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

**Институт математики и высоких технологий**

**Кафедра системного программирования**

**ДИПЛОМНАЯ РАБОТА**

***Тема:***

***Разработка библиотеки генерации тестов для задач по информатике***

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

**Научный руководитель: Цирунян Армен**

**Ереван 2017**

**Оглавление**

[Введение 3](#_Toc501408307)

[Описание работы 4](#_Toc501408308)

[Требования к библиотеке и задачи 5](#_Toc501408309)

[Необходимые понятия 7](#_Toc501408310)

[Разработка и актуальные проблемы 9](#_Toc501408311)

[Архитектура библиотеки 12](#_Toc501408312)

[Классы 13](#_Toc501408313)

[Test 13](#_Toc501408314)

[CompositeTest 13](#_Toc501408315)

[ConstPrimitiveTest<T> 14](#_Toc501408316)

[RangePrimitiveTest<T> 14](#_Toc501408317)

[Array 15](#_Toc501408318)

[Delimiter 17](#_Toc501408319)

[ConstStringSet 17](#_Toc501408320)

[RandomTestSet 17](#_Toc501408321)

[Grammar 17](#_Toc501408322)

[RegEx 18](#_Toc501408323)

[Graph 20](#_Toc501408324)

[GraphMerger 25](#_Toc501408325)

[AddDeleteResponsibility 26](#_Toc501408326)

[SetPostprocessFunction 27](#_Toc501408327)

[Разбор задач 27](#_Toc501408328)

[Решение задач и генерация ответов 29](#_Toc501408329)

[Организация олимпиад и связь с ejudge 30](#_Toc501408330)

[Проверка домашних заданий 30](#_Toc501408331)

[Публикация, распространение и использование библиотеки 30](#_Toc501408332)

[Пути развития 31](#_Toc501408333)

[Заключение 31](#_Toc501408334)

[Источники 32](#_Toc501408335)

# 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 во избежание 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 регулярному выражению 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 2017 года. Было представлено 8 задач, некоторые из них будут описаны ниже.

# Описание работы

TestMaker – библиотека, предоставляющая интерфейсы генерации различных тестов для олимпиад и домашних заданий. Реализована на языке С++. При разработке были использованы многие шаблоны проектирования, алгоритмы, была осуществлена работа в нескольких потоках.

Библиотека предоставляет такие мощные инструменты как диапазоны значений, массивы на диапазонах, случайные графы, регулярные выражения и многое другое. Ниже обо всем этом будет подробно рассказано. Приятного чтения.

# 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>***

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)

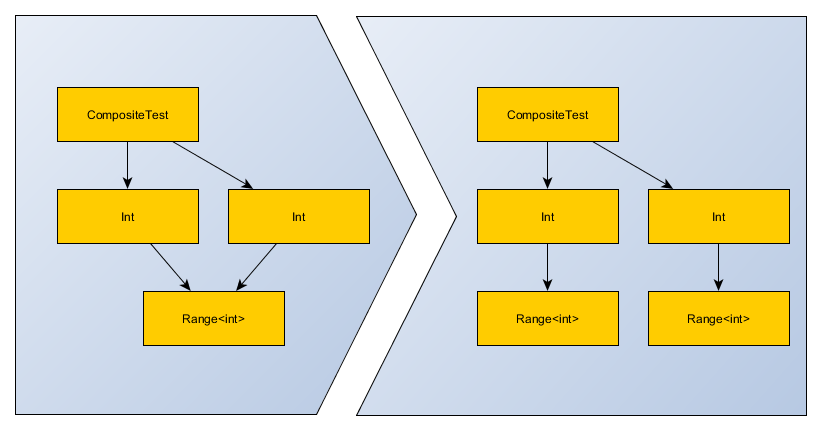
test.Add(m);

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.

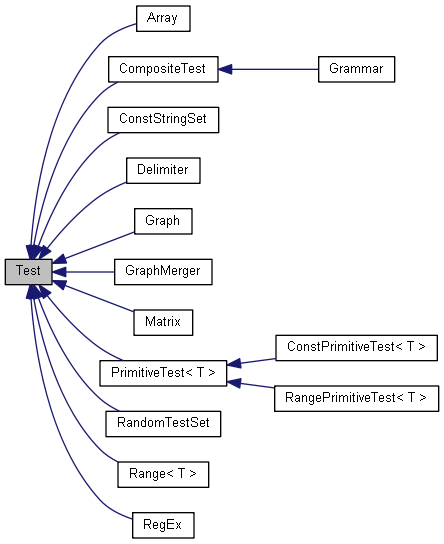
Теперь давайте представим, что мы можем создать каким-то способом зависимые переменные. Давайте вспомним, как устроены классы в прошлой курсовой работе. Есть класс CompositeTest, который является совокупностью тестов поменьше. Это сделано для того, чтобы работать с совокупностью тестов как с одним целым. 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 сделано 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 (например, 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.  
Слева – структура теста, которая была изначально. В нем созданы два объекта типа Int, который ссылаются на тот же объект Range<int>, чтобы в них было то же самое значение. Справа – структура того же теста, только после вызова функции Clone(). Можно заметить, что Int-ы теперь имеют разные объекты Range<int> под собой и, поэтому, их значения в общем случае не будут одинаковыми.



Да и кроме всего этого, библиотека была довольно бедной, и ею нельзя было сделать большинство видов тестов. В данной версии возможности библиотеки сильно расширены, и обо всех возможностях читайте ниже.

# Архитектура библиотеки

Архитектура библиотеки стала намного более комплексной, если сравнивать с прошлым результатом. Ниже представлена картинка, описывающая текущую архитектуру:



Из картинки видно, что Test является базовым классом для всех остальных. Ниже будут описаны все классы, но описание будет не полным. Для ознакомления со всеми методами обратитесь к официальной документации.

# Классы

## Test

Test – базовый абстрактный класс всей библиотеки. В нем содержится весь необходимый функционал, который наследуют остальные классы. Реализует шаблон проектирования Компоновщик.

## CompositeTest

CompositeTest – ещё один класс, который реализует шаблон проектирования Компоновщик. Является совокупностью нескольких (может одного) тестов, и имеет тот же интерфейс, что и остальные тесты, что позволяет работать группой тестов как с одним. Имеет функцию Add, которой нужно добавлять тест к остальному множеству.

**Range<T>**

Range – класс, который отвечает за диапазон объектов. Принимает два объекта типа PrimitiveTest<T>, которые являются началом и концом диапазона (почему принимает PrimitiveTest<T> будет описано ниже). При запросе, возвращает случайный объект в заданном диапазоне. Так же наследуется от Test, но не предполагается использовать как отдельный тест. Используется для инкапсуляции функций генерирования случайных чисел при создании примитивных типов. В данный момент поддерживаются типы int, double, char, long long.

**PrimitiveTest<T>**

Описывает примитивные типы и является базовым классом для ConstPrimitiveTest<T> и RangePrimitiveTest<T>.

## ConstPrimitiveTest<T>

ConstPrimitiveTest <T> принимает один объект типа T. Используется для того, чтобы определять константные числа. Например, чтобы использовать число в контексте этой библиотеки, нужно создать

ConstPrimitiveTest<double>\* pi = new ConstPrimitiveTest<double>(3.14);

и использовать её в дальнейшем коде. Данный класс так же используется для создания RangePrimitiveTest<T>.

## RangePrimitiveTest<T>

RangePrimitiveTest <T> принимает один объект типа Range<T>. Это является диапазоном нашего элементарного теста. При этом можно получить довольно сложную структуру теста, и это и почти решает нашу проблему с зависимыми переменными. Раньше, мы делали что то такое:

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);

чтобы получить хоть какую то зависимость. Но сейчас можно делать и вот так:

PrimitiveTest<int>\* start\_number = new ConstPrimitiveTest<int>(10);

PrimitiveTest<int>\* end\_number = new ConstPrimitiveTest<int>(100);

Range<int>\* number\_range1 = new Range<int>(start\_number, end\_number);

PrimitiveTest<int>\* number1 = new RangePrimitiveTest<int>(number\_range1);

Range<int>\* number\_range2 = new Range<int>(start\_number, number1);

PrimitiveTest<int>\* number2 = new RangePrimitiveTest<int>(number\_range2);

Таким образом, можно получить диапазон вот такого типа [10, [10, 100]]. Это значит, что начальная граница нашего диапазона является 10, а конечная граница является в свою очередь диапазоном от 10 до 100. Так мы и получаем число, которое зависит от другого числа. Назовем ***связанными*** тесты, в которых мы используем такую связь переменных. Назовем ***свободными*** все остальные тесты. Данные термины будут использоваться в тексте далее.

Теперь поговорим об устройстве данной конструкции, и что происходит снизу, когда мы собираем число такого типа.

Сперва мы строим PrimitiveTest <int> start\_number и end\_number, которые являются ConstPrimitiveTest <int> и принимают 10 и 100 соответственно. Потом мы создаём Range<int> number\_range1, который принимает start\_number и end\_number, и уже из него мы создаем число RangePrimitiveTest <int>, которое является числом с диапазоном от 10 до 100. После этого мы создаём новый Range<int>, который принимает start\_number и только созданный RangePrimitiveTest <int> в качестве аргументов, и создает число, диапазоном которого является [10, [10, 100]]. Для более глубокого понимания обратитесь к официальной документации или к коду.

## Array

Для того, чтобы создать массив Тестов нужно использовать класс Array. При этом, в нем есть возможность для хранения как ***связанных***, так и ***свободных*** тестов. Для свободных тестов ничего не изменилось с прошлой версии, Array всё так же принимает размер массива, элемент-шаблон, по которому генерируется весь массив (вызовом функции Clone()), и разделители строк и элементов. Для связанных же тестов все меняется, потому что если вызвать функцию Clone(), то это приведет к непредсказуемому поведению программы. Для этого была введена дополнительная возможность для передачи в Array ***образующей функции*** –функции, которая **строит** и **возвращает** объект типа Test\*, и это тест, по образу которого мы хотим создать массив. Теперь в самом массиве, вместо того, чтобы клонировать, мы каждый раз вызываем функцию, которая создаёт наш тест и уже не клонируем его, а только держим указатель на объект, который создала образующая функция и все связи сохраняются. При этом, после создания и возвращения, функция передаёт право владения (ownership) объектом тому, кто её вызвал. Так как Array принимает Test\*, который может быть сколь угодно сложным, то и конструкция может быть многомерной и очень комплексной. Но в некоторых случаях организовать многомерный массив является сложной задачей, и вот описание такого примера.

Предположим, нам надо создать матрицу размера nxm, где и n и m лежат в промежутке от 1 до 10. Попробуем написать такой тест:

Array\* arr1 = new Array(CreateElement(1, 10), CreateElement(1), " ", " ");

arr1->PrintSize(false);

Array\* arr2 = new Array(CreateElement(1, 10), arr1, "\n");

arr2->Generate()->Print();

Сначала разъясним все незнакомые пока функции:

* CreateElement(a, b) – возвращает случайный элемент из промежутка [a, b]. Может принимать 1 параметр, что значит вернуть этот самый элемент, только в виде Test\*. Может принимать типы char, int, long long и double. При этом право владения переходит к нам и данный код приведет к утечке памяти.
* Третий параметр класса Array – это разделитель между элементами массива при выводе.
* Четвертый параметр класса Array – это разделить после вывода размера массива и после вывода всего массива. Если вызвать функцию arr1->PrintSize(false), то размер, как и разделитель выведены не будут.

Вернёмся к тесту. Вроде бы все должно сработать правильно, но выводится что-то типа такого:

9

1 1 1 1 1

1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1

1 1 1 1 1 1 1 1

1

1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1

Совсем не то, что нам требовалось. Дело в том, что когда мы создаём первый массив, то говорим, что его размер может варьироваться от 1 до 10. И когда мы создадим массив массивов, каждый элемент будет разной длины. Чтобы сделать матрицу, можно использовать класс Matrix (который имеет тот же интерфейс что и Array), но если потребуется структура посерьёзней, то придётся писать для этого свой собственный класс.

## Delimiter

Класс описывающий разделители. Если нужно поставить какие-то специальные разделители в тесте, например пробел или символ новой строки.

## ConstStringSet

Класс, содержащий строки, которые должны выводиться «как есть». Например, если надо вывести «Yes» или «No».

## RandomTestSet

Класс, который получает на ввод некоторое количество тестов и возвращает один из этих тестов в случайном порядке.

## Grammar

Класс, представляющий контекстно-свободную грамматику. Для понимания необходимы следующие термины из теории формальных языков:

* N — набор (алфавит) нетерминальных символов
* — набор (алфавит) терминальных символов
* P — набор правил вида: «левая часть» {\displaystyle \rightarrow } «правая часть», где:
  + «левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал
  + «правая часть» — любая последовательность терминалов и нетерминалов
* S — стартовый (или начальный) символ грамматики из набора нетерминалов.

Конструктор класса принимает 2 строки: первая это описание грамматики, вторая это набор правил. Например:

Grammar\* gr = new Grammar("({1,2,3}, {a,b,c}, P, 1)",

"${1}->${2}${3}\n${2}->a${2}\n${2}->e\n${3}->${3}b\n${3}->e");

gr->Generate()->Print();

Данный пример описывает следующую грамматику:

* N = {1, 2, 3} - нетерминальные символы
* = {a, b, c} - терминальные символы
* P будет описан ниже
* S = 1

Правила имеют следующий вид:

* 1 → 23
* 2 → a2
* 2 → e (пустой символ)
* 3 → 3b
* 3 → e (пустой символ)

Как можно заметить, нетерминалы нужно указывать в фигурных скобках с лидирующим знаком доллара. Это сделано для того, чтобы терминалы не путались с терминалами.

Итак, данная грамматика описывает языки вида:

Таким образом, можно описать любую контекстно-свободную грамматику и по этой грамматике сгенерировать тесты. Все сгенерированные тесты будут словами из языка, порождённым этой грамматикой.

## RegEx

Класс, представляющий регулярные выражения. Определим, что такое регулярное выражение.

Регулярные выражения состоят из [констант](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [операторов](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), которые определяют [множества](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) [строк](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и множества [операций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) на них соответственно. На данном конечном [алфавите](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D1%84%D0%B0%D0%B2%D0%B8%D1%82) Σ определены следующие константы:

* ∅ - [пустое множество](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D1%81%D1%82%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE)
* ε обозначает строку, не содержащую ни одного символа – пустая строка. Эквивалентно «».
* "a", где "a" - символ алфавита Σ - [символьный литерал](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85))
* Метасимволы или символы-джокеры - с[имволы, используемые для замены других символов или их последовательностей, приводя таким образом к символьным шаблонам.](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B8) Ниже перечислены все метасимволы(что значат квадратные скобки будет объяснено ниже):
  + \s - эквивалентно [ \n\t]
  + \S - эквивалентно [^ \n\t]
  + \d - эквивалентно [0-9]
  + \D - эквивалентно [^0-9]
  + \w - эквивалентно [A-Za-z0-9\_]
  + \W - эквивалентно [^A-Za-z0-9\_]

и следующие операции:

* RS обозначает множество {αβ | α ∈ R & β ∈ S}. Например, {"a", "b"}{"c", "d"} = {"ac", "ad", "bc", "bd"}. Данная операция называется [сцепление или конкатенация](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%86%D0%B8%D1%8F).
* R|S обозначает [объединение](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%B4%D0%B8%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2) R и S. Например, {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}. Данная операция называется [дизъюнкция](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B7%D1%8A%D1%8E%D0%BD%D0%BA%D1%86%D0%B8%D1%8F), чередование.
* R\* обозначает минимальное [надмножество](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B4%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) множества R, которое содержит ε и [замкнуто](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D0%BA%D0%BD%D1%83%D1%82%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) относительно конкатенации. Это есть множество всех строк, полученных конкатенацией нуля или более строк из R. Например, {"a", "b"}\* = {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", …}. Данная операция называется итерация или замыкание Клини.
* R+ обозначает то же, что и итерация, только без учитывания пустого символа. Например, {"a", "b"}+ = {"a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", …}.
* R{n, m} обозначает минимальное [надмножество](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B4%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE) множества R, полученных конкатенацией от n до m строк из R. Например, {"a", "b"}{2, 3} = {"aa", "ab", "ba", "bb", "aaa", "aab", "aba", …,"bbb" }.  
  Фигурные скобки из примеров использованы для понимания, часть, описывающая данный тип скобок только «{2, 3}».
* (R) - группа, группирует несколько элементов (например, (\w \d\n){2,7} - сгенерировать слово и число в строке от 2х до 7и раз)
* [ABC] – обозначает случайный один символ из перечисленных
* [^ABC] - обозначает случайный один символ из не перечисленных
* [A-C] - обозначает случайный один символ из данного промежутка
* [^A-C] - обозначает случайный один символ не из данного промежутка

Регулярные выражения, реализованные в данной работе отличаются от классических регулярных выражений, реализованных в текстовых редакторах и языках программирования, потому что надо *генерировать* строки, а не искать и заменять.

Конструктор класса принимает одну строку и одно необязательное целое число – максимальная глубина для операций \* и +, которое по умолчанию является 1000. Строка же является именно регулярным выражением, по которому и нужно генерировать строки. Рассмотрим один пример:

RegEx\* regex = new RegEx("([a-zA-Z]{100}\n)\*", 100);

regex->Generate()->Print();

Данное регулярное выражение генерирует буквенные строки длины 100 ([a-zA-Z]{100}) в новой строке до 100 раз (итерация за скобкой, последнее число 100 относится к итерации, ограничение сверху).

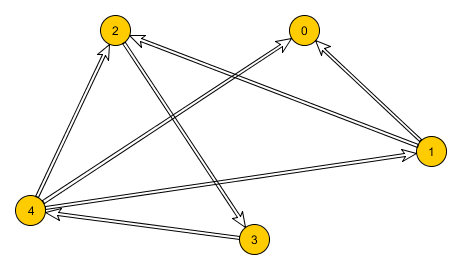
Применение регулярных выражений огромно. Например, сгенерировать правильный e-mail адрес, номер телефона, сгенерировать массивы слов/чисел изменяемой длины, и многое другое. Но есть и плохая сторона: например, если генерируется массив, то его размер узнать из регулярного выражения невозможно, оно просто сгенерирует массив, без его размера. Дело в том, что само выражение не знает, когда оно остановится, оно работает, пока не будет какое-то останавливающее его условие, а условие может быть довольно сложным и не тривиальным для предподсчёта. Но на этот счёт, у нас есть вариант постобработки вывода, о котором мы поговорим чуть позже.

## Graph

Graph – это класс описывающий графы. В данном классе поддерживаются создание графов как направленных, так и неориентированных, с петлями и без, и с определённым методом вывода.

На данный момент методов вывода три: матрица смежности, список смежности и список ребер. Опишем каждый из методов.

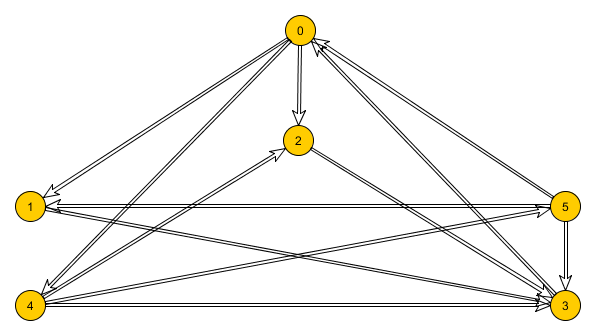
Матрица смежности – это матрица размера nxn, где n это количество вершин в графе. В ячейке (i, j) матрицы мы записываем 1, если нет ребра, проходящего из i в j и 0 в обратном случае. Таким образом, наша матрица может быть матрицей типа bool. Ниже пример такой матрицы для следующего графа:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 |
| 4 | 1 | 1 | 1 | 0 | 0 |

Первые строка и столбец описывают номера вершин, а оставшаяся подматрица и является матрицей смежности графа G. Очевидно, что с ростом количества вершин графа, матрица растёт в квадратном соотношении. В основном, графы бывают слишком большие для матрица смежности и тогда можно использовать список смежности.

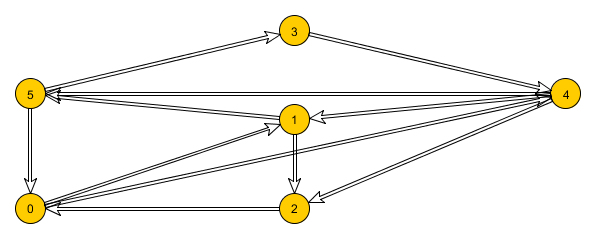
Список смежности, в отличии от матрицы, не содержит ненужные нули - в нем хранятся только ребра между вершинами. Но, так как это уже не матрица, записывать 0 и 1 мы уже не можем, потому что не сможем потом узнать с кем именно соседствует вершина. Поэтому вместо 0 и 1 будем записывать номер соседствующей вершины. Ниже – пример:



|  |
| --- |
| G |
| 0 | 1 | 2 | 4 |
| 1 | 3 |
| 2 | 3 |
| 3 | 0 |
| 4 | 2 | 3 | 5 |
| 5 | 0 | 1 | 3 |

В первом столбце записаны номера вершин, а рядом с ними их соседи. С этим методом мы можем сэкономить место для хранения графа, так как не храним дополнительные ненужные поля. Но в случае почти полного графа, хранение списка смежности будет не эффективно, потому что в матрице мы храним всего один бит для каждого ребра, а в списке от 4 до 8и бит исходя из битности системы и размера графа. Кроме того, скорость обхода по соседям конкретной вершины так же увеличена, т.к. все соседи вершины описаны друг за другом. Но в случае почти полного графа скорость обхода та же что и в матрице, потому что количество исходящих из вершины ребер почти равно количеству вершин.

И последний метод вывода это список ребер.



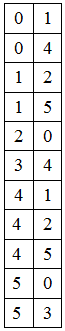


Таблица справа является списком ребер графа описанного выше. Преимущества этой записи является быстрым поиском по ребрам, и относительная экономия памяти по сравнению с листом и матрицей смежности.

Эти три метода вывода поддерживаются при выводе сгенерированных графов.

Все приведенные примеры являются ориентированными графами, но графы могут быть так же неориентированными.

Построение графа была одной из самых сложных задач данной работы. Граф сам по себе имеет много вариаций, и нужно было поддерживать хотя бы самые примитивные: ориентированный, неориентированный, дерево, ориентированный без циклов. Изначально в данной работе было 3 класса для графов: Graph (базовый класс), DirectedGraph и NonDirectedGraph. Так чтобы сгенерировать, например, дерево, надо было создать NonDirectedGraph и указать флаг «дерево». Алгоритмы построения дерева и ориентированного графа без циклов описаны далее.

* В случае дерева, можно было его строить итеративно, добавляя каждый шаг по одному ребру, если это ребро не образует цикл. Весь вопрос был в том, чтобы понять, когда ребро образует цикл, а когда нет. Можно было просто на каждый шаг искать цикл, но это было бы слишком неэффективно, скорость построения была бы где-то O(m2) , где m – количество рёбер. Для того, чтобы быстро определять, образует ли добавленное ребро цикл, мы будем использовать структуру данных «система непересекающихся множеств». Эта структура позволяет контролировать множеством элементов, которые разбиты на непересекающиеся подмножества. При этом каждому подмножеству назначается его представитель — элемент этого подмножества. Система непересекающихся множеств определяется множеством двух операций: {Find, Union}. Мы будем держать k множеств, и когда нужно будет добавлять ребро, то узнаем представителей вершин обоих концов ребра и если они не равны, то ребро можно добавить и оно не будет образовывать цикл. В обратном же случае, надо будет выбирать другое ребро для добавления. Если выбирать рёбра случайным образом, то при этом может быть такая ситуация, что нужное ребро будет искаться очень долго. Для этого надо будет найти какой-то метод для не случайного выбора рёбер, но такой метод и не был найден.
* В случае ориентированного графа без циклов мы можем пронумеровать каким-то образом вершины, и проводить ребра с вершин с меньшим номером в вершины с большим. Очевидно, что цикла в таком случае не будет, т.к. из большей вершины никогда не будет проведено ребро в меньшую вершину.

Все эти алгоритмы работали не настолько быстро насколько хотелось. Для этого был придуман совершенно другой подход к построению графов.

Для начала мы избавились от ненужных классов DirectedGraph и NonDirectedGraph и оставили только Graph, в котором добавили возможность установить ориентированность ребер с помощью параметра. Далее было принято решение что наш граф будет всегда связанным и изначально будет сгенерировано дерево (орграф без циклов) и только потом генерироваться дальше. Таким образом, если передать графу n вершин и n – 1 ребро и указать, что мы хотим неориентированный граф, то гарантированно сгенерируется дерево.

При этом при генерации используется понятие маленького и большого графа. Маленьким графом является тот граф, в котором количество ребер меньше половины всех ребер. Большой граф тот, в котором количество ребер больше или равно половины всех ребер. Во время генерации, решается с каким графом мы имеем дело – большим или маленьким. Если граф маленький, то мы начинаем строить граф с фиксированного ранее дерева, а если граф большой, то мы берем полный граф и начинаем удалять из него ребра, не трогая ребра из фиксированного дерева. Выбирать рёбра будем случайно из всех. В данном случае, это будет не так плохо, потому что мы будем в худшем случае удалять половину рёбер, и, следовательно, вероятность попасть по ребру, которое мы взять не можем, стремится к 50%.

Конструктор Graph имеет такой вид:

Graph( PrimitiveTest<int>\* \_number\_of\_vertices,

PrimitiveTest<int>\* \_number\_of\_edges,

Test\* \_weights = nullptr,

bool \_directed = false,

bool \_buckle = false);

Опишем аргументы по очереди.

1. Количество вершин. Принимает PrimitiveTest<int>, что является диапазоном.
2. Количество ребер. Принимает так же PrimitiveTest<int>. Величина графа определяется только после фиксации числа из диапазона.
3. Веса графа. Принимает тест любого типа. Веса могут выводиться, только если метод вывода установлен в лист ребер и будут выводиться через пробел после самого ребра.
4. Устанавливает, направлен ли граф (можно менять после создания графа).
5. Устанавливает, есть ли в графе петли (можно менять после создания графа).

Но стоит не забывать, что сгенерированные графы всегда связные. А что если нам надо сгенерировать не связные? В этом нам поможет GraphMerger.

## GraphMerger

GraphMerger – класс, который объединяет графы для получения не связанных графов. В конструкторе получаем или массив или переменное количество графов, которое впоследствии объединяем. Объединение происходит следующим методом: получаем все графы в виде матриц, листов смежности или листов ребер и узнаем их размеры. Суммируем их размеры, создаем массив суммарного размера и под i-тым индексом пишем число i и после этого случайным образом мешаем массив. При этом каждая вершина каждого графа получает свой уникальный номер. При выводе вместо того, чтобы выводить настоящий номер вершины выводим её новый номер и объединяем все графы в один. Сделано это для того, чтобы вершины отдельных графов имели случайные номера, а не последовательные.

На этом краткое описание классов окончено, для более глубокого ознакомления обратитесь к официальной документации или коду.

Но перед тем как пройти к разбору задач, нужно рассказать ещё про две функции, которые есть во всех классах.

# AddDeleteResponsibility

Функция AddDeleteResponsibility была создана для того, чтобы контролировать владение ресурсом. Иногда бывают моменты, когда мы создаем объекты в функции, которые нужны для конструирования другого объекта, теряем контроль над ними и не можем удалить. Для этого, мы можем добавить объекту право владением этими переменными, чтобы только при его удалении удалились и они. Вот пример использования данной функции:

template <typename T = int>

inline PrimitiveTest<T>\* CreateElement(T \_start, T \_end)

{

ConstPrimitiveTest<T>\* start\_number = new ConstPrimitiveTest<T>(\_start);

ConstPrimitiveTest<T>\* end\_number = new ConstPrimitiveTest<T>(\_end);

PrimitiveTest<T>\* number = CreateElement<T>(start\_number, end\_number);

number->AddDeleteResponsibility(start\_number);

number->AddDeleteResponsibility(end\_number);

return number;

}

Это функция, которая создает элемент по диапазону. При этом указатели переменных start\_number и end\_number при выходе из функции будут потеряны. Чтобы этого не случилось, отдадим право владения той переменной, над которой мы всё ещё имеем контроль, в этом случае number.

# SetPostprocessFunction

Иногда бывают ситуации, где инструментов библиотеки не бывает достаточно. Для того, чтобы обработать такие ситуации был минималистично использован паттерн Стратегия. Мы просто передадим в класс функцию, которая работает с выводом и меняет его исходя их наших требований. Из самых актуальных проблем является то, что регулярные выражения не позволяют получить размер сгенерированного массива и можно обработав вывод получить размер массива. Применение данного инструмента колоссально. Вот пример использования:

void regexPostProcess(std::string& \_result)

{

for (int i = 0; i < \_result.size(); i++)

{

\_result[i]++;

}

}

...

RegEx\* regex = new RegEx("([ab]{100}\n)\*", 100);

regex->SetPostprocessFunction(regexPostProcess)->Generate()->Print();

Сгенерируем строки из a и b, а потом поменяем a на b и b на c.

На этом описание функционала окончено, и пришло время решить задачи, сформулированные вначале.

# Разбор задач

***Задача 1***

const int n = 16;

for (int i = 1; i <= n; i++)

{

CreateElement(i)->Generate()->Print();

new\_line\_delimiter->Generate()->Print();

}

***Задача 2***

PrimitiveTest<int>\* n = CreateElement(10);

PrimitiveTest<int>\* m = CreateElement(10);

Test\* pair = new CompositeTest();

pair->Add(CreateElement(-1000, 1000))

->Add(space\_delimiter)

->Add(CreateElement(-1000, 1000));

Array\* arr = new Array(n, pair, "\n");

arr->PrintSize(false);

Test\* test = new CompositeTest();

test->Add(n)

->Add(space\_delimiter)

->Add(m)

->Add(new\_line\_delimiter)

->Add(arr)

->Generate()

->Print();

***Задача 3***

ConstStringSet\* css = new ConstStringSet();

for (char ch = 'a'; ch <= 'z'; ch++)

{

css->Add(ch);

}

Array\* word = new Array(CreateElement(1, 32), css, "", "");

word->PrintSize(false);

Array\* test = new Array(CreateElement(1, 100000), word, "\n");

test->PrintSize(true);

test->Generate()->Print();

И для худшего случая:

ConstStringSet\* css = new ConstStringSet();

css->Add('a');

Array\* word = new Array(CreateElement(1, 32), css, "", "");

word->PrintSize(false);

Array\* test = new Array(CreateElement(1, 10), word, "\n");

test->PrintSize(true);

test->Generate()->Print();

***Задача 4***

GraphMerger\* gm = new GraphMerger();

int numbers\_of\_trees = CreateElement(1, 100)->Generate()->Get();

int max\_count\_of\_vertices = 100000;

for (int i = 0; i < numbers\_of\_trees; i++)

{

int number\_of\_vertices = CreateElement(1, max\_count\_of\_vertices)

->Generate()->Get();

if (number\_of\_vertices <= 1)

break;

Graph\* tree = new Graph(CreateElement(number\_of\_vertices),

CreateElement(number\_of\_vertices - 1));

gm->Add(tree);

}

gm->PrintType(Graph::CONNECTION\_LIST)->Generate()->Print();

***Задача 5***

RegEx\* regex = new RegEx("[a-z]+(\\.[a-z]+)\*@[a-z]+(\\.[a-z]+)\*", 2);

regex->Generate()->Print();

Стоит заметить, что код можно написать не только так, и способ написания может варьироваться от программиста к программисту.

# Решение задач и генерация ответов

Сгенерировать тесты это, конечно, важно, но чтобы сгенерировать по тестам правильные ответы нужно иметь готовые решения задач. После того, как правильные коды задач будут написаны, нужно прогнать тесты по решениям и сгенерировать для каждой задачи свои ответы. Для этого на языке shell написан один скрип, который и делает всю эту работу.

# Организация олимпиад и связь с ejudge

В 2017 году была проведена одна олимпиада на автоматической проверочной системе ejudge и есть планы по организации такого рода олимпиад намного чаще. Для того, чтобы организовать соревнования, достаточно, чтобы кто-нибудь придумал задачу и написал правильный код для неё, после этого, с помощью TestMaker сгенерируются тесты, с помощью shell скрипта создадутся правильные ответы на тесты и всё готово, чтобы начать соревнование. Процесс подготовки к соревнованию может длиться от 2 до 7 дней, смотря насколько сложные будут задачи, и тесты к ним. Но всегда бывают тесты, которые нужно писать руками и от этого никуда не деться, потому что для каждой задачи худшие случаи разные и обработать по какому-то шаблону это не получится.

# Проверка домашних заданий

Так же как и для олимпиады, данную библиотеку можно использовать для проверки домашних заданий с абсолютно такой же стратегией. Просто, иногда, домашние задания бывают настолько легкими, что не стоит даже запускать их, просмотрев глазами можно увидеть ошибки и недочеты. Но в общем случае, библиотека применима и к домашним заданиям.

# Публикация, распространение и использование библиотеки

Библиотека в скором времени будет опубликована в открытом доступе в онлайн ресурсе github, и все, кто захотят ею пользоваться, будут иметь эту возможность. Кроме того, у библиотеки есть подробная документация с примерами, которой люди со всего мира cмогут пользоваться.

# Пути развития

На данный момент библиотека не содержит каких-либо больших, явных недочётов. В дальнейшем библиотека будет дорабатываться, будут добавляться новые возможности, и улучшаться старые. Каких-либо больших нововведений пока не предусматривается.

# Заключение

На этом заканчивается описание библиотеки TestMaker. Проект ещё не совсем окончен, но приближается к своему завершению. В результате этой работы было изучено много нового – шаблонов проектирования, мультипоточности, архитектуры библиотеки и много нового. В дальнейшем будут новые профессиональные вершины, новые проекты, новые завоевания. И на этом дипломная работа является оконченной, спасибо за внимание.

# Источники

* **microsoft.com**
* **wikipedia.com**
* **stackoverflow.com**
* **Introduction to Algorithms,**[**T. H. Cormen**](https://en.wikipedia.org/wiki/Thomas_H._Cormen)**,**[**C. E. Leiserson**](https://en.wikipedia.org/wiki/Charles_E._Leiserson)**,**[**R. L. Rivest**](https://en.wikipedia.org/wiki/Ron_Rivest)**,** [**C. Stein**](https://en.wikipedia.org/wiki/Clifford_Stein)**.**
* **Design Patterns: Elements of Reusable Object-Oriented,** [**E. Gamma**](https://en.wikipedia.org/wiki/Erich_Gamma)**, R. Helm,**[**R. Johnson**](https://en.wikipedia.org/wiki/Ralph_Johnson_(computer_scientist))**,** [**J. Vlissides**](https://en.wikipedia.org/wiki/John_Vlissides)**,** [**G. Booch**](https://en.wikipedia.org/wiki/Grady_Booch)**.**