электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.4'23

Метод генерации регрессионных модульных тестов

***Дерягин Д.А.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
*кафедра «*Программное обеспечение ЭВМ и информационные технологии*»*

Научный руководитель: Ломовской И.В., старший преподаватель
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
bauman@bmstu.ru

Введение

В отличие от генерации тестовых данных модульные тесты для классов должны иметь некоторую последовательность вызовов методов, а не только набор входных параметров.

Модуль представляет собой некоторый функционально законченный компонент системы, состоящий из следующих элементов:

- конструктор для инициализации модуля;
- список доступных операций над модулем;
- текущее состояние;
- список целей для покрытия тестами.

Предлагаемый метод разрабатывается применительно к объектно-ориентированным программам, где в качестве модулей выступают классы, которые подходят под вышеописанное определение. В качестве операций выступают методы класса, который также имеет набор конструкторов для инициализации, состояние в виде собственных полей и сам программный код. В классе должны быть выделены цели для покрытия согласно одному из критериев тестового покрытия [1]. В качестве целей могут выступать операторы исходного кода, а также ветки или пути в графе потока управления. Метод можно обобщить и на другие компоненты, в которых выделяются перечисленные элементы. В качестве модулей могли бы выступать, например, отдельные подсистемы, являющиеся частями некоторой распределенной системы или просто веб-службы.

Возвращаясь к объектно-ориентированному подходу, отметим, что класс имеет некоторый набор методов, которые изменяют его состояние. А поведение зависит от этого состояния. Таким образом, появляются некоторые сценарии использования класса,

представляющие собой последовательность вызовов методов. Кроме того, тестируемый класс может выступать как часть библиотеки. В этом случае заранее неизвестно как он будет использоваться, а значит, модульные тесты должны покрывать всевозможные сценарии использования, чтобы убедиться, что класс функционирует должным образом. В этом случае возможно лишь написание модульных тестов. Системное тестирование провести не удастся из-за отсутствия клиентского кода.

Описываемый метод предназначен для автоматической генерации тестов. Под тестом в данном контексте понимается тестовый сценарий. В первую очередь необходимо рассмотреть, из каких частей он состоит.

Достаточно вспомнить какие операции проделывает разработчик во время написания автоматических модульных тестов при объектно-ориентированном подходе. В первую очередь необходимо создать экземпляр тестируемого класса, воспользовавшись одним из конструкторов; далее, вызвав некоторую последовательность методов, перевести класс в желаемое состояние; вызвать тестируемый метод класса; сравнить полученные результаты с ожидаемыми. Одной лишь последовательности вызовов недостаточно для создания теста. Большинство методов имеют входные параметры, а значит, для них должны быть подобраны конкретные значения. Результатом работы метода, который проверяется в тесте, может быть возвращаемое значение, либо изменение внутреннего состояния. Если класс использует другие классы, то при модульном тестировании, они заменяются заглушками или используются реальные экземпляры для проведения интеграционного тестирования.

Таким образом, генерируемые тестовые сценарии будут соответствовать примерно следующей структуре:

1. Создание зависимых классов или заглушек, используемых в качестве параметров в тестируемом классе.
2. Вызов последовательности методов зависимых классов с некоторыми параметрами для привидения их в какое-либо состояние.
3. Создание объекта тестируемого класса через один из доступных конструкторов.
4. Вызов последовательности методов тестируемого класса для приведения его в какое-либо состояние.
5. Вызов метода, содержащего целевой объект для покрытия тестом.

Добавив проверочные утверждения согласно схеме, которая будет описана далее, можно получить набор тестов, которые предположительно будут фиксировать текущее поведение модуля.

Описание метода генерации модульных тестов

На Рис. 1 и

Рис. 2 представлена IDEF0 диаграмма описываемого метода.

На вход подается тестируемый класс, из которого формируется список сигнатур методов, список целей для покрытия согласно выбранному критерию, пути в графе потока управления и инструментированный код этого класса. Модуль инструментирования выполняет обход синтаксического дерева каждого метода и таким образом составляет список путей в графе потока управления.

При помощи генетического алгоритма и модуля генерации тестов формируются тестовые сценарии, которые подаются на вход модуля исполнения тестов. При исполнении тестов для каждого тестового сценария формируется список покрываемых им целей и значение функции приспособленности, если ни один из тестов текущего набора не покрывает целевой объект. После завершения алгоритма генерации результирующий набор минимизируется для устранения избыточности и записывается в текстовый файл в виде программного кода на одном из языков программирования.

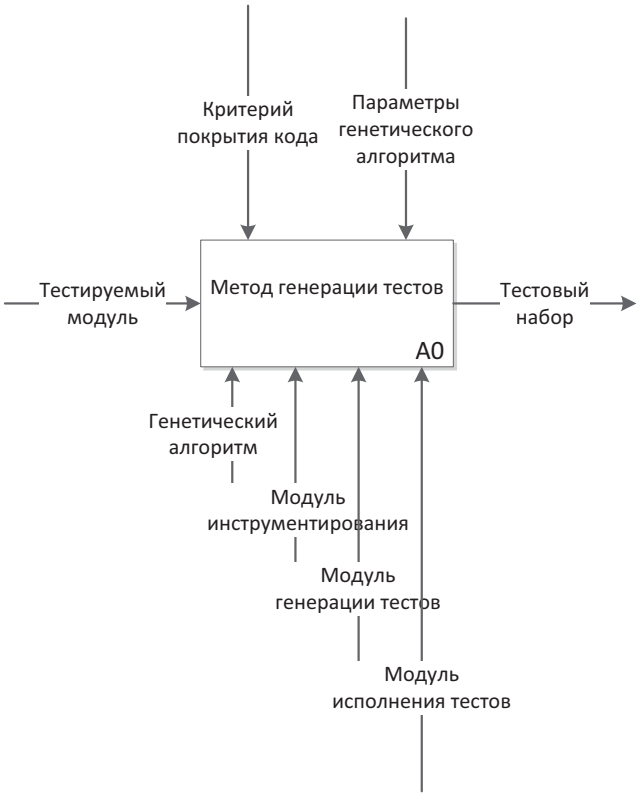

Рис. 1. Метод генерации тестов. IDEF0 первого уровня

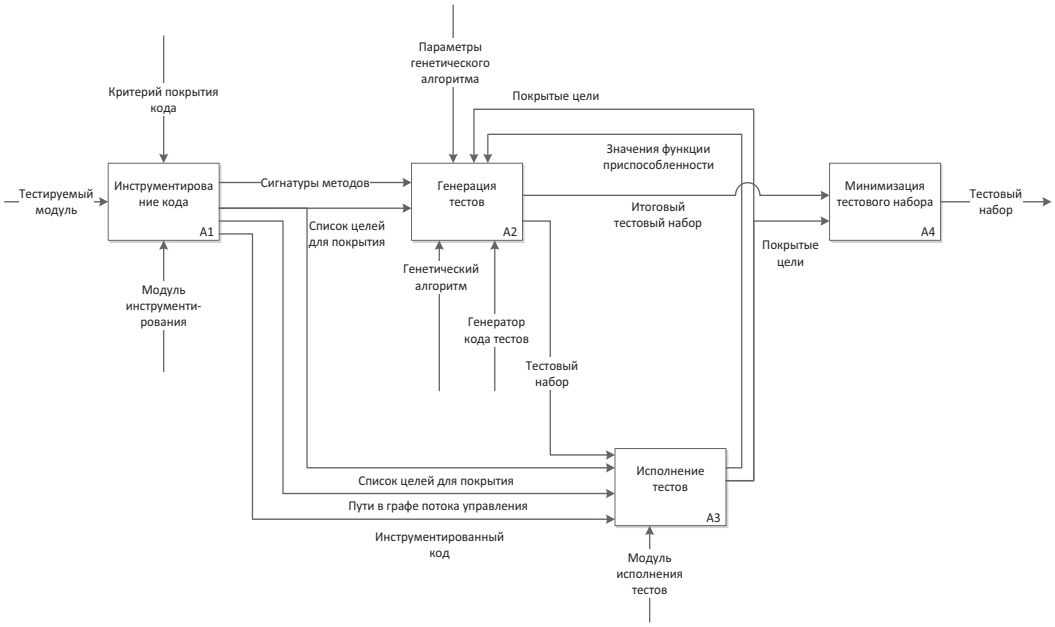

Рис. 2. Метод генерации тестов. IDEF0 второго уровня

Использование генетического алгоритма

На Рис. 3 представлена схема генетического алгоритма [2] для генерации тестов.

В схеме представлены основные шаги генетического алгоритма применительно к генерации тестовых сценариев. Среди рассмотренных критериев тестового покрытия был выбран критерий C1 [1]. Сравнение метрик тестового покрытия не является целью данной работы, поэтому для реализации был выбран лишь один критерий, который влияет на задание функции приспособленности. Список объектов для покрытия формируется на основании выбранного критерия на первом шаге алгоритма.

На втором шаге формируется начальная популяция, которая представляет собой определенное, заранее заданное число особей. Особь однозначно определяется ее хромосомой. В качестве хромосомы для генетического алгоритма выступает тестовый сценарий. В нашем случае хромосома представляет собой вызов конструктора объекта и некоторой последовательности вызовов методов с определенными параметрами. Детальный процесс формирования хромосомы будет рассмотрен в следующем параграфе. Создание начальной популяции происходит случайным образом.

В ходе работы алгоритма новые тестовые сценарии будут генерироваться до тех пор, пока не будут покрыты все целевые объекты. Но поскольку в некоторых случаях невозможно достичь 100% покрытия кода, следует добавить ограничение на максимальное время исполнения алгоритма для предотвращения зацикливания.

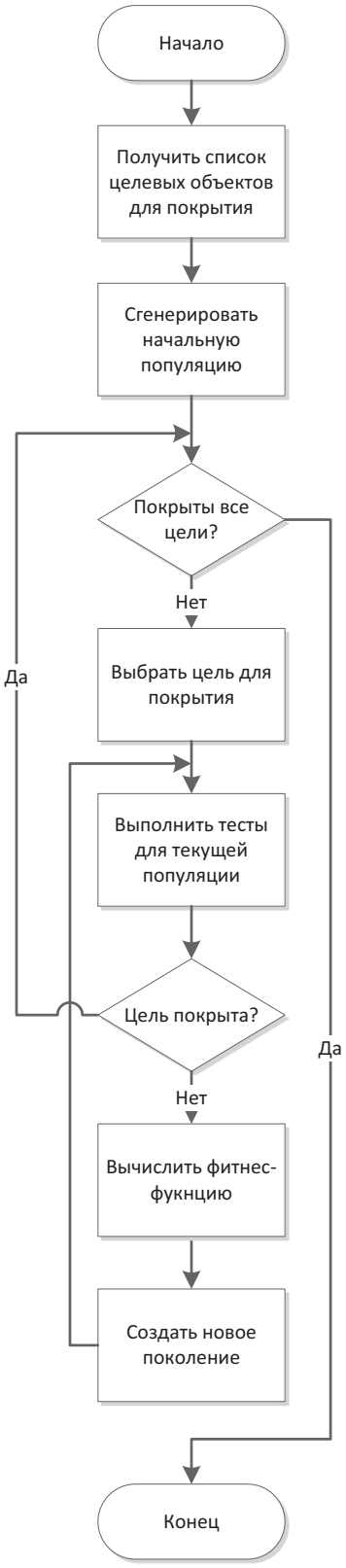

Рис. 3. Схема генетического алгоритма для генерации тестов

На следующем шаге выбирается цель, которую нужно покрыть во время текущей итерации. Наличие покрываемой цели в случае генетического алгоритма является обязательным, так как на ее основе вычисляется функция приспособленности. Значение этой функции влияет на выбор особей, которые наиболее близки к покрываемой цели. Без задания цели и правильного выбора функции приспособленности генетический алгоритм будет работать не быстрее чем случайный генератор [3]. Для покрытия конкретной цели также стоит задать ограничение в виде максимального количества итераций. То есть, если после создания некоторого числа поколений в них не появилось теста, покрывающего заданную цель, будем считать, что покрыть ее невозможно за целесообразное время.

Для того чтобы проверить, есть ли в текущей популяции тестовый сценарий, покрывающий заданную цель, необходимо запустить на исполнение данный тестовый набор и получить метрику покрытия. На основе нее можно вычислить функцию приспособленности для каждой особи. Наиболее приспособленные особи имеют более высокую вероятность перейти в следующее поколение. Таким, образом, при правильном задании функции приспособленности возможно улучшение покрытия целевого объекта с каждой итерацией. Стоит отметить, что если во время исполнения тестов, была покрыта новая цель, но не текущая цель, то ничего не мешает сразу добавить такой тест в результирующий набор.

На последнем шаге генетического алгоритма создается новое поколение путем применения операторов скрещивания и мутации.

После завершения всех итераций генетического алгоритма результирующее множество может содержать излишние тестовые сценарии. Обусловлено это тем, что на очередной итерации может появиться тест, который покрывает не только текущий целевой объект, но и цель из прошлой итерации. Это означает, что необходимо произвести обработку результирующего множества. Это делается достаточно простым способом. На каждой итерации выбирается тест, покрывающий наибольшее число целей и добавляется в итоговое множество в том случае, если он покрывает какой-либо новый объект. Таким образом, излишние тесты, не добавленные в итоговый набор, просто отбрасываются.

Формирование хромосомы

В случае использования генетического алгоритма для генерации тестовых данных в качестве хромосомы принимается вектор входных параметров тестируемой функции [3]. Для тестирования модуля этого не достаточно, так как модуль имеет набор операций, которые можно над ним совершать. Последовательность вызовов этих операций с

определенными параметрами является неким сценарием использования данного класса. В случае модульного тестирования нужно рассмотреть всевозможные сценарии работы. Как уже было отмечено ранее, хромосома однозначно определяет особь, которая в нашем случае представляет собой тестовый сценарий. Таким образом, генерируемый тестовый сценарий должен включать в себя не только входные параметры, но и последовательность вызовов методов. Кроме того, в сценарий нужно включить проверки, которые является неотъемлемой частью теста. Как минимум, нужно проверить возвращаемое значение и конечное состояние объекта.

На Диагр. 1 представлена грамматика [4], описывающая хромосому. Нетерминалы заключены в угловые скобки.

<хромосома> := <операции> @ <значения>
<операции> := <операция> {: <операции>}?
<операция> := $id = конструктор({<параметры>}?)
| $id = класс # null
| $id .метод({<параметры>}?)
<параметры> := <параметр> {, <параметры>}?
<параметр> := встроенный тип {<генератор>}?
| $id
<генератор> := [мин; макс]
| [класс генератора]
<значения> := <значение> {, <значения>}?
<значение> := int
| double
| boolean
| String

Диагр. 1 Грамматика, описывающая хромосому.

Как видно из Диагр. 1 хромосома состоит из двух частей, разделенных знаком «@». Левая часть хромосомы состоит из последовательности вызовов конструктора и методов, разделенных знаком «:». Вторая часть содержит набор параметров вызова этих операций. В работах, где генерируются тестовые данные, хромосома состоит только из второй части [3].

Каждая операция представляет собой либо создание нового объекта, либо вызов метода. В первом случае объекту присваивается уникальный идентификатор, а во втором этот идентификатор используется для вызова. Особый случай – это замена объекта

пустым значением null. Оно указывается вместе с названием класса, вместо которого будет подставляться null в качестве входного параметра. Значение null присваивается идентификатору объекта, поэтому необходимо учитывать, что нельзя вызывать методы над этим объектом, а можно лишь передавать его в методы тестируемого класса.

В качестве параметров вызова методов выступают либо простые встроенные типы, либо идентификатор ранее созданного объекта. Строки в ряде языков программирования не относятся к простым типам и представляются в виде объекта. Однако их все равно можно отнести к встроенным типам в рамках описываемого метода. Ключевой разницей между встроенными типами и другими классами является то, что встроенные типы создаются при помощи специального генератора, а объекты класса путем вызова их конструктора с определенными параметрами. Стандартные генераторы возвращают случайное число в заданном диапазоне, который можно менять. Также есть возможность задания своего собственного генератора, который будет возвращать значения по каким-либо своим правилам.

Во второй части хромосомы содержатся сами значения параметров встроенных типов, которые имеют один из перечисленных в грамматике типов.

Не все хромосомы, описываемые грамматикой на Диагр. 1 являются допустимыми. На них накладываются дополнительные ограничения:

все идентификаторы, присутствующие в хромосоме должны быть проинициализированы раньше, чем будут использованы в качестве входных параметров;

для каждого встроенного типа, присутствующего в качестве параметра в левой части хромосомы, должно быть соответствующее значение в правой части;

вызовы конструкторов и методов, присутствующие в хромосоме должны соответствовать реальным сигнатурам методов тестируемого класса.

Рассмотрим пример допустимой хромосомы:

$a = A() : $b = B() : $b.setX(int) : $a.m1(String, $b) @ 5, “abc” (1)

Допустим, что необходимо протестировать метод m1 класса A. Сам класс имеет конструктор без параметров, однако метод принимает на вход строковый параметр и экземпляр класса B. Для того, чтобы передать экземпляр класса B в качестве параметра, его необходимо сначала создать, вызвав его конструктор. Далее возможны какие-либо вызовы методов класса B, которые некоторым образом изменят его состояние. Последним действием идет вызов тестируемого метода m1. Как видно из примера, идентификаторам $a и $b присваивается значения раньше, чем они используются. В левой части хромосомы используется лишь два встроенных типа, значения для которых перечислены в правой

части. Полагая, что сигнатуры методов классов A и B соответствуют тем, что представлены в хромосоме, можно считать, что такая хромосома является допустимой.

Операторы мутации и скрещивания

В алгоритме, представленном на Рис. 3 присутствует такая операция, как создание нового поколения. Этот процесс происходит путем отбора наиболее приспособленных особей и применения к ним операторов скрещивания и мутации. Перечислим операции, используемые описываемым методом генерации тестов.

Изменение значения параметра

Генерируем новое значение для какого-либо параметра из правой части хромосомы.

Пример:

$a = A() : $a.m1(String, int) @ **“abc”**, 7

$a = A() : $a.m1(String, int) @ **“”**, 7

Изменение конструктора

Создаем класс одним из других его конструкторов, если такие присутствуют. Если предыдущий конструктор принимал какие-либо параметры, то они удаляется. Для нового конструктора необходимо добавить требуемые параметры.

Пример:

$a = A() : $a.m1(String) @ “abc”

$a = **A(int)** : $a.m1(String) @ **3**, “abc”

Вставка вызова метода

В случайное место добавляется вызов метода, требуемые параметры также создаются.

Пример:

$a = A() : $a.m1(String) @ “abc”

$a = A() : **$a.m2(int)** : $a.m1(String) @ **5**, “abc”

Позиция для вставки метода определяется случайным образом, но эта позиция должна быть между инициализацией класса и до его первого использования в качестве параметра.

Удаление вызова метода

Удаление какого-либо случайно выбранного вызова метода кроме последнего. Неиспользуемые объекты и значения также удаляются.

Пример:

$a = A() : **$a.m2(int)** : $a.m1(String) @ **5**, “abc”

$a = A() : $a.m1(String) @ “abc”

В данном примере значение “5” больше не используется, поэтому его можно удалить. Объект “$b” используется в качестве параметра для тестируемого метода, поэтому он остается в составе хромосомы.

Скрещивание хромосом

Отбираются две хромосомы для скрещивания, в каждой из них выделяется случайный участок, который будет участвовать в скрещивании. Участок должен быть между конструктором целевого класса и последним вызовом метода. После чего, выбранные участки меняются местами, ненужные вызовы методов или конструкторов удаляются, в случае необходимости добавляются новые. Используемые параметры также меняются местами.

Пример:

$a=A() : $a.m1(int) : $a.m2(String) : $a.m3(double) @ 1, “abc”, 0.5

$a=A() : $a.m4(int, int) : $a.m5(String, int) @ 1, 2, “abc”, 3

$a=A() : $a.m1(int) : $a.m4(int, int) @ 1, 1, 2

$a=A() : $a.m2(String) : $a.m3(double) : $a.m5(String, int) @ “abc”, 0.5, “abc”, 3

В приведенном примере из первых двух хромосом в результате скрещивания получаются две другие хромосомы. Подчеркиванием выделены скрещиваемые участки, которые меняются местами. Соответствующие параметры также переходят в другую хромосому. Поскольку все необходимые объекты уже присутствуют в каждой из хромосом, никаких дополнительных операций не требуется.

Стоит также отметить, что в случае наличия повторяющихся идентификаторов, их необходимо переименовать.

Генераторы значений

Для подбора значений параметров в хромосоме используются генераторы. Каждый встроенный тип имеет свое название. Если в сигнатурах методов встречаются только названия этих типов, то используется генератор по умолчанию.

Ниже перечислены предусмотренные виды генераторов по умолчанию.

Integer и double: с вероятностью 10% генерируются нулевые значения. С вероятностью 30% генерируются отрицательные значения. С вероятностью 60% генерируются положительные значения.

Boolean: значения true и false генерируются случайно с одинаковой вероятностью, равной 0.5.

String: строки создаются из символов латинского алфавита в нижнем и верхнем регистре и цифр. Пустая строка генерируется с вероятностью 20%. В заданном интервале с равномерным распределением генерируется длина строки. Вероятность последующих вставок символа равна 0.5^{n+1}.

Также можно создавать собственные классы генераторов.

Генератор коллекций: Вероятность добавления нового элемента в коллекцию равна 0.5^{n+1}. В 20% случаев в качестве значения добавляется null.

Возможность конфигурирования используемых генераторов позволит в будущем обрабатывать более сложные сценарии, когда значения, получаемые стандартным образом, не смогут обеспечить необходимую метрику покрытия кода из-за ограниченности своих значений.

Использование заглушек

В объектно-ориентированных программах не так часто встречаются классы, которые не имеют зависимостей от других классов. Как правило, тестируемый класс использует другие классы для выполнения каких-либо операций. Необходимо рассмотреть различные способы написания тестов в подобных случаях.

Метод позволяет использовать реальные экземпляры зависимых классов. Для этого необходимо лишь знать их конструкторы и сигнатуры методов. Генерируемые хромосомы в этом случае будут содержать не только конструкторы и вызовы методов тестируемого класса, но и его зависимостей. Подобный тест является интеграционным, а не модульным. С одной стороны, проведение интеграционного тестирования также необходимо. С другой стороны, если стоит задача протестировать модуль, такой подход неприемлем, поскольку

тестируется не только целевой класс, но и зависимости, которые могут содержать ошибки. Исправление этих ошибок изменит их поведение, а значит и результаты тестов, что неприемлемо при модульном тестировании.

Для изолированного тестирования, как правило, используют заглушки, которые представляют собой наследников зависимого класса, но с переопределенными методами, которые возвращают заранее определенные значения. При автоматической генерации неизвестно заранее, какие значения требуются для обеспечения требуемого покрытия кода, поэтому их также необходимо генерировать. Совместное использование генераторов и заглушек позволит создать наиболее полный тестовый набор. Например, если тестируемый класс A принимает на вход другой класс B, то можно использовать заглушку класса B, возвращаемые значения которой будут генерироваться стандартным или пользовательским генератором.

Проставление проверочных утверждений

Тестовый набор, который обеспечивает даже сто процентное покрытие кода, не является приемлемым, если в нем отсутствуют проверочные утверждения.

В том случае, когда тестами покрываются функциональные требования, разработчик теста заранее знает ожидаемые возвращаемые значения. Эти значения содержатся в спецификации к тестируемому модулю. И даже в случае тестирования белого ящика, разработчик проставляет проверочные утверждения на основе функциональных требований.

Для решения поставленной ранее задачи, где спецификация может отсутствовать, нет необходимости знать требуемые значения. Генерируемые тесты должны лишь некоторым образом фиксировать текущее поведение программы. Поэтому в качестве ожидаемых значений проставляются значения, полученные в результате исполнения тестового набора.

В процессе генерации тестов проверочные утверждения проставляются согласно следующим правилам:

Если вызываемый метод возвращает некоторое значение, то добавляется утверждение, которое сравнивает это значение с фактическим.

Если в результате выполнения теста было выброшено исключение, то метод оборачивается в блок try catch, тип выброшенного исключения сравнивается с фактическим.

Если в результате выполнения метода меняется состояние объекта, то есть изменяется какое-либо поле, то следующим оператором проставляется проверочное утверждение, которое сравнивает результирующее состояние с фактическим.

Классификация обнаруживаемых ошибок

В общем случае под программной ошибкой понимается недокументированное или нежелательное поведение программы на определенных входных данных.

Майерс дает следующее нестрогое определение: «Если программа не делает того, чего пользователь от нее вполне обосновано ожидает, значит налицо программная ошибка» [5].

Кроме того, существует также мнение, что расхождение между программой и ее спецификацией и есть программная ошибка. В некотором смысле это не совсем верно, так как ошибки в спецификации не являются ошибками согласно такому определению.

В рамках данной работы не будем рассматривать ошибки, возникающие на стадии формулирования и сбора требований, проектирования, и т.д. Также будем брать в расчет нефункциональные требования, которые приводят к ошибкам безопасности, совместимости, взаимодействия, производительности.

Рассмотрим лишь следующие типы ошибок:

1. Функциональные ошибки.

Под функциональной ошибкой понимается возвращение программой неверных результатом с точки зрения функциональных требований.

2. Исключительные ситуации.

Возникают в случае невозможности выполнить требуемую операцию.

3. Зависание.

Невозможность выполнить требуемую операцию за определенный промежуток времени.

4. Аварийное завершение.

Завершение работы программы до момента получения результатов.

Для поиска различных дефектов, относящихся к третьей и четвертой категориям, существуют специальные инструменты статического и динамического анализа. Разрабатываемый метод не предполагает генерации тестов, помогающих в нахождении подобных ошибок.

Также нельзя утверждать, что сгенерированные тесты позволят обнаружить функциональные ошибки, т.к. разрабатываемый метод генерирует тесты без

спецификации, значит требуемое поведение неизвестно, а результаты зафиксированные тестами могут быть неверными.

Целью данного метода является генерация характеристических тестов [6], которые фиксирует настоящее поведение программы. Майкл Физерс в своей книге рекомендует писать подобные тесты, прежде чем приступать к внесению изменений в унаследованный код. После модификации модуля следует запустить характеристические тесты, по которым будет видно на каких данных и каким образом изменилось поведение программы. Разработчик должен самостоятельно проверить, какие изменения относятся к желаемым, а какие были привнесены случайным образом. Зная новые функциональные требования, разработчик может исправить тесты, чтобы они отвечали этим требованиям.

Методика проведения экспериментов

Целью проводимых экспериментов является оценка эффективности работы предложенного метода генерации тестов.

Для оценки тестового набора в процессе генерации используется критерий покрытия веток C1. Таким образом, в первую очередь необходимо экспериментальным путем определить какой уровень тестового покрытия обеспечивают генерируемые тесты. Также необходимо учесть время, затрачиваемое на генерацию тестового набора.

Тем не менее, нельзя судить о качестве тестового набора лишь по проценту тестового покрытия. Во-первых, критерии покрытия никаким образом не проверяют наличие проверочных утверждений. Если тесты лишь покрывают все ветки кода, но не имеют проверочных утверждений, то они будут успешно выполняться и после внесения модификаций. В то время как основной целью создания регрессионных тестов является способность обнаруживать изменения в коде.

Возникает задача каким-то образом протестировать сами тесты. Для ее решения существует специальная методика, которая называется мутационное тестирование (mutation testing) [7]. Для проверки эффективности тестового набора в исходный код тестируемой программы или модуля вносят различные изменения. Получившуюся программу называют мутантом. Для каждого такого мутанта исполняют тестовый набор. Если какой-либо тест завершается с ошибкой, то говорят, что этот тест убивает данного мутанта. Качество тестового набора оценивается по проценту убитых мутантов. Если же большое количество мутированных программ проходит тесты, то это говорит о неэффективности или неполноте тестового набора. Для проведения мутационного тестирования существуют специальные библиотеки, которые умеют генерировать

мутантов и строить итоговые отчеты. Сложность создания мутантов заключается в том, что должны соблюдаться три условия:

1. Тест должен достигать мутированного блока.
2. Мутант должен вносить изменения, которые изменят поведение программы, иначе такой мутант называется эквивалентным. Эквивалентных мутантов невозможно убить, а значит, они лишь снижают эффективность мутационного тестирования.
3. Измененное поведение должно влиять на вывод программы, который может быть проверен тестом.

Для мутационного тестирования сгенерированных тестов была выбрана библиотека PIT [8]. Это наиболее быстрая и простая в использовании библиотека для языка Java, которая активно развивается.

Для проведения экспериментов были взяты несколько классов из стандартного пакета Java. Для выбранных классов были сгенерированы тестовые наборы и мутационное тестирование.

Типы модификаций

Виды вносимых в программу модификаций описаны ниже. Такие изменения называются операторами мутации, которые специально разрабатываются, чтобы их было трудно обнаружить [8]. Кроме того, генерируемые мутанты не должны быть эквивалентными.

Изменение граничных условий

Операторы <, > заменяются соответственно на <=, >= и наоборот.

Изменение условий на противоположные

Условия меняются согласно принятым правилам [8].

Изменение арифметических и логических выражений

Выражения меняются согласно принятым правилам [8].

Изменение операторов инкрементирования

Заменяет операторы инкрементирования на операторы декрементирования и наоборот.

Изменение знака арифметических выражений

Меняет знак результата математического выражения.

Изменение возвращаемых значений

В зависимости от типа возвращаемого значения в результат вносятся некоторые изменения. Например, true заменятся на false для boolean, или прибавляется единица для integer.Также могут возвращаться нулевые значения, либо null для ссылок.

Удаление вызовов методов, не имеющих возвращаемого значения

При наличии в каком-либо участке кода вызова метода, который не возвращает никакого значения, этот вызов удаляется.

Эксперименты

Эксперименты проводились согласно описанной методике на вычислительной машине с процессором Intel Core i5-2410M, тактовой частой 2.3 ГГц, 8гб оперативной памяти, 64 разрядной операционной системой Windows 7.

Генетический алгоритм имеет следующие параметры запуска:

1. Максимальное количество попыток покрытия одной ветки. Если ветку не удается покрыть за заданное число итераций, то выбирается следующая ветка.
2. Максимальное время выполнения. Алгоритм может выполняться больше отведенного промежутка времени. Время работы не будет превышать максимальное время выполнения + одна итерация цикла по всем веткам.
3. Размер популяции. Задает количество особей в рамках одной итерации.

Stack

Класс Stack реализует классическую структуру данных – стек. Присутствуют операции вставки и удаления из стека, поиск элемента, а также некоторые другие методы. В

Таблица ***1*** представлены основные характеристики класса и результаты работы метода. Количество строк кода лишь косвенным образом дает представление и сложности класса. Также представлено количество открытых методов, которые могут вызываться для обеспечения тестового покрытия. Как видно из результатов для класса Stack удалось достигнуть 100% величины тестового покрытия всего лишь за 1 секунду. Генетический алгоритм запускался со следующими параметрами: Максимальное количество попыток покрытия одной ветки – 10, Максимальное время выполнения – 60, размер популяции – 10.

Таблица 1

Генерация тестов для класса Stack

Количество строк кода в классе	119
Количество открытых методов	6
Количество покрытых целей	10/10
Процент достигнутого покрытия	100%
Количество исполнений тестов	130
Время работы	1 сек

Результаты мутационного тестирования представлены в Таблица 2. Как видно из результатов 100% покрытие веток кода вовсе не гарантирует 100% обнаружение модификаций. Тем не менее, для такого простого класса удалось достигнуть высокого процента убитых мутантов – 93%. Сгенерированным тестам не удалось обнаружить лишь одного мутанта. Модификация, которая не была обнаружена ни одним тестом, заключалась в том, что был изменен знак в возвращаемом выражении. На первый взгляд, покажется, что такое изменение должно быть обнаружено во время проверки возвращаемого значения. Однако на сгенерированных тестовых данных в арифметическом выражении прибавлялся и вычитался нуль. Такая модификация существенно влияет на поведение программы, но конкретный набор тестов ее никак не обнаружил.

Таблица 2

Мутационное тестирование для класса Stack

Общее число убитых мутантов	14/15 (93%)
Изменение граничных условий	1/1 (100%)
Изменение условий на противоположные	3/3 (100%)
Изменение арифметических и логических выражений	2/3 (67%)
Изменение возвращаемых значений	6/6 (100%)
Удаление вызовов методов	2/2 (100%)

StringTokenizer

Вспомогательный класс StringTokenizer из стандартного пакета Java предоставляет возможность разбиения строк на символьные подстроки. Он имеет чуть более сложную логику, чем класс Stack. За 2 секунды удалось достигнуть 97% тестового покрытия, что отражено в Таблица 3. Генетический алгоритм запускался со следующими параметрами: Максимальное количество попыток покрытия одной ветки – 30, Максимальное время выполнения – 60, размер популяции – 10.

В некоторых случаях ветки не удается покрыть из-за того, что стандартные генераторы попросту не возвращают необходимых тестовых данных, которые могут обеспечить попадание в необходимую ветку. Также может быть такая ситуация, когда вероятность генерации каких-либо данных очень низка, например, если класс ожидает на вход строку определенного формата. Подобная проблема, как правило, решается путем написания собственных генераторов. Основным недостатком в данном случае является необходимость привлечения разработчика, который понимает логику работы модуля и может написать подходящий генератор.

Таблица 3

Генерация тестов для класса StringTokenizer

Количество строк кода в классе	340
Количество открытых методов	9
Количество покрытых целей	33/34
Процент достигнутого покрытия	97,05%
Количество исполнений тестов	2790
Время работы	2 сек

Как видно из Таблица 4 при проведении мутационного тестирования для класса StringTokenizer удалось достичь уже меньшего покрытия мутантов, чем для класса Stack. Хуже всего удавалось обнаружить изменение граничных условий, что объясняется достаточно просто. При покрытии всех веток кода для условного оператора достаточно сгенерировать два значения, которые обращают условное выражение в истину и ложь. Изменение граничного условия никак не влияет на результат работы на каких-либо данных кроме граничных. Написание подобных тестов возможно при использовании решателей булевых формул, однако поиск последовательности вызовов методов, который обеспечит необходимые данные на границе перехода, является достаточно нетривиальной задачей.

Обнаружение мутанта, который удаляет вызов метода, не возвращающего значение, возможно при наличии проверочных утверждений, которые проверяют состояние класса после вызова метода. Если такой тест завершается успешно, значит, удаление не привело к изменению состояния, что опять же может произойти только на конкретных тестовых данных.

Таблица 4

Мутационное тестирование для класса StringTokenizer

Общее число убитых мутантов	39/51 (76%)
Изменение граничных условий	9/15 (60%)
Изменение условий на противоположные	18/21 (86%)
Изменение операторов инкрементирования	5/5 (100%)
Изменение возвращаемых значений	6/8 (75%)
Удаление вызовов методов	1/2 (50%)

ArrayList

Класс ArrayList является реализацией динамического списка на основе массива. 100% покрытия ветвей достигнуть не удалось, что указано в

Таблица ***5***. Генетический алгоритм запускался со следующими параметрами: Максимальное количество попыток покрытия одной ветки – 20, Максимальное время выполнения – 120, размер популяции – 10.

В одном из непокрытых блоков генерируется исключение в том случае, если класс не реализует интерфейс «Clonable». Такое возникнуть попросту не может, потому что ArrayList реализует этот интерфейс, что и отражено в комментарии к этому блоку кода.

Два непокрытых блока оказались в методе с модификатором «protected». Этот модификатор означает, что вызывать метод могут только потомки класса. Такой результат говорит о том, что не были учтены некоторые особенности объектно-ориентированных языков. Protected-методы можно было бы отнести к открытым методам и тестировать наравне с ними.

Оставшиеся шесть блоков относятся к методам сериализации и десериализации, которые являются закрытыми и вызываются через специально предусмотренный механизм платформы, позволяющий проводить сериализацию путем вызова этих закрытых методов. Разумеется, из класса эти методы не вызываются, а значит невозможно было их покрыть предложенным способом генерации тестов.

Учитывая комментарии о непокрытых блоках, можно судить о том, что генератору удалось обеспечить 100% покрытие веток кода в класс ArrayList.

Таблица 5

Генерация тестов для класса ArrayList

Количество строк кода в классе	525
Количество открытых методов	22
Количество покрытых целей	57/66
Процент достигнутого покрытия	86,36%
Количество исполнений тестов	11740
Время работы	130 сек

Результаты мутационного тестирования ArrayList представлены в **Ошибка! Источник ссылки не найден.**

Таблица ***6***. Покрытие 66% мутированных программ говорит о необходимости дополнения тестов по крайней мере тестированием граничных условий.

Таблица 6

Мутационное тестирование для класса ArrayList

Общее число убитых мутантов	83/126 (66%)
Изменение граничных условий	11/19 (58%)
Изменение условий на противоположные	24/30 (80%)
Изменение арифметических и логических выражений	16/32 (50%)
Изменение операторов инкрементирования	5/7 (71%)
Изменение возвращаемых значений	14/17 (82%)
Удаление вызовов методов	13/21 (62%)

Заключение

Был рассмотрен процесс генерации модульных тестов. Предложен метод генерации регрессионных модульных тестов, фиксирующих текущее поведение модуля. Описанный метод позволяет добавлять новые генераторы тестовых значений, проводить интеграционное тестирование, а также тестировать модули изолированно, используя заглушки для зависимых классов. Сгенерированные тесты должны отслеживать как можно большее количество модификаций в модуле.

Список литературы

1. Владимиров М.А. Критерии полноты тестового покрытия в генетических алгоритмах генерации тестов. Режим доступа: http://citforum.ru/SE/testing/completeness_criterion/ (дата обращения: 05.04.2014).
2. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы. М.: Физматлит, 2006. 320 с.
3. Pargas R.P., Harrold M.J., Peck R.R. Test data generation using genetic algorithms // The Journal of Software Testing, Verification and Reliability. 1999. № 9. Pp. 263-282.
4. Tonella P. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis // Evolutionary testing of classes. New York. 2004. Pp. 119-128.
5. Myers G.J., Sandler C., and Badget T. The Art of Software Testing. 3rd ed. New Jersey: Wiley, 2011. 240 pp.
6. Feathers M. Working Effectively with Legacy Code. 1st ed. New Jersey: Prentice Hall, 2004. 456 p.
7. Feathers M. Working Effectively with Legacy Code. 1st ed. New Jersey: Prentice Hall, 2004. 456 p.
8. Mutation testing // Wikipedia. Available at: http://en.wikipedia.org/wiki/Mutation_testing, accessed 18.05.2014.
9. Real world mutation testing. Available at: http://pitest.org/, accessed 18.05.2014.