ОСНОВЫ ТЕОРИИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Пышкин Евгений Валерьевич
к.т.н., доцент

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Блок 9. Введение в модульное тестирование

Проблема тестирования разрабатываемых классов

- Напомним важные понятия
  - ***Абстрагирование*** ***позволяет выделить существенные характеристики некоторого объекта, отличающие его от всех других видов объектов. Абстракция четко определяет концептуальные границы объекта с точки зрения наблюдателя***
  - ***Инкапсуляция*** ***– это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция также служит для того, чтобы изолировать внешнее поведение объекта от его внутреннего устройства***

Пример, рассмотренный ранее: класс Rational

- Как будут создаваться объекты класса «рациональная дробь»? Что происходит, если задаваемое пользователем значение знаменателя некорректно (равно нулю)?
- Как при реализации действий, выполняемых над объектами класса «рациональная дробь», абстрагироваться от внутреннего представления объекта?
- Поскольку Rational, по сути своей, является числовым типом, то, с точки зрения «философии» языка C++, было бы полезно обеспечить поддержку стереотипного использования операций языка (например, арифметических) так, чтобы с объектами - рациональными дробями можно было работать аналогично использованию объектов встроенных числовых типов языка.
- При реализации операций следует стремиться к исключению дублирования кода: так, при реализации операции сложения обычно уместно воспользоваться ранее реализованной операцией составного присваивания со сложением.

Что было разработано (код, содержащий ошибки)

```
// Определение класса "рациональная дробь"
class Rational {
        int num;  // Числитель (может иметь знак)
        int denom;                  // Знаменатель (положителен)

public:
        // Конструктор
        Rational( int _num = 0, int _denom = 1 ) :
                num( _num ), denom( _denom ) {
                if( denom == 0 )
                                throw out_of_range( "Illegal denominator" );
                if( denom < 0 ) {
                                num = -num;
                                denom = -denom;
                }
        }
        int getNum() const {      return num; }
        int getDenom() const { return denom; }
```

Что было разработано (код, содержащий ошибки)

```
// Перегрузка арифметических операций
Rational& operator+=( const Rational& arg2 ) {
        // Вычисление наибольшего общего делителя
        // числителя и знаменателя
        int cmnDivisor = gcd( denom, arg2.denom );

        num = num * (arg2.denom / cmnDivisor) +
                arg2.num * (denom / cmnDivisor);
        denom = denom / cmnDivisor * arg2.denom;

        return *this;
}

Rational operator+( const Rational& arg2 ) const {
        Rational result( num, denom );
        return result += arg2;
}

// Другие арифметические операции
// ...
```

Что было разработано (код, содержащий ошибки)

```
        // Перегрузка операций отношения
        int operator<( const Rational& arg2 ) const {
                int cmnDivisor = gcd( denom, arg2.denom );
                return num * (arg2.denom / cmnDivisor) <
                                arg2.num * (denom / cmnDivisor);
        }

        int operator==( const Rational& arg2 ) const {
                return num == arg2.num && denom == arg2.denom;
        }
            //...
private:
        // Служебная функция:
        // вычисление наибольшего общего делителя
        static int gcd( int a, int b ) {
                if( a==0 ) return 1;
                assert( a>0 && b>0 );
                while( a!= b ) a>b ? a-=b : b-=a;
                return a;
        }
};
```

Проблемы разработки тестового кода

- Как убедиться в работоспособности класса Rational и в правильности реализации его функциональности?
- Написать тестовые функции

Например, такие:

```
// Тест корректности создания объекта Rational
bool test1() {
    Rational r1( 1, 3 );
    if( r1.getNum() == 1 && r1.getDenom() == 3 ) return true;
    return false;
}

// Тест для операции отношения равенства
bool test2() {
    Rational r1( 1, 3 );
    Rational r2( 1, 3 );

    if( r1==r2 ) return true;
    return false;
}
```

Например, такие:

```
// Тест для операции составного присваивания
bool test3() {
    Rational r1( 1, 2 );
    Rational r2( 1, 4 );

    r1+=r2;
    if( r1.getNum() == 3 && r1.getDenom() == 4 ) return true;
    return false;
}
```

Проблемы разработки тестового кода

- Как запустить тест?

- cout << "test1 (constructor): ";
- if(test1()==true) cout << "Ok";
- else cout << "failed";
- cout << endl;

Проблемы разработки тестового кода

- Поскольку это придется делать часто, лучше написать функцию…

```
void runTest( bool (*test)(), const char *name = "" ) {
    cout << name << ": ";
    if( true == test() ) cout << "Ok";
    else                 cout << "FAILED";
    cout << endl;
}
```

Проблемы разработки тестового кода

- … а затем вызвать ее много раз

```
void testSequence() {
    runTest( test1, "test1 (constructor)" );
    runTest( test2, "test2 (==)" );
    runTest( test3, "test3 (+=)" );
}
```

Проблемы разработки тестового кода

- Вот, что мы получим в результате ее работы:

```
test1 (constructor): Ok
test2 (==): Ok
test3 (+=): Ok
```

- Действительно ли все в порядке?

Дополним систему тестов

```
// Тест для операции отношения равенства
bool test4() {
    Rational r1( 1, 2 );
    Rational r2( 2, 4 );

    if( r1==r2 ) return true;
    return false;
}

void testSequence() {
    runTest( test1, "test1 (constructor)" );
    runTest( test2, "test2 (==)" );
    runTest( test3, "test3 (+=)" );
    runTest( test4, "test4 (== #2)" );

}
```

- Результат работы:

test1 (constructor): Ok
test2 (==): Ok
test3 (+=): Ok
test4 (== #2): FAILED

- Анализ ошибки

Класс с необходимыми исправлениями

```
// Определение класса "рациональная дробь"
class Rational {
   int num;       // Числитель (может иметь знак)
   int denom;// Знаменатель (положителен)

public:
   // Конструктор
   Rational( int _num = 0, int _denom = 1 ) :
      num( _num ), denom( _denom ) {
      if( denom == 0 )
          throw out_of_range( "Illegal denominator" );
      if( denom < 0 ) {
         num = -num;
         denom = -denom;
      }
      simplify();
   }
    // ...
```

```
private:
   // "Сокращение" дроби
   Rational& simplify() {
      int cmnDivisor = gcd( abs( num ), denom );

      num /= cmnDivisor;
      denom /= cmnDivisor;

      return *this;
   }
    // ...
};
```

Промежуточные итоги

- Разработка тестовых методов весьма полезна и позволяет разработчику убедиться в том, что он реализовал то, что намеревался реализовать.
- Тесты не всегда гарантируют обнаружение ошибок.
- При разработке инфраструктуры тестирования необходимо иметь возможность вносить изменения в состав и порядок запускаемых тестов.

Какие аспекты упущены?

- Модификация состава исполняемых тестов не должна требовать переделки ранее разработанных функций.
- Необходимо предусмотреть возможность относительно просто осуществить запуск всех ранее разработанных тестов, чтобы убедиться, что внесенные изменения не нарушили работоспособность ранее протестированного кода (то есть осуществить то, что называется регрессионным, или регрессивным, тестированием).
- Тесты могут характеризоваться различными атрибутами, а не только тестовой функцией. Такими атрибутами могут быть: наименование теста, уникальный идентифицирующий код, краткое описание теста.

Какие аспекты упущены?

- Исполнение теста может предполагать определенные предусловия.
- Перед выполнением основной тестовой процедуры могут требоваться предварительные действия (например, с целью удовлетворения предусловий теста).
- По окончании выполнения тестовой процедуры могут требоваться некоторые дополнительные действия (например, очистка памяти, восстановление состояния общих для различных тестов объектов и др.).
- Последовательность тестов как отдельная сущность объектной модели может иметь собственные атрибуты и методы.

Какие аспекты упущены?

- Последовательность тестов также может требовать выполнения некоторых предварительных и завершающих действий.
- Исполнение последовательности тестов также может трактоваться как успешное или неудачное в зависимости от того, были ли пройдены все тесты, образующие последовательность, или нет.
- Реализация исполнения тестов в последовательности должна быть отделена от реализации исполнения отдельных тестов.
- Тесты могут подразумевать разный уровень критичности. Например, тест операции == в нашем примере может быть отнесен к критическим, так как эта операция может быть использована при написании других тестов.

Пример мини-фреймфорка

- Интерфейс теста

```
class TestInterface {
public:
    virtual int getId() const = 0;
    virtual const char *getName() const = 0;
    virtual void prerun() = 0;
    virtual bool run() = 0;
    virtual void postrun() = 0;
};
```

Пример мини-фреймфорка

```
// Генератор уникальных целочисленных кодов
class IdGenerator {
	int count;
	static IdGenerator *idGen;

	IdGenerator() {
		count = 0;
	}
public:
	static IdGenerator *IdGeneratorInstance() {
		if( idGen == 0 ) return idGen = new IdGenerator();
		return idGen;
	}

	int getId() {
		return count++;
	}
};
```

Пример мини-фреймфорка

```
// Абстрактный тест
class AbstractTest : public TestInterface {
        string name;                    // Имя теста
        int id;                         // Уникальный код теста
public:
        // Конструктор теста – по умолчанию имя отсутствует
        AbstractTest( const string& _name = "" ) {
                name = _name;
                // Получение экземпляра генератора уникальных кодов
                IdGenerator *idGen = IdGenerator::IdGeneratorInstance();
                // Получение кода теста
                id = idGen->getId();
        }

        int getId() const { return id; }
        const char *getName() const {  return name.c_str();  }

        // Методы prerun() и postrun() по умолчанию пусты
        void prerun() {}
        void postrun() {}
};
```

Пример мини-фреймфорка

- На этой основе можно определить тестовую последовательность
  - TestSuite
- Затем – привязать к ней тесты, реализующие интерфейс TestInterface

- См. пример проекта