ГОУ ВПО РОССИЙСКО-АРМЯНСКИЙ (СЛАВЯНСКИЙ) УНИВЕРСИТЕТ

Институт математики и информатики

Кафедра системного программирования

ДИПЛОМНАЯ РАБОТА

Тема:

Разработка библиотеки генерации тестов для задач по информатике

Выполнил студент 4-ого курса: Оганесян Левон Эдгарович

Научный руководитель: Цирунян Армен

Ереван 2018

Оглавление

Введение

Проверка домашних заданий является наиболее долгой и сложной частью преподавательского труда. Из-за огромного количества домашних заданий качество проверки сильно падает, и преподаватель упускает из виду мелкие детали. Во избежание таких проблем была создана библиотека, которая позволяет создавать тестовые примеры, и проверять корректность решений присылаемых студентами.

Кроме этого, данной библиотекой возможно генерировать тестовые примеры для олимпиадных задач, намного облегчая весь процесс организации олимпиад.

16-ого сентября 2017 года была проведена олимпиада для первого и второго курсов с использованием данной библиотеки. Было представлено 8 задач, некоторые из них будут описаны в разделе «Требования к библиотеке и задачи».

Данная работа является продолжением курсовой работы.

Описание работы

Данная работа – библиотека для генерации различных тестовых примеров олимпиад и домашних заданий. Реализована на языке C++. При разработке были использованы шаблоны проектирования, алгоритмы, многопоточное программирование.

Библиотека предоставляет такие инструменты как диапазоны значений, массивы на диапазонах, случайные графы, регулярные выражения, грамматики и многое другое.

Требования к библиотеке и задачи

Для определения требований, нам нужно сформулировать задачи, с которыми должна будет справиться данная библиотека:

Задача №1

<Условие задачи>

Исходные данные

Дано целое число n ($1 \leq n \leq 16$) – длина перестановки.

<Описание результата>

Сгенерировать тестовые примеры данной задачи не представляет труда, но в ней есть одна особенность: количество всех возможных тестовых примеров всего 16 (1, 2, …, 16). Если сгенерировать много случайных тестов, то есть вероятность, что все варианты будут учтены, но мы сделаем иначе и сгенерируем именно то, что нам нужно.

Задача №2

<Условие задачи>

Исходные данные

Дано число n ($1 \leq n \leq 10^5$) – количество чисел в массиве. Далее через пробел дано число m ($n \leq m \leq 10^5$) – количество итераций. На следующих n строках записаны пары чисел $a_i b_j$, каждое из которых может быть по модулю не больше 1000.

<Описание результата>

Задачей №2 мы покажем, как устроены наши типы, массивы, и зависимости типов друг от друга.

В задаче №3 нужно привести часть условия, потому что от этого напрямую зависит, какого типа будет наш тест:

Задача №3

<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 n, m ($1 \leq n \leq 10^5, 1 \leq m \leq 10^5$), где n это количество вершин в графе, а m - количество ребер. Далее в m строках будут перечислены пары целых чисел u_i v_i ($1 \leq u_i, v_i \leq n$), это значит, что присутствует ребро между вершинами u_i и v_i. В графе нет петель и кратных ребер.

<Описание результата>

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

Задача №4

<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 10^6 байт.

<Описание результата>

Последняя задача покажет нам работу с регулярными выражениями.

Итак, требования к библиотеке:

- Примитивные типы
- Зависимые друг от друга переменные
- Массивы
- Графы
- Регулярные выражения
- Грамматики
- Возможность сгенерировать любой тест
- Класс для обработки множества тестовых примеров одной задачи
- И многое другое.

Используемые 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).

Разработка и актуальные проблемы

В курсовой работе третьего курса были приведены примеры задач и то, как можно создать для них тесты, но некоторые виды задач не было возможности решить теми методами из-за неверной архитектуры библиотеки, например, задачи, в которых переменные каким-то образом зависят друг от друга (назовем эти переменные **зависимыми**). Для примера рассмотрим первую часть задачи №2:

Дано число n ($1 \leq n \leq 10^5$) – количество чисел в массиве. Далее через пробел дано число m ($n \leq m \leq 10^5$) – количество итераций.

Прошлой представленной курсовой работой невозможно было получить переменную m, которая всегда больше переменной n или как-то зависела от другой переменной. Но была возможность сделать что-то типа:

```
1.  CompositeTest test;
2.  int start = 0;
3.  int end = 100000;
4.  Range<int> n_range(start, end);
5.  Int n(n_range);
6.  Range<int> m_range(n.Get(), end);
7.  Int m(m_range);
8.  test.Add(n)
9.  test.Add(new_line_delimiter)
10. test.Add(m);
11. test.Print();
12. test.Generate();
13. test.Print();
```

Здесь мы создаем CompositeTest (который является множеством тестов, строчка 1), создаем переменную n (строчки 2-5), создаем переменную m (строчки 6-7), указывая вместо левой её границы значение переменной n и добавляем всё это в тест (строчки 8-10). После вызова функции Print() (строчка 11) выведется всё верно, но функция Print() на строке 13 будет вести себя иначе, потому что, если вызвать Generate(), то структура теста изменится. Произойдёт следующее:

1. n_range устанавливается в $(0, 100000)$
2. n устанавливается в 100*
3. m_range устанавливается в $(100, 100000)$
4. m устанавливается в 200
5. вызывая $Print()$ получаем $n = 100, m = 200$
6. вызываем $Generate()$
7. n устанавливается в 10000
8. m устанавливается в 1000 (потому что его промежуток всё так же остался $(100, 100000)$)
9. вызывая $Print()$ получаем $n = 10000, m = 1000$, что противоречит условию $n \leq m$.

* Все числа приведены в виде примера.

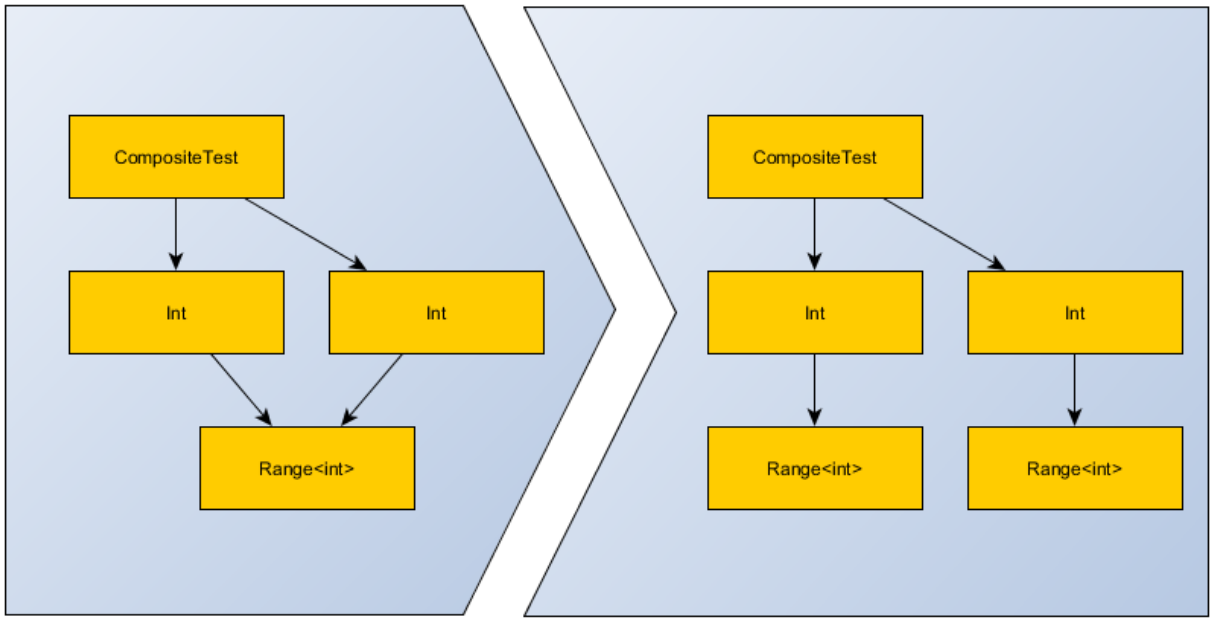
Однако создать тест возможно сделав такой «трюк»:

```
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() и это будет работать так, как и ожидается, но есть несколько причин, почему так делать не стоит. Да и изначально библиотека разрабатывалась с философией «один раз создать, много раз использовать», так что вариант с большим кодом в цикле нас не устраивает.

Теперь давайте представим, что мы можем создать каким-то способом зависимые переменные. Вспомним, как были устроены классы в прошлой, курсовой работе. Был класс CompositeTest, который являлся совокупностью тестов поменьше. Это было 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. Когда вызывается функция Add(), то на переданном в неё тесте вызывается функция Clone(), и CompositeTest сохраняет у себя именно клон всего теста. Функция Clone() реализует шаблон проектирования Прототип с целью сохранения копии теста в CompositeTest

неизменной, в том случае, если исходный тест вдальнейшем изменится. Однако, если была создана зависимая переменная, то нельзя просто скопировать тест (вызвать Clone()), потому что прототип рекурсивно начнет вызывать функции клонирования подтестов данного теста и так далее и каждый тест будет возвращать свою совершенно новую копию. При этом все связи будут потеряны, потому что все объекты будут созданы заново на уровне примитивов. Рассмотрим это на примере какого-нибудь несложного теста:

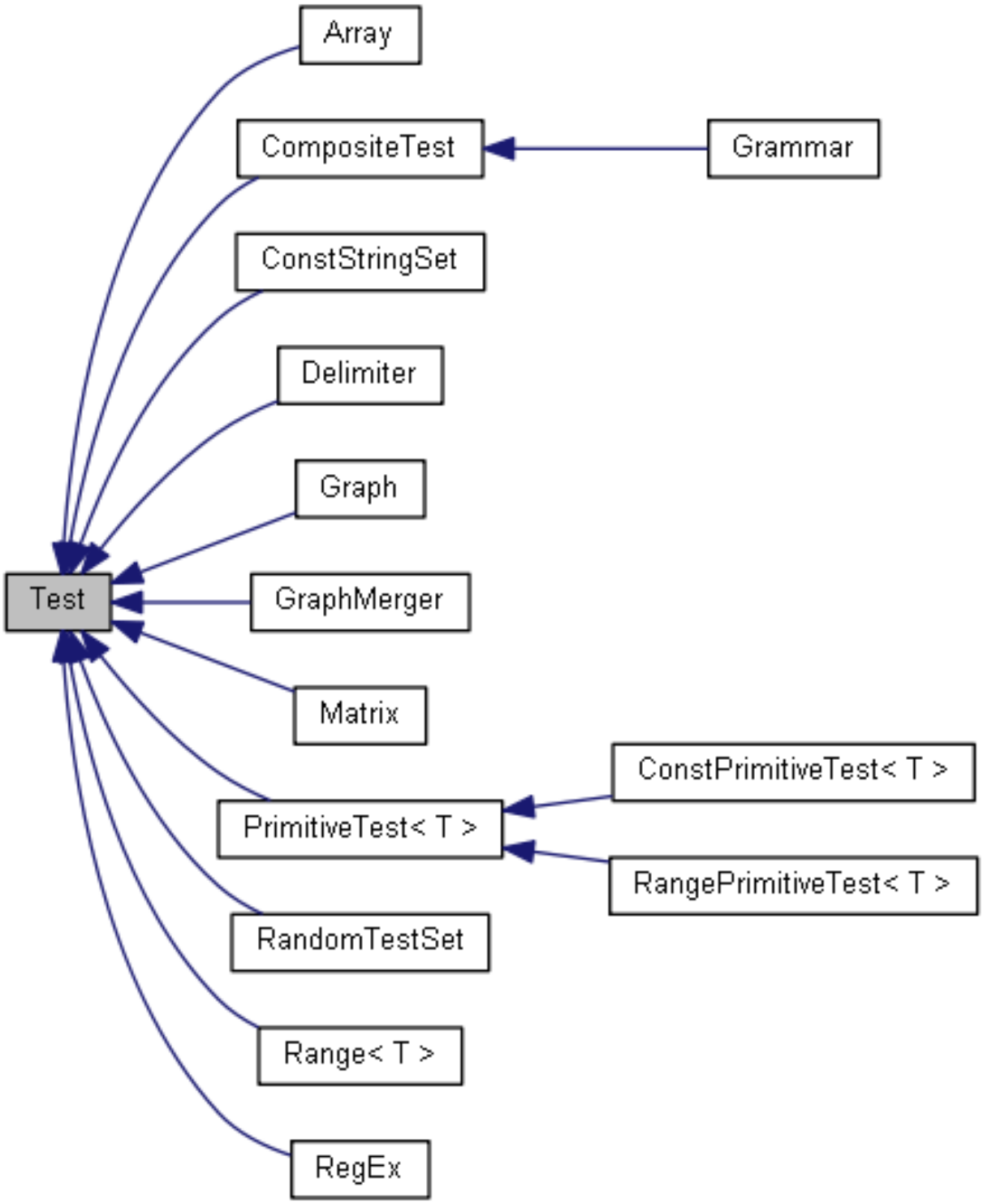


Слева – структура теста, которая была изначально. В нем созданы два объекта типа Int, который ссылаются на тот же объект Range<int>. Справа – структура того же теста, только после вызова функции Clone(). Можно заметить, что Int-ы теперь имеют разные объекты Range<int> и, поэтому, их значения в общем случае не будут одинаковыми.

Это лишь один пример некорректной работы библиотеки. Также нельзя было сделать большое множество часто встречающихся тестов. **В данной версии, возможности библиотеки сильно расширены.**

Архитектура библиотеки

Архитектура библиотеки сильно расширилась, если сравнивать с прошлым результатом. Ниже представлена схема, описывающая текущую архитектуру:



Из схемы видно, что Test является базовым классом для всех остальных. Далее следует описание всех классов. Для подробного 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 классы. Это позволяет работать с группой тестов как с одним. Исключением является функция Add(), которая добавляет тест к множеству, она реализована только в этом классе.

PrimitiveTest<T>

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

Range<T>

Range – класс, который отвечает за диапазон элементов. Принимает два объекта типа PrimitiveTest<T>, которые являются началом и концом диапазона. Функция Get() возвращает случайный объект в заданном диапазоне. Так же наследуется от Test и используется для инкапсуляции функций генерирования случайных чисел при создании примитивных типов. В данный момент поддерживаются типы int, double, char, long.

ConstPrimitiveTest<T>

ConstPrimitiveTest <T> принимает один объект типа T. Используется для того, чтобы определять константные числа. Например, чтобы использовать число π 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 диапазонов, и это почти решает нашу проблему с зависимыми переменными. Раньше, мы делали вот так

```
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 сделать так:

```
1. PrimitiveTest<int> start_number(10);
2. PrimitiveTest<int> end_number(100);
3. Range<int> number_range1(start_number, end_number);
4. PrimitiveTest<int> number1(number_range1);
5. Range<int> number_range2(start_number, number1);
6. PrimitiveTest<int> number2(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. Можно заметить, что код стал значительно громоздкий и это является одной из актуальных проблем библиотеки. Назовем ***зависимыми*** тесты, в которых мы используем такую связь переменных. Назовем ***независимыми*** все остальные тесты. Данные термины будут использоваться в тексте далее.

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

Сначала (строки 1-2) мы строим PrimitiveTest <int> start_number и end_number, которые являются ConstPrimitiveTest <int> и принимают целые числа 10 и 100 соответственно в аргументах конструктора. Далее (строка 3) мы создаём Range<int> number_range1, который в конструкторе принимает

start_number и end_number. Уже из него мы создаем число RangePrimitiveTest <int> number1 (строка 4), которое является числом с диапазоном от 10 до 100. После этого мы создаём новый Range<int> number_range1 (строка 5), который принимает start_number и только созданный number1 в качестве аргументов, и создает число (строка 6), диапазоном которого является [10, [10, 100]]. Подробнее с данными классами можно ознакомиться в официальной документации.

Array

Для того, чтобы создать массив тестов нужно использовать класс Array. При этом, в нем есть возможность для хранения как ***зависимых***, так и ***независимых*** тестов. Для независимых тестов ничего не изменилось с прошлой версии, Array всё так же принимает размер массива, элемент-шаблон, по которому генерируется весь массив (вызовом на нем функции Clone()), и разделители строк и элементов. Для зависимых же тестов все меняется, потому что если вызвать функцию Clone(), то структура теста разрушится. Для этого была введена дополнительная возможность для передачи в Array ***образующей функции*** – функции, которая **строит** и **возвращает** объект типа Test*, и данными тестами заполняется весь массив. Теперь, для каждого теста один раз вызывается данная функция, хранится указатель на объект, который она вернула и, таким образом, все зависимости теста сохраняются. При этом, после создания и возвращения, функция передаёт право владения объектом тому, кто её вызвал.

Так как Array содержит Test*-и, которые могут быть сколь угодно сложными, то и итоговая конструкция может быть многомерной и так же довольно сложной. Но в некоторых случаях организовать многомерный массив является сложной задачей, и вот описание такого примера.

Предположим, нам надо создать матрицу размера $n_x m$, 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();
```

Для начала 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 вызывающему и, если не очистить данный объект, то это приведет к утечке памяти.
- Первый и второй аргументы это соответственно размер массива и элемент, по которому весь массив должен сгенерироваться.
- 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 следующий за ним выведены не будут.

Вернёмся к тесту. Может казаться, что 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
```

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

Delimiter

Класс, описывающий разделители. Используется, когда 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 некоторое количество тестов и при запросе возвращает один из этих тестов в случайном порядке.

Grammar

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

- N — набор (алфавит) нетерминальных символов
- ∑ — набор (алфавит) терминальных символов
- P — набор правил вида: «левая часть» → «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 – правила представлены вторым параметром грамматики
- S = 1

Правила имеют следующий вид:

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

Нетерминалы нужно указывать в фигурных скобках с лидирующим знаком доллара. Так нетерминалы явно отличаются от терминалов и их сложно спутать.

Данная грамматика описывает языки вида:

$$L = \{a^n b^m \mid n \geq 0, m \geq 0\}$$

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

RegEx

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

Регулярные выражения состоят из констант и операторов, которые определяют множества строк и множества операций на них соответственно. На данном конечном алфавите Σ определены следующие константы:

- ∅ - пустое множество
- ε обозначает строку, не содержащую ни одного символа – пустая строка. Эквивалентно «».
- "a", где "a" - символ алфавита Σ - символьный литерал

- Метасимволы - символы, используемые для замены других символов или их последовательностей, приводя таким образом к символьным шаблонам. Метасимволов всего шесть:
    - \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.
- R|S обозначает объединение R и S. Например, {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}. Данная операция называется дизъюнкция, чередование.
- R* обозначает минимальное надмножество множества R, которое содержит ε и замкнуто 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 учета пустого символа. Например, {"a", "b"}+ = {"a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", …}.
- R{n, m} обозначает минимальное надмножество множества R, полученных конкатенацией от n до m строк из R. Например, {"a", "b"}{2, 3} = {"aa", "ab", "ba", "bb", "aaa", "aab", "aba", …,"bbb" }.
- (R) – данная операция называется группой. Группа объединяет несколько элементов (например, (\w \d\n){2,7} - сгенерировать слово и число в отдельной строке от 2х до 7и раз)
- [ABC] – обозначает случайный один символ из перечисленных
- [^ABC] - обозначает случайный один ASCII символ из не перечисленных
- [A-C] - обозначает случайный один символ из данного промежутка

- [^A-C] - обозначает случайный один ASCII символ не из данного промежутка

Регулярные выражения, реализованные в данной работе, отличаются от классических регулярных выражений, реализованных в текстовых редакторах и языках программирования, так как их основная цель искать или заменять строки, а не генерировать.

Конструктор класса принимает одну строку и одно необязательное целое число – максимальную глубину для операций * и +, которая 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 в отдельных строках до 100 раз (итерация в конце относится ко всей скобке, последнее же число 100 относится к итерации, и является для нее ограничением сверху).

У регулярных выражений довольно широкое применение. Например, сгенерировать правильный e-mail адрес, номер телефона, сгенерировать массивы слов/чисел изменяемой длины, и многое другое.

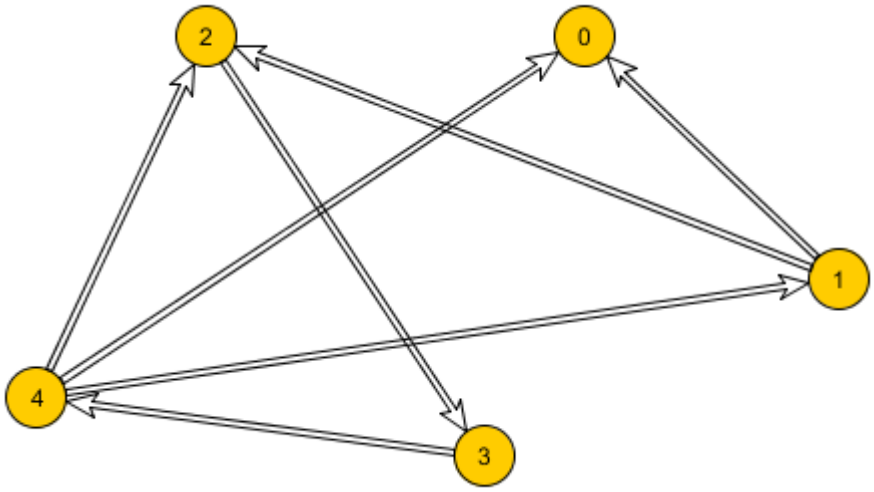
Но так же есть и недостаток, нельзя узнать размеры сгенерированных тестов в рамках самого регулярного выражения. Например, если генерируется массив, то его размер узнать из регулярного выражения невозможно, оно просто сгенерирует массив, без его размера. Дело в том, что само выражение не знает, когда произойдет остановка, оно работает, пока не случится терминальное условие, а условие может быть довольно сложным и не тривиальным для предподсчёта. На этот счёт есть вариант постобработки вывода, о нем можно прочитать в конце главы описания классов.

Graph

Graph – это класс описывающий графы. В данном классе поддерживается создание графов как ориентированных, 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.

Матрица смежности – это матрица размера $n_x n$, где n это количество вершин в графе. В ячейке (i, j) матрицы записано 0, если нет ребра, проходящего из i в j и 1 в обратном случае. Матрица имеет тип bool. Пример такой матрицы:

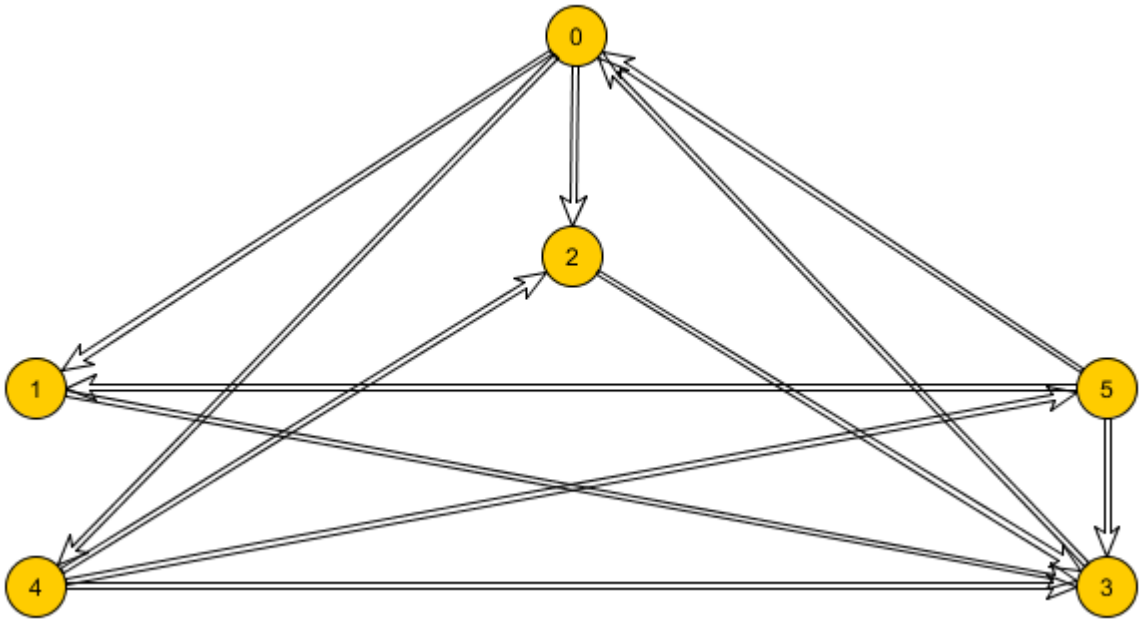


G	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	1	1	1	0	0

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 хранения их в матрицах смежности.

Список смежности, в отличие от матрицы, не содержит ненужные нули - в нем хранятся только ребра между вершинами. Но, так как нет однозначного соответствия для вершин, просто записывать 0 и 1 уже нельзя, потому что

нельзя будет в дальнейшем узнать с кем именно соседствует вершина. Поэтому вместо 0 и 1 будем записывать номер соседствующей вершины. Пример:

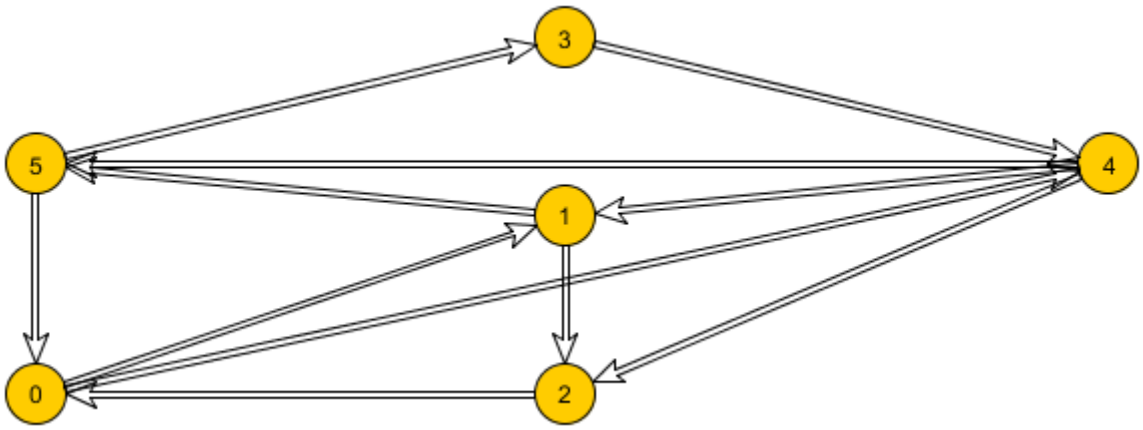


G			
0	1	2	4
1	3		
2	3		
3	0		
4	2	3	5
5	0	1	3

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 хранятся дополнительные ненужные поля. Но в случае почти полного графа, хранение списка смежности не будет эффективно, потому что в матрице мы храним всего один бит для каждого ребра, а в списке от 4 до 8-и байт, исходя из битности системы и размера графа. Кроме того, скорость обхода по соседям конкретной вершины также увеличена, т.к. все соседи вершины описаны друг

за другом. В случае почти полного графа скорость обхода та же что и в матрице, потому что количество исходящих из вершины ребер почти равно количеству вершин.

Последний метод вывода это **список ребер**.



0	1
0	4
1	2
1	5
2	0
3	4
4	1
4	2
4	5
5	0
5	3

Таблица слева является списком ребер данного графа. Преимущества такой записи является быстрым поиском по ребрам, и относительная экономия памяти по сравнению с листом и матрицей смежности.

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

Построение графа была одной из самых сложных задач данной работы. Теория графов содержит в себе много вариаций, и целью являлось поддерживать, хотя бы, самые часто используемые: ориентированный и неориентированный графы, деревья, ориентированный графы без циклов. Изначально в прошлой 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 были следующими:

- В случае дерева, можно было его строить итеративно, добавляя каждый шаг по одному ребру, если это ребро не образует цикл. Сложность состояла в том, чтобы определить, как брать ребра на каждом шагу и когда

новое ребро образует цикл, а когда нет. Наивным решением было просто на каждом шаге брать случайное ребро и искать цикл, но это было бы слишком неэффективно, скорость построения дерева была бы близка к $O(m^2)$, где m – количество рёбер. Для более продвинутого алгоритма быстрого определения образования цикла, нужно 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 и Union. Операция Find находит представителя множества элемента. Скорость работы данной операции амортизированое O(1). Операция Union объединяет два множества в одно, сложность так же равна O(1). Когда нужно 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. Данный алгоритм позволяет за O(1) определить возможно ли добавить конкретное ребро в граф. Но если выбирать рёбра случайным образом, то возможна ситуация, где нужное ребро не будет найдено в течении очень долгого времени. Для достижения скорости O(m) построения дерева нужно найти метод для неслучайного выбора рёбер, который работает амотризированно O(1) для каждого добавленного в дерево ребра.

- В случае ориентированного графа без циклов вершины были пронумерованы произвольным образом, и было разрешено проводить ребра только 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.

Эти алгоритмы работали не настолько быстро насколько хотелось. Для решения проблемы скорости построения графов был придуман совершенно другой подход.

Для начала были убраны ставшие ненужными классы DirectedGraph и NonDirectedGraph и остался только Graph, в котором добавили возможность установить ориентированность ребер с помощью параметра. Основной идеей нового подхода являлось то, что наш граф всегда связный и изначально будет сгенерировано дерево (или ориентированный граф без циклов) и только потом генерироваться дальше, если нужно. Таким образом, если передать графу n

вершин и n – 1 ребро и указать, что мы хотим неориентированный граф, то гарантированно должно было сгенерироваться дерево.

В начале алгоритма фиксируем произвольную перестановку, длина которой равна количеству вершин. По этой перестановке фиксируем произвольное дерево. При фиксации дерева никакие ребра в сам граф не добавляются.

При генерации также 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 случайный выбор вершин не является проблемой, потому что в худшем случае удалится половина рёбер, и, следовательно, вероятность попасть по ребру, которое мы взять не можем, стремится к $\frac{1}{2}$.

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);
```

Аргументы:

1. Количество вершин, может быть диапазоном.
2. Количество ребер. Так же может быть диапазоном. 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).

Как было упомянуто, сгенерированные графы всегда связные. А что если нам надо сгенерировать не связные? В этом нам поможет GraphMerger.

GraphMerger

GraphMerger – класс, объединяющий графы. Используется для получения несвязанных графов. В конструкторе получает массив или переменное количество графов, которые впоследствии объединяются. Объединение происходит следующим методом: получаются все графы в виде матриц, листов смежности или листов ребер и узнаются их размеры. Далее размеры суммируются, создаётся массив суммарного размера и в него записываются числа {1, 2, 3, …} в порядке возрастания. Массив случайным образом перемешивается. При этом каждая вершина каждого графа получает свой уникальный номер. Объединение графов реализовано посредством содержания листа смежности для всех графов одновременно. И при выводе, соответственно, выводится новый номер вершины. Перемешивание способствует разбросу номеров вершин одного связанного графа.

TestCreator

TestCreator – класс для записи тестов в файлы. Конструктор выглядит следующим образом:

```
TestCreator(  Test* _test,
              int _count,
              std::string _path,
              int _number_of_threads = 1,
              std::string _file_name_prefix = std::string(),
              std::string _extension = ".txt",
              int _start_from = 0)
```

Первый параметр – тест, который нужно генерировать. Далее идут количество тестов для генерации, путь директории на локальном диске, в которую эти тесты должны записываться и какое количество потоков должно быть использовано. Оставшиеся параметры описывают формат названия файлов: префикс файла, расширение, и с какого номера начинать.

Перед тем как перейти к разбору задач, нужно рассмотреть ещё две функции, которые есть во всех классах.

AddDeleteResponsibility

Функция AddDeleteResponsibility позволяет контролировать владение ресурсом. Возможны случаи, где в функции создаются объекты, которые нужны для конструирования другого объекта, над которыми в дальнейшем контроль теряется и их невозможно удалить. Для этого, с помощью функции AddDeleteResponsibility возможно предоставить объекту право владения над нужными ему переменными, и удалены они будут только при удалении самого объекта. 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;
}
```

Данная функция 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 на их владение той переменной, над которой мы всё ещё имеем контроль, в этом случае number.

SetPostprocessFunction

Бывают ситуации, где инструментов библиотеки не бывает достаточно. Для того, чтобы обработать такие ситуации был минималистично использован шаблон проектирования Стратегия. В каждый класс можно передать функцию, которая работает с выходными данными и меняет его исходя их наших требований. Так, например, вышеупомянутую проблему регулярных выражених (которая не позволяют получить размер сгенерированной

информации) можно решить, обработав выходные данные, и получить размеры теста. 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();
```

Генерируется строка из символов «a» и «b», и функция постобработки меняет «a» на «b» и «b» на «c».

На этом описание функционала окончено. Ниже представлены коды генерации тестов для задач сформулированных 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(1, 100000);
PrimitiveTest<int>* m = CreateElement(n, 100000);

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

```
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::LIST_OF_EDGES)->Generate()->Print();
```

Задача 4

```
RegEx* regex = new RegEx("[a-z]+(\\.[a-z]+)*@[a-z]+(\\.[a-z]+)*", 10000);
regex->Generate()->Print();
```

Решение задач и генерация ответов

Генерирация тестов лишь половина работы, нужно по готовым решениям так же генерировать ответы для тестов. После того, как правильные решения задач будут написаны, нужно прогнать тесты по решениям и сгенерировать для каждой задачи свои ответы. Для этого на языке shell написан скрипт, который делает всю эту работу.

Организация олимпиад и связь с ejudge

В 2017 году была проведена олимпиада на автоматической проверочной системе ejudge и в дальшейшем такого рода олимпиады так же будут организовываться. Для того, чтобы организовать соревнование, достаточно, чтобы были придуманы задачи и были написаны для них правильные коды, после этого с помощью данной библиотеки сгенерировать тесты, с помощью shell скрипта создать правильные ответы на тесты и всё готово, чтобы начать соревнование. Процесс подготовки к соревнованию может длиться от 2 до 7 дней, смотря насколько сложными будут задачи и тесты к ним. Но всегда бывают тесты, которые нужно писать руками и от этого никуда не деться, потому что каждая задача имеет свои худшие случаи и обработать по какому-то шаблону все не получится.

Проверка домашних заданий

Так же как и для олимпиады, данную библиотеку можно использовать для проверки домашних заданий с абсолютно такой же стратегией.

Публикация, распространение и использование библиотеки

Библиотека опубликована в открытом доступе в онлайн ресурсе github, и все имеют возможность пользоваться ею. У библиотеки есть подробная документация с примерами, которая так же находится в свободном доступе.

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. Больших нововведений пока не предусматривается.

Заключение

На этом заканчивается описание данной библиотеки. В результате этой работы было изучено много нового материала – шаблоны проектирования, многопоточность, архитектуры библиотек и многое другое.

Источники

- wikipedia.org
- stackoverflow.com
- Introduction to Algorithms, T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein.
- Design Patterns: Elements of Reusable Object-Oriented, E. Gamma, R. Helm, R. Johnson, J. Vlissides, G. Booch.