Оглавление

Тестирование **2**

2.1 Тестирование и отладка 2

2.1.1 Введение в юнит-тестирование 2

2.1.2 Декомпозиция решения в задаче «Синонимы» 3

2.1.3 Простейший способ создания юнит-тестов на C++ 5

2.1.4 Отладка решения задачи «Синонимы» с помощью юнит-тестов 7

2.1.5 Анализ недостатков фреймворка юнит-тестов 10

2.1.6 Улучшаем `assert` 11

2.1.7 Внедряем шаблон `AssertEqual` во все юнит-тесты 13

2.1.8 Изолируем запуск отдельных тестов 15

2.1.9 Избавляемся от смешания вывода тестов и основной программы 17

2.1.10 Обеспечиваем регулярный запуск юнит-тестов 18

2.1.11 Собственный фреймворк юнит-тестов. Итоги 19

2.1.12 Общие рекомендации по декомпозиции программы и написанию юнит-тестов 20

Тестирование

2.1. Тестирование и отладка

2.1.1. Введение в юнит-тестирование

Вспомним, что говорилось в курсе «C++: белый пояс». Если решение не принимается тестирующей системой, то нужно:

1. Внимательно перечитать условия задачи;
2. Убедиться, что программа корректно работает на примерах;
3. Составить план тестирования (проанализируем классы входных данных);
4. Тестировать программу, пока она не пройдет все тесты;
5. Если идеи тестов кончились, но программа не принимается, то выполняем декомпозицию программы на отдельные блоки и покрываем каждый из них юнит-тестами.

Как юнит-тесты помогают в отладке:

1. Позволяют протестировать каждый компонент изолированно;
2. Их проще придумывать;
3. Упавшие юнит-тесты указывают, в каком блоке программы ошибка.

2.1.2. Декомпозиция решения в задаче «Синонимы»

На примере задачи «Синонимы» из курса «C++: белый пояс» покажем применение юнит-тестов.

Условие задачи:

Два слова называются синонимами, если они имеют похожие значения. Надо реализовать словарь синонимов, обрабатывающий три вида запросов:

- `ADD word1 word2` – добавить в словарь пару синонимов (`word1`, `word2`);
- `COUNT word` – выводит текущее количество синонимов для слова `word`;
- `CHECK word1 word2` – проверяет, являются ли слова синонимами.

Ввод:	Вывод:
ADD program code	
ADD code cipher	
COUNT cipher	1
CHECK code program	YES
CHECK program cipher	NO

Посмотрим на решение, которое у нас уже есть и которое надо протестировать:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;

int main() {
  int q;  // считываем количество запросов
  cin >> q;
  // храним словарь синонимов (для строки хранит множество всех её синонимов)
  map<string, set<string>> synonyms;
  for (int i = 0; i < q; ++i) {  // обрабатываем запросы
    string operation_code;
    cin >> operation_code;

    if (operation_code == "ADD") {  // считываем две строки и добавляем в словарь
```

```
      string first_word, second_word;
      cin >> first_word >> second_word;
      synonyms[second_word].insert(first_word);
      synonyms[first_word].insert(second_word);
    } else if (operation_code == "COUNT") {  // считываем одну строку и выводим
    // размер
      string word;
      cin >> word;
      cout << synonyms[word].size() << endl;
    } else if (operation_code == "CHECK") {  // считываем два слова и проверяем
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (synonyms[first_word].count(second_word) == 1) {
        cout << "YES" << endl;
      } else {
        cout << "NO" << endl;
      }
    }
  }
  return 0;
}
```

Отправляем в тестирующую систему и видим, что решение не принялось. Он говорит нам, что неправильный ответ на третьем тесте, и больше информации нет. Решение надо тестировать на различных входных данных. Но вроде всё работает и стоит переходить к юнит-тестированию. А для него надо сначала выполнить декомпозицию задачи «Синонимы». Давайте ввод и вывод оставим в `main`, а обработку запроса вынесем в отдельные функции:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;
void AddSynonyms(map<string, set<string>>& synonyms,  // функция добавления
                 const string& first_word, const string& second_word) {
  synonyms[second_word].insert(first_word);
  synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(map<string, set<string>>& synonyms,  // количество синонимов
                       const string& first_word) {
  return synonyms[first_word].size();
```

```
}
bool AreSynonyms(map<string, set<string>>& synonyms,  // проверка
                 const string& first_word, const string& second_word) {
  return synonyms[first_word].count(‘second_word) == 1;
}
int main() {
  int q;
  cin >> q;
  map<string, set <string >> synonyms;
  for (int i = 0; i < q; ++i) {
    string operation_code;
    cin >> operation_code;
    if (operation_code = "ADD") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      AddSynonyms(synonyms, first_word, second_word)
  } else if (operation_code = "COUNT") {
      string word;
      cin >> word;
      cout << GetSynonymCount(synonyms, word) << endl;
  } else if (operation_code == "CHECK") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
      } else {
        cout << "NO" << endl;
      }
    }
  }
  return 0;
}
```

2.1.3. Простейший способ создания юнит-тестов на C++

Посмотрим, как писать юнит-тесты и как они должны себя вести на примере функции `Sum`, которая находит сумму двух чисел.

```
#include <iostream>
```

```cpp
#include <cassert>  // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
  return x + y;
}
void TestSum() {  // собираем набор тестов для функции Sum
  assert(Sum(2, 3) = 5);  // мы ожидаем, что 2+3=5
  assert(Sum(−2, −3) −5);  // проверка отрицательных чисел
  assert(Sum(−2, 9) −2);  // проверка прибавления 0
  assert(Sum(−2, 2) 9);  // проверка, когда сумма = 0
  cout << "TestSum OK" << endl;
}
int main() {
  TestSum0;
  return 0;
}
// TestSum OK
```

Тесты отработали и не нашли ошибок. Теперь посмотрим, что должно быть при наличии ошибок:

```cpp
#include <iostream>
#include <cassert>  // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
  return x + y − 1;  // сделали заведомо неправильно
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Нам написали, в каком файле, в какой строке какой `Assert` не сработал. Это облегчает поиск ошибок. Мы можем добиться другой ошибки, например:

```cpp
#include <iostream>
#include <cassert>  // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
  if(x < 0) {
    x −= 1
  }
  return x + y;
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Видим, что первый тест прошёл, а на втором уже ошибка. Таким образом, мы можем проверять каждую функцию по отдельности на ожидаемых значениях.

2.1.4. Отладка решения задачи «Синонимы» с помощью юнит-тестов

Для задачи «Синонимы» покроем каждую функцию юнит-тестами и сократим код заменой `map<string;set<string>>` на что-то более короткое:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <cassert>
using namespace std;
using Synonyms = map<string, set<string>>;
// сократили запись типа и везде изменили на Synonyms
void AddSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
  synonyms[second_word].insert(first_word);
  synonyms[first_word].insert(first_word);  // тут должен не сработать AddSynonyms
}
size_t GetSynonymCount(Synonyms& synonyms, const string& word) {
  return synonyms[word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
  return synonyms[first_word].count(second_word) == 1;
}

void TestAddSynonyms() {  // тестируем AddSynonyms
  {
    Synonyms empty;  // тест 1
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
      {"a", {"b"}},  // ожидаем, что при добавлении синонимов появятся две записи в
      // словаре
      {"b", {"a"}}
    };
    assert(empty == expected);
```

```
  }
  {  // заметим, что мы формируем корректный словарь и ожидаем, что он останется корректным
    Synonyms synonyms = {  // если вдруг корректность нарушится, то assert скажет, где
      {"a", {"b"}},  // тест 2
      {"b", {"a", "c"}},
      {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
      {"a", {"b", "c"}},
      {"b", {"a", "c"}},
      {"c", {"a", "b"}}
    };
    assert(synonyms == expected);
  }
  cout << "TestAddSynonyms OK" << endl;
}

void TestCount() {  // тестируем Count
  {
    Synonyms empty;
    assert(GetSynonymCount(empty, "a") == 0);
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    assert(GetSynonymCount(synonyms, "a") == 2);
    assert(GetSynonymCount(synonyms, "b") == 1);
    assert(GetSynonymCount(synonyms, "z") == 0);
  }
  cout << "TestCount OK" << endl;
}

void TestAreSynonyms() {  // тестируем AreSynonyms
  {
    Synonyms empty;  // пустой словарь для любых двух слов вернёт false
    assert(!AreSynonyms(empty, "a", "b"));
    assert(!AreSynonyms(empty, "b", "a"));
```

```
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    assert(AreSynonyms(synonyms, "a", "b"));
    assert(AreSynonyms(synonyms, "b", "a"));
    assert(AreSynonyms(synonyms, "a", "c"));
    assert(AreSynonyms(synonyms, "c", "a"));
    assert(!AreSynonyms(synonyms, "b", "c"));  // false
    assert(!AreSynonyms(synonyms, "c", "b"));  // false
  }
  cout << "TestAreSynonyms OK" << endl;
}
void TestAll() {  // функция, вызывающая все тесты
  TestCount();
  TestAreSynonyms();
  TestAddSynonyms();
}
int main() {
  TestAll();
  return 0;
}
// TestCount OK
// TestAreSynonyms OK
// main.cpp:26: void TestAddSynonyms(): Assertion `empty == expected' failed.
```

Видим, что `Count` и `AreSynonyms` работают нормально, а вот в `AddSynonyms` у нас ошибка. Смотрим, что не так. Идём в `AddSynonyms` и видим, что:

```
synonyms[first_word].insert(first_word);  // мы должны к 1 слову добавить 2, а не 1
```

Теперь после исправления все наши тесты отработали успешно. Снова пробуем отправить в тестирующую систему наше решение, закомментировав вызов `TestAll();` в `main`. Тестирование завершилось и решение принято тестирующей системой. Таким образом мы на простом примере продемонстрировали эффективность декомпозиции программы и юнит-тестов.

2.1.5. Анализ недостатков фреймворка юнит-тестов

По ходу разработки юнит-тестов во время решения задачи «Синонимы» мы смогли написать небольшой юнит-тест фреймворк. Посмотрим, что за фреймворк получился. Во-первых, он основан на функции `assert`. Его главный плюс – мы узнаём, какая именно проверка сработала неправильно. На предыдущей задаче мы видели:

```
// main.cpp:26: void TestAddSynonyms(): Assertion `empty == expected' failed.
```

Основные недостатки:

1. При проверке равенства в консоль не выводятся значения сравниваемых переменных. И мы не знаем, чем была переменная `empty`;
2. После невыполненого `assert` код падает. Если в `TestAll` поставить `TestAddSynonyms` на первое место, то остальные два теста даже не начнутся;
3. Кроме того, у нас пока что результаты тестов выводят `OK` в стандартный вывод и смешиваются с тем, что должен выводить код.

В C++ уже существует много фреймворков для работы с тестами, в которых этих недостатков нет.

C++ Unit Testing Frameworks:

1. Google Test
2. CxxTest
3. Boost Test Library

Далее мы свой Unit Testing Framework улучшим для того, чтобы показать, что текущих знаний C++ хватает для таких вещей. И вы будете понимать как он работает, и сможете его менять под свои нужды.

2.1.6. Улучшаем assert

Избавимся от первого недостатка `assert`: когда он срабатывает, мы не видим, чему равен каждый из операндов. Т. е. мы хотим видеть для кода такой вывод:

```
int x = Add(2, 3);
assert(x == 4);
// Assertion failed: 5 != 4
```

И работало оно для любых типов данных:

```
vector<int> sorted = Sort({1, 4, 3});
assert(sorted == vector<int> {1, 3, 4}));
// Assertion failed: [1, 4, 3] != [1, 3, 4]
```

Такие универсальные выводы помогут догадаться о возможной ошибке. Т. е. нам нужна функция сравнения двух переменных какого-то произвольного типа. Напишем шаблон `AssertEqual` перед `TestAddSynonyms()`.

```
#include <exception>  // подключим исключения
#include <sstream>  // подключили строковые потоки
...
template <class T, class U>
void AssertEqual (class T& t, const U& u) {
// значения двух разных типов для удобства
  if (t != u) {  // если значения не равны, то мы даём знать, что этот assert не сработал
    ostringstream os;
    os << "Assertion failed: " << t << "!=" << u;
    throw runtime_error(os);  // бросим исключение с сообщением со значениями t и u
  }
}
```

Встроим это в `TestCount()`. Заменим `assert` на наш `AssertEqual` внутри `TestCount()`.

```
void TestCount() {
  {
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty, "a"), 0);
    AssertEqual(GetSynonymCount(empty, "b"), 0);
  }
  {
    Synonyms synonyms = {
```

```
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2);
    AssertEqual(GetSynonymCount(synonyms, "b"), 1);
    AssertEqual(GetSynonymCount(synonyms, "z"), 0);
  }
}
// Warning ... comprison between signed and unsigned ...
```

Код скомпилировался, но мы получили Warning из-за сравнения между знаковым и беззнаковым типами в `AssertEqual`. Это происходит потому, что все константы (2, 1 и 0 в нашем случае) имеют тип `int` (как уже было сказано в неделе 1), который мы сравниваем с типом `size_t`. Исправляем это, дописав к ним `u` справа: `1` → `1u` и т. д.

Теперь, сделав нарочную ошибку где-нибудь в `GetSynonymCount`, мы получим предупреждение: `Assertion failed: 1 != 0`. Но пока мы не видим, какой именно `Assert` сработал. Исправим это, передавая в `Assert` строчку `hint` и также добавим каждому `Assert`'у строку идентификации, по которой мы сможем однозначно понять, какой именно `Assert` выдал ошибку:

```
void AssertEqual (class T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
    os << "Assertion failed: " << t << "!=" << u << "Hint: " << hint;
    throw runtime_error(os);
  }
}
...
void TestCount() {
  {
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty, "a"),
        0u, "Synomym count for empty dict a");
    AssertEqual(GetSynonymCount(empty, "b"),
        0u, "Synomym count for empty dict b");
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
```

```
      {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms,"a"), 2u, "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms,"b"), 1u, "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms,"z"), 0u, "Nonempty dict, count z");
  }
}
// Asserting failed: 1!= 0 Hint: Synomym count for empty dict
```

2.1.7. Внедряем шаблон AssertEqual во все юнит-тесты

Добавим Assert в AreSynonyms. Только AssertEqual нам не подходит, потому что в данной функции у нас только два константных значения: true и false. Вместо этого напишем аналог классического assert, который назовём Assert (C++ чувствителен к регистру). И если мы испортим функцию AreSynonyms, то получим соответствующую ошибку с подсказкой.

```
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}
...  // модернизируем наш TestAreSynonyms
void TestAreSynonyms() {
  {
    Synonyms empty;
    Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
    Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    Assert(AreSynonyms(synonyms, "a", "b"),  "AreSynonyms nonempty a b");
    Assert(AreSynonyms(synonyms, "b", "a"),  "AreSynonyms nonempty b a");
    Assert(AreSynonyms(synonyms, "a", "c"),  "AreSynonyms nonempty a c");
    Assert(AreSynonyms(synonyms, "c", "a"),  "AreSynonyms nonempty c a");
    Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
    Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
```

```
  }
}
// Asserting failed: 0!= 1 Hint: AreSynonyms empty a b
```

Получили нужную ошибку, и по ней мы можем увидеть, где что-то не так. Осталась только функция `AddSynonyms`, и ловим ошибку:

```
void TestAddSynonyms() {
  {
    Synonyms empty;
    AddSynomyns(empty, "a", "b");
    const Synonyms expected = {
      {"a", {"b"}},
      {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Add to empty");
  }
  {
    Synonyms synonyms = {
      {"a", {"b"}},
      {"b", {"a", "c"}},
      {"c", {"b"}}
    };
    AddSynomyns(synonyms, "a", "c");
    const Synonyms expected = {
      {"a", {"b", "c"}},
      {"b", {"a", "c"}},
      {"c", {"b", "a"}}
    };
    AssertEqual(synonyms, expected, "Nonempty");
  }
}
// no match for 'operator <<' (operand types are std::ostream<char> and std::map ...)
```

Ошибка произошла с `empty` и `synonyms`, которые являются `map<string, set<string>`. Мы пытаемся их вывести в стандартный поток вывода (см. неделя 1). Пишем перегрузку оператора вывода для `map` и для `set`, предварительно исправив ошибку в `AreSynonyms`, допустим ошибку в `AddSynonyms` и поймаем её:

```
template <class T>  // учимся выводить в поток set
ostream& operator << (ostream& os, const set<T>& s) {
```

```
  os << "{";
  bool first = true;
  for (const auto& x : s) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << x;
  }
  return os << "}";
}
template <class K, class V>  // учимся выводить в поток map
ostream& operator << (ostream& os, const map<K, V>& m) {
  os << "{";
  bool first = true;  // грамотная расстановка запятых
  for (const auto& kv : m) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << kv.first << ": " << kv.second;
  }
  return os << "}";
}
// Asserting failed: {a: {a}, b: {a} } != {a: {b}, b: {a}. Hint: Add to empty
```

Таким образом, мы внедрили шаблон `AssertEqual`, который позволяет найти ошибку, узнать, с чем она возникла и найти конкретное место в коде благодаря подсказке.

2.1.8. Изолируем запуск отдельных тестов

Теперь исправим следующий недостаток `Assert`: если он срабатывает, то код падает и другие тесты не выполняются. Аварийное завершение программы у нас возникало из-за вылета исключения в `AssertEqual`. Теперь будем ловить эти исключения в `main`:

```
int main() {
  try {
    TestAreSynonyms();  // ловим исключение
```

```
  } catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAreSynonyms" << " fail: " << e.what() << endl;
  }  // если мы словили исключение, то работа всё равно продолжится
  try {
    TestCount();
  } catch (runtime_error& e) {
    ++fail_count;
    cout <<" TestCount" << " fail: " << e.what() << endl;
  }
  try {
    TestAddSynonyms();
  } catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAddSynonyms" << " fail: " << e.what() << endl;
  }
}
```

Но это неудобно, код дублируется. Нам бы хотелось, чтобы этот `try/catch` и вывод были написаны в одном месте, а мы туда могли бы передавать различные тестовые функции, и они бы там выполнялись, и исключения бы от них ловились, всё бы работало. В C++ это можно сделать, ведь функции имеют тип и их можно передавать в другие функции как аргумент. Создадим шаблон функции `RunTest`, который будет запускать тесты и ловить исключения.

```
...
template <class TestFunc>

void RunTest(TestFunc func, const string& test_name) {  // передаём тест и его имя
  try {
    func();
  } catch (runtime_error& e) {
    cerr << test_name << " fail: " << e.what() << endl;  // ловим исключение
  }
}

int main() {  // когда передаём функцию как переметр, передаём только её имя без скобочек
  RunTest(TestAreSynonyms, "TestAreSynonyms");
  RunTest(TestCount, "TestCount");
  RunTest(TestAddSynonyms, "TestAddSynonyms");
}
```

Заметим, что все тесты работают в любом порядке. Таким образом, мы с вами применили шаблон функций, для того чтобы передавать в качестве параметров функции другие функции, и смогли за счет этого написать универсальный такой шаблон, который для любого юнит-теста ловит исключения и позволяет нам все юнит-тесты, которые мы написали, выполнять при каждом запуске нашей программы.

2.1.9. Избавляемся от смешания вывода тестов и основной программы

Добавим в `RunTest` ещё одну удобную вещь: будем выводить `OK` снаружи каждого юнит-теста.

```
void RunTest(TestFunc func, const string& test_name) {
  try {
    func();  // заменим cout на cerr - стандартный поток ошибок
    cerr << test_name << " OK" << endl;  // выводит OK, если всё работает
  } catch (runtime_error& e) {
    cerr << test_name << " fail: " << e.what() << endl;  // fail, если ошибка
  }
}
```

Всё это время сам алгоритм решения задачи «Синонимы» (который всё это время был закомментирован), всё ещё хорошо работает. Вот он сам:

```
int main() {  // когда передаём функцию как переметр, передаём только её имя без скобочек
  RunTest(TestAreSynonyms,"TestAreSynonyms");
  RunTest(TestCount,"TestCount");
  RunTest(TestAddSynonyms,"TestAddSynonyms");
  int q;
  cin >> q;
  Synonyms synonyms;
  for (int i = 0; i < q; ++i) {
    string operation_code;
    cin >> operation_code;
    if (operation_code == "ADD") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      AddSynomyns(synonyms, first_word, second_word);
    } else if (operation_code == "COUNT") {
      string word;
      cin >> word;
```

```
      cout << GetSynonymCount(synonyms, word) << endl;
    } else if (operation_code == "CHECK") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
      } else {
        cout << "NO" << endl;
      }
    }
  }
}
```

Вся проблема в том, что и юнит-тесты, и сама программа выводят в стандартный вывод. Пусть юнит-тесты выводят в `stderr` (стандартный поток ошибок). По окраске вывода в Eclipse можно отличать **стандартный вывод**, стандартный ввод и стандартный поток ошибок. Вывод: TestAreSynonyms OK, 5 TestAddSynonyms OK, TestCount OK, 1, COUNT a, **0**. Теперь не нужно окружать комментарием эти части кода перед отправкой, ведь они всё равно не повлияют на работу самой программы.

2.1.10. Обеспечиваем регулярный запуск юнит-тестов

Мы хотим, чтобы юнит-тесты были автоматическими, и их код находился где-то отдельно. Кроме того, стоит считать ошибки, чтобы если существует хоть одна, то программа не ждала получения данных от пользователя. Таким образом, мы хотим:

1. Запускаем тесты при старте программы. Если хоть один тест упал, программа завершается;
2. Если все тесты прошли, то должно работать решение самой задачи.

Для этого обернём наш шаблон `RunTest` в класс `TestRunner`:

```
class TestRunner {  // класс тестирования
  public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {
      try {  // RunTest стал шаблонным методом класса
```

```
        func();
        cerr << test_name << " OK" << endl;
      } catch (runtime_error& e) {
        ++fail_count;  // увеличиваем счётчик упавших тестов
        cerr << test_name << " fail: " << e.what() << endl;
      }
    }
    ~TestRunner() {  // деструктор класса TestRunner, в котором анализируем fail_count
      if (fail_count > 0) { //это как раз тот момент, когда
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1);  // завершение программы с кодом возврата 1
      }
    }
  private:
    int fail_count = 0;  // счётчик числа упавших тестов
};
void TestAll() {  // переместили все тесты в одну функцию
  TestRunner tr;
  tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
  tr.RunTest(TestCount, "TestCount");
  tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}
int main() {
  TestAll();  // т.к. мы деструктор класса объявили в самом классе,
...}  // выполняется он в конце TestAll
```

Теперь, если мы словили хоть одну ошибку, то программа перестанет выполняться с кодом возврата 1. Если же ничего плохого не произошло, то мы можем ввести число команд и сами команды.

2.1.11. Собственный фреймворк юнит-тестов. Итоги

Подведём итоги написания собственного фреймворка. Его основные свойства:

- Если срабатывает `assert`, в консоль выводятся его аргументы (работает для контейнеров);
- Вывод тестов не смешивается с выводом основной программы;

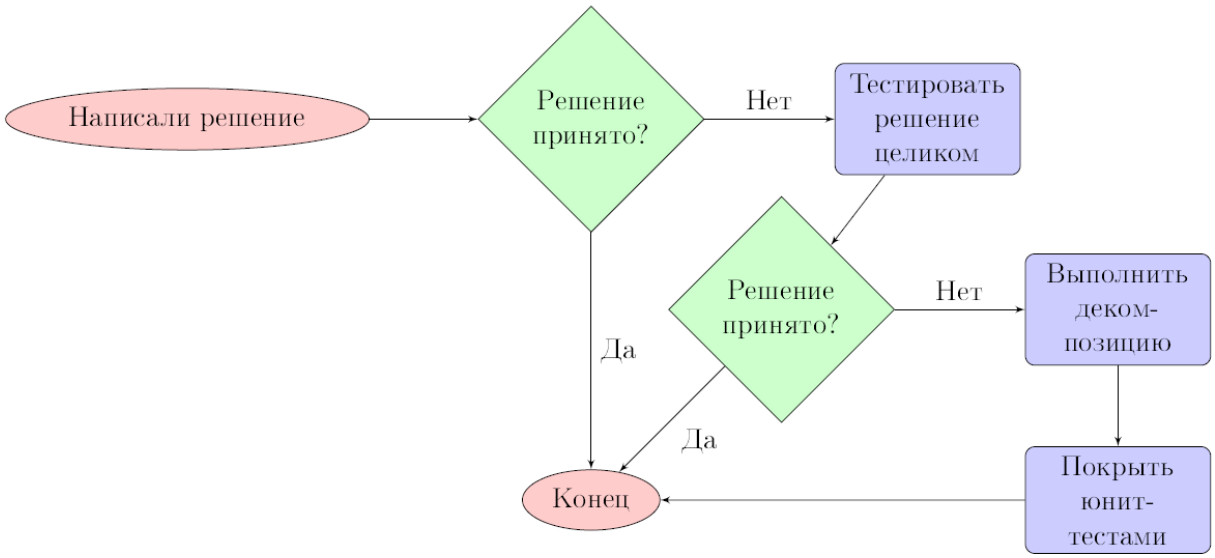
- При каждом запуске программы выполняются все юнит-тесты;
- Если хотя бы один тест упал, программа завершится с ненулевым кодом возврата.

Для того, чтобы пользоваться фреймворком, надо написать:

```
void TestSomething() {  // функция, что-то тестирующая
  AssertEqual(..., ...);
  // выполняем какие-то проверки с помощью AssertEqual
}
void TestAll() {
  TestRunner tr;
  tr.RunTest(TestSomething, "TestSomething")
  // вызываем методом RunTest
}
int main() {
  TestAll();  // должна быть до самой программы
}  // код фреймворка выложен рядом с видео
```

2.1.12. Общие рекомендации по декомпозиции программы и написанию юнит-тестов

Общий алгоритм решения задач с помощью декомпозиции и юнит-тестов:



Но кроме этой схемы, стоит выполнять декомпозицию задачи по ходу написания самого кода. Декомпозицию лучше делать сразу:

- Отдельные блоки проще реализовать и вероятность допустить ошибку ниже;
- Их проще тестировать, соответственно, выше вероятность найти ошибку или убедиться в её отсутствии;
- В больших проектах декомпозиция упрощает понимание и переиспользование кода;
- Уже реализованные функции можно брать и ипользовать в другом месте;
- Сама декомпозиция иногда защищает от ошибок.

Вспомним задачу «Уравнение» из курса «C++: белый пояс». Нужно было найти все различные действительные корни уравнения $Ax^2 + By^2 + C = 0$. Гарантируется, что $A^2 + B^2 + C^2 > 0$. Монолитное решение могло выглядеть так:

```
#include <cmath>
#include <iostream>
using namespace std;
int main() {
```

```
  double a, b, c, D, x1, x2;
  cin >> a >> b >> c;
  D = b * b - 4 * a * c;
  if ((a == 0 && c == 0) || (b == 0 && c == 0)) {
    cout << 0;
  } else if (a == 0) {
    cout << -(c / b);  // тут ошибка. Если b == 0, мы всё равно разделим на 0
  } else if (b == 0) {
    cout << " ";
  } else if (c == 0) {
    cout << 0 << " " << -(b / a);
  } else if (D < 0) {
    cout << " ";
  } else if (D == 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    cout << x1;
  } else if (D > 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    x2 = ((-1 * b) - sqrt(D)) / (2 * a);
    cout << x1 << " " << x2;
  }
  return 0;
}
```

Быстро просмотрев этот код, сложно понять, работает он или нет. А в нём есть ошибка, которую из-за монолитности кода сложно заметить сразу. Теперь рассмотрим декомпозированное решение той же задачи:

```
#include <cmath>
#include <iostream>
using namespace std;

void SolveQuadradicEquation(double a, double b, double c) {
  // ... тут решение гарантированного квадратного уравнения
}

void SolveLinearEquation(double b, double c) {
  // b * x + c = 0
  if (b != 0) {  // тут не забыли проверить деление на 0
    cout << -c / b;
  }
```

```
}

int main() {
  double a, b, c;
  cin >> a >> b >> c;
  if (a != 0) {  // точно знаем, что уравнение квадратное
    SolveQuadradicEquation(a, b, c);
  } else {  // просто решаем линейное
    SolveLinearEquation(b, c);
  }
  return 0;
}
```

Юнит-тесты тоже лучше делать сразу. Причины:

- Разрабатывая тесты, вы сразу продумываете все варианты использования вашего кода и все крайние случаи входных данных;
- Тесты позволяют вам сразу проконтролировать корректность вашей реализации (особенно актуально для больших проектов);
- В больших проектах обширный набор тестов позволяет убедиться, что вы ничего не сломали во время дополнения кода.