.Add(n)

test.Add(new\_line\_delimiter)

test.Add(m);

test.Print();

}

То есть не вызывать Generate(). Такое конечно сделать можно, но есть много причин, почему такое делать не стоит. Да и изначально библиотека разрабатывалась с той философией, что надо один раз создать тест, а потом нужное количество раз вызвать Generate() и Print(), так что последний вариант нас не устраивает.

Теперь давайте представим, что мы можем создать каким-то способом зависимые переменные. Давайте вспомним, как устроены классы в прошлой курсовой работе. Есть класс CompositeTest, который является совокупностью тестов поменьше. Это сделано для того, чтобы работать с совокупностью тестов как с одним тестом. Из верхних примеров можно увидеть, какой у этого класса интерфейс и как с ним работать. Под собой он, при каждом вызове функции Add(), вызывает на каждом тесте функцию Clone() и сохраняет у себя клон всего теста. Функция Clone() реализует шаблон проектирования Прототип и сделан это для того, чтобы если потом тест изменится, то осталось неизменно то, что было добавлено в CompositeTest. Однако, если мы создали зависимые переменные, то мы не сможем просто взять и скопировать (вызвать Clone()), прототип сработает так: после того, как мы добавим тест в CompositeTest, он вызовет на этом тесте функцию Clone(), тест же в свою очередь рекурсивно вызовет функцию Clone() для всех тестов от которых он зависит и так до того момента, пока Clone() не будет вызываться на примитивном тесте типа Int. Когда Clone() вызовется на примитивном типе, то создастся совершенный новый примитивный тип идентичный вызывающему и, поднимаясь по рекурсии, будет происходить тот же процесс везде. При этом все связи будут потеряны, потому что все объекты будут созданы заново на уровне примитивов, структура теста останется почти та же, но связей уже не будет. Чтобы пояснить, приведём пример какого-нибудь несложного теста.

# Архитектура библиотеки

# Классы

## Test

## PrimitiveTest<T>

## CompositeTest

## Range

## ConstPrimitiveTest<T>

## RangePrimitiveTest<T>

## PrimitiveTest<T>

## Delimeter

## Array

## ConstStringSet

## RandomTestSet

## Grammar

## RegEx

## Graph

# Разбор задач

# Решение задач и генерация ответов

# Организация олимпиад и связь с ejudge

# Проверка домашних заданий

# Публикация, распространение и использование библиотеки

# Возможные дальнейшие планы

# Источники

# Приложение

## Дополнительные функции

## Документация