В данном фрагменте, очевидно, не хватает многих других полезных операций для работы с рациональными дробями, но для текущего обсуждения его достаточно.

1.3. Проблемы разработки тестового кода

После получения данного первоначального наброска давайте зададимся вопросом о том, каким образом убедиться в работоспособности класса `Rational` и в правильности реализации его функциональности. Самое простое решение состоит в написании небольших тестовых функций. По сути, эти функции являются простейшими модульными тестами:

```
// 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;
}

// 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;
}
```

Здесь тестовая функция формирует значение булевского типа в зависимости от того, пройден тест или нет. По результату запуска каждого теста можно вывести соответствующее диагностическое сообщение:

```
cout << "test1 (constructor): ";
```

```
if( test1()==true ) cout << "Ok";
else                cout << "failed";
cout << endl;
```

Даже из данного маленького фрагмента видно, что каждый тест может характеризоваться дополнительными атрибутами (например, названием и кратким описанием). Эти атрибуты позволят идентифицировать тест.

Очевидно, что при тестировании даже одного метода класса может потребоваться определить некоторую последовательность тестов. То есть задачи запуска и проверки результата работы тестовой функции неизбежно повторяются. Поэтому этот процесс можно реализовать в виде отдельной функции, принимающей конкретный тест в качестве параметра, например, с использованием указателя на тестовую функцию:

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

Теперь запуск последовательности тестов упрощается. Нужно просто несколько раз вызвать функцию `runTest`:

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

В нашем случае запуск функции `testSequence` порождает следующий вывод:

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

Итак, все тесты успешно проходят. Однако данный результат, скорее, иллюстрирует несовершенство тестов, чем отсутствие ошибок в классе `Rational`. Действительно, при анализе операции равенства мы упустили тот факт, что дробь не обязательно задается пользователем в несократимом виде. Очевидно, что с точки зрения математики значения «одна вторая» и «две четверти» должны быть равны. В терминах тестирующей функции это может быть записано следующим образом:

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

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

Последовательность тестов дополняется новым тестом:

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

}
```

В итоге, как и следовало ожидать, только что написанный тест позволяет убедиться в наличии ошибки, имеющейся в реализации класса `Rational`:

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

В данной ситуации ошибка, проявляющаяся в виде некорректно работающей операции сравнения, заключается в отсутствии приведения дроби к несократимому виду в процессе создания объекта.

1.4. Анализ ошибки и устранение дефекта

Таким образом, небольшой набор модульных тестов позволил выявить существенную ошибку в реализации, причем эта ошибка свойственна именно программному решению. На уровне математической модели, в конечном счете, для последующих вычислений не так важно, представлена ли дробь в несократимой форме. Иное дело – компьютерная программа. При переходе от математической модели к программной мы должны учесть ограничения разрядности используемых типов данных (в нашем случае – типа `int` для хранения числителя и знаменателя). Очевидно, что если мы оставим выявленный дефект без внимания, при выполнении вычислений над рацио-

нальными дробями мы можем столкнуться с переполнением разрядной сетки. Причем переполнение может происходить не только в тех случаях, когда это неизбежно и обусловлено спецификой обрабатываемых значений (например, дробей с большими значениями числителей и знаменателей, которые вдобавок являются простыми числами), но и в тех случаях, когда это вызвано только лишь несовершенством программной реализации.

Дефект, выявленный в ходе тестирования, может быть устранен написанием служебного метода для приведения дроби к несократимому виду. Вызов этого метода помещаем в конструктор класса.

Определение класса с исправленной версией конструктора имеет следующий вид:

```
// 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;
  }
  // …
};
```

1.5. Промежуточные итоги

Рассмотренный выше процесс разработки класса и выявления программного дефекта позволяет указать на некоторые важнейшие особенности модульного тестирования, которые были наглядно продемонстрированы в процессе написания тестов для класса `Rational`:

- ❑ 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.

Спроектированная инфраструктура для запуска тестов предельно проста и не учитывает ряд важных аспектов организации модульного тестирования. Вот эти аспекты:

- 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.
- 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.
- 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.

Таким образом, существует множество важных факторов, которые следует принимать во внимание разработчику средств модульного тестирования.

1.6. Конструирование мини-фреймворка для модульного тестирования

Давайте попробуем учесть перечисленные выше аспекты, написав более удобные в дальнейшей работе классы для модульных тестов. Фактически в данном разделе мы построим хотя и несколько игрушечную, но вполне реалистичную мини-инфраструктуру модульного тестирования.

Давайте предположим, что каждый тест характеризуется некоторым уникальным кодом, необходимым, например, для ссылки на результаты тестов в различных документах. Поскольку коды не должны совпадать, для генерации идентифицирующих целочисленных кодов мы разработали небольшой класс, реализующий паттерн «Одиночка» (Singleton). Данный паттерн используется тогда, когда допускается создание и использование не более одного объекта данного класса (то есть мы не сможем по ошибке создать два генератора и тем самым нарушить положение об уникальности идентифицирующих кодов).

При каждом вызове метода `getId()` генератор возвращает последовательные целые числа, начиная с 1 (соответственно общее число гарантированно несовпадающих кодов здесь определяется возможностями типа `int`).

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

  IdGenerator() {
     count = 0;
```