*Математические структуры и моделирование*
*2013. № 1(27). С. 82–88*

УДК 004.053

# АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ СЛОЖНОСТИ АЛГОРИТМОВ С ПОМОЩЬЮ MOCK-ОБЪЕКТОВ

**Е.А. Тюменцев**

В статье описывается способ тестирования алгоритмической сложности алгоритмов на основе Mock-объектов, позволяющий автоматизировать проверку, исследовать и классифицировать алгоритмы с точки зрения сложности, даже если их реализация неизвестна.

Практически для любой задачи существуют различные способы решения, которые отличаются друг от друга алгоритмической сложностью. Например, в классе двунаправленный список list<T> элементов типа T есть метод int Count(), который возвращает количество элементов в этом списке. Существуют две очевидные возможности (хотя вариантов, на самом деле, больше):

- Класс list<T> хранит количество элементов списка в специально отведённом для этого поле. Тогда метод int Count() будет иметь сложность $O(C)$, где $C$ — константа.

- Метод int Count() перебирает все элементы списка для подсчёта количества элементов. Тогда это метод будет иметь сложность $O(n)$, где $n$ — длина списка.

Чтобы выбирать правильные способы решения, программисту необходимо знать их алгоритмическую сложность. Наиболее распространённым методом оценки сложности является её явное вычисление по исходному коду. К сожалению, исходный код доступен не всегда, а такой метод требует определённой квалификации от программиста и ручной работы. В настоящей статье будет описан подход к написанию автоматизированных тестов на алгоритмическую сложность программного кода на основе Mock-объектов.

**Определение 1** (Mock-объект)**.** Объект-имитация реального объекта программного окружения, реализующая только некоторые аспекты реального объекта с целью тестирования определённого поведения программного окружения.

---


LLC "Hello World! Technologies"
E-mail: etyumentcev@hwdtech.ru

Mock-объекты не являются библиотекой или конкретной технологией тестирования, они представляют собой сложившуюся методологию тестирования объектно-ориентированного программного обеспечения, основанную на принципе обращения зависимостей (The Dependency Inversion Principle) [1]. Подробнее об истории выработки практики применения Mock-объектов можно прочитать в [2].

Суть предлагаемого способа тестирования можно выразить следующей цепочкой утверждений:

- Алгоритмическая сложность прямо зависит от количества и состава операций, выполненных над входным набором данных.
- Операция в объектно-ориентированном программировании представляет собой поведение некоторого объекта.
- Поведение объекта описывает какой-либо интерфейс.
- Mock-объект подменяет собой реальный объект с целью проведения тестирования.
- Подмена реального объекта Mock-объектом происходит за счёт того, что оба этих объекта реализуют один и тот же интерфейс, а программа в целом должна удовлетворять принципу подстановки Лисков [3].
- Следовательно, чтобы определить алгоритмическую сложность какого-либо алгоритма, необходимо предоставить этому алгоритму вместо реальных объектов Mock-объекты. Каждый метод Mock-объекта будет считать каждый собственный вызов. После того, как тестируемый алгоритм завершит свою работу, будет известно, сколько каждый метод был вызван.
- Остаётся только проверить значения соответствующих счётчиков в зависимости от количества входных данных.

Проиллюстрируем вычисление алгоритмической сложности на примерах. Для примеров был использован C++ и компилятор MSVS 2010. Предположим, что у нас есть бинарное дерево с операцией вставки Insert.

```
template<typename T> class Tree
{
  struct TreeNode
  {
    TreeNode(T const& value)
    {
      this->value = value;
      left = 0;
      right = 0;
    }
    TreeNode* left;
```

```
    TreeNode* right;
    T value;
  };

public:

  Tree() { root = 0; }
  void Insert(T const& value)
  {
    if(0 == root)
      root = new TreeNode(value);
    else
      _insert(root, value);
  }

private:

  void _insert(TreeNode *root, T const & value)
  {
    if(value == root->value)
      return;
    else
    {
      if(value < root->value)
      {
        if(0 == root->left)
          root->left = new TreeNode(value);
        else
          _insert(root->left, value);
      }
      else
      {
        if(0 == root->right)
          root->right = new TreeNode(value);
        else
          _insert(root->right, value);
      }
    }
  }

  TreeNode *root;
};
```

Мы хотим протестировать сложность вставки в данное дерево. Для этого создадим Mock-объект — класс A. Сложность будем считать в операциях чте-

ния и записи данного объекта (при необходимости можно было бы считать и каждую операцию в отдельности). К пишущим операциям отнесём конструктор копии и оператор присваивания. К читающим операциям - ==, < .

```
class A
{
public:
  A(): someValue(0){}
  A(int value): someValue(value) {}
  A(A const& other)
  {
    ++other.writeCounter;
    this->someValue = other.someValue;
  }
  A& operator=(A const& other)
  {
    ++other.writeCounter;
    this->someValue = other.someValue;
    return *this;
  }
private:
  int someValue;
public:
  static int readCounter;
  static int writeCounter;

  static void Clear()
  {
    readCounter = 0;
    writeCounter = 0;
  }

  friend bool operator==(A const& a1, A const & a2);
  friend bool operator<(A const& a1, A const & a2);
};

int A::readCounter = 0;
int A::writeCounter = 0;

bool operator==(A const& a1, A const & a2)
{
  ++a1.readCounter;
  return a1.someValue == a2.someValue;
}
```

```
bool operator<(A const& a1, A const & a2)
{
  ++a1.readCounter;
  return a1.someValue < a2.someValue;
}

bool operator==(std::pair<const A, int> & p1,
                std::pair<const A, int> & p2)
{
  return p1.first == p2.first;
}
```

Хорошо известно, что в худшем случае вставка $n$ элементов в бинарное дерево поиска имеет сложность порядка $O(n^2)$. Проверим это на примере:

```
Tree<A> tree;
for(int i = 0; i < 1024; ++i)
{
  tree.Insert(A(i));
}

cout << "Insert 1024 items to nonbalanced tree: read "
     << A::readCounter << " write "
     << A::writeCounter << endl;
```

Вывод на консоль:

```
Insert 1024 items to nonbalanced tree:
read 1047552 write 1024
```

Получаем $n * (n - 1)$ операций чтения, и n операций записи, что в сумме даёт $n^2$ общего числа операций.

В наилучшем случае вставка $n$ элементов в бинарное дерево поиска имеет сложность порядка $O(n * \log_2(n))$. Напишем специальную функцию вставки, которая нам обеспечит наилучший случай:

```
void Insert(Tree<A> & t, int l, int r)
{
  int val = (l+r)/2;
  t.Insert(A(val));

  if(l!= val)
    Insert(t, l, val);
  if(r - val > 1)
    Insert(t, val, r);
  else
    if(r-val == 1)
      t.Insert(r);
}
```

Теперь проверим количество операций:

```
Tree<A> tree;
Insert(tree, 0, 1023);
cout << "Insert 1024 items to nonbalanced tree: read "
     << A::readCounter  << " write "
     << A::writeCounter << endl;
```

Вывод на консоль:

```
Insert 1024 items to nonbalanced tree:
read 41483 write 1024
```

$\log_2(1024) = 10$, а, следовательно, исследуемый нами случай имеет сложность порядка $C * O(n * \log_2(n))$.

Очевидно, что рассмотренный код несложно преобразовать в автоматический тест на базе одной из известных библиотек для организации тестирования.

Таким способом можно тестировать и исследовать не только собственные классы, но и сторонний код. Рассмотрим классы list<T> и map<Key, Value> стандартной библиотеки STL языка C++. Заметим, что в качестве Mock-объекта можно использовать тот же самый класс A, который применялся для тестирования класса Tree.

Известно, что вставка в список имеет порядок $O(C)$.

```
list<A> list;
for(int i = 999; i >= 0; --i)
  list.insert(list.begin(), A(i));
cout << "Insert to list: read " << A::readCounter
     << " write " << A::writeCounter << endl;
```

Вывод на консоль:

```
Insert to list: read 0 write 1000.
```

В худшем случае операция поиска в линейном списке имеет порядок $O(n)$.

```
find(list.begin(), list.end(), A(999));
cout << "Find in list: read " <<  A::readCounter
     << " write " << A::writeCounter << endl;
```

Вывод на консоль:

```
Find in list: read 1000 write 0.
```

Вставка $n$ элементов в map имеет сложность порядка $O(n * \log_2(n))$.

```
map<A,int> map;
for(int i = 0; i < 1024; ++i)
  map[A(i)] = i;
cout << "Insert to map 1024 items: read " << A::readCounter
     << " write " << A::writeCounter << endl;
```

Вывод на консоль:

```
Insert to map 1024 items: read 31828 write 2048
```

По выводу на консоль видно, что при вставке в map вставляется не сам элемент, а его копия, что для больших объектов без разделения состояния между экземплярами может привести к накладным расходам, сводящим на нет преимущества от использования map.

В настоящей статье был рассмотрен подход к реализации тестирования алгоритмической сложности объектно-ориентированных систем с применением Mock-объектов, позволяющий:

- автоматизировать проверку алгоритмической сложности используемого решения;
- вычислять трудоёмкость в терминах выполняемых операций;
- как следствие предыдущего пункта, применять количественные критерии, не зависящие от вычислительного устройства, при описании требований к производительности;
- исследовать и классифицировать алгоритмы с точки зрения сложности, даже если их реализация неизвестна;
- применять разработанные тесты и на реальных данных.

## Литература

1. The Dependency Inversion Principle. URL: `http://www.objectmentor.com/resources/articles/dip.pdf` (дата обращения: 01.12.12)
2. A Brief History of Mock Objects. URL: `http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html` (дата обращения: 01.12.12)
3. The Liskov Substitution Principle. URL: `http://www.objectmentor.com/resources/articles/lsp.pdf` (дата обращения: 01.12.12)