6. Денинг В., Эссиг Г., Маас С. Диалоговая система «человек-ЭВМ». Адаптация к требованиям пользователя. М.: Мир, 1984.

7. Деревицкий Д. П., Фрадков А. Л. Прикладная теория дискретных адаптивных систем управления. М.: Наука, 1981.

8. Адаптивные системы автоматического управления / под ред. В. Б. Яковлева. Л.: Изд-во Ленингр. ун-та, 1984.

9. Trapeznikov S., Dinenberg F., Kuchin S. InterBase: A Natural Language Interface system for popular commercial DBMSs // Proc. of the EAST-WEST conf. on artificial intelligence, Moscow, 1993. P. 189–193.

10. Раскин Д. Интерфейс: новые направления в проектировании компьютерных систем. СПб.: Символ-Плюс, 2005.

B. A. Al-Nami
Saint-Petersburg state electrotechnical university «LETI»

ADAPTATION OF TEXTS THE OPPOSITE DIRECTION (RIGHT TO LEFT) IN INFORMATION PRODUCTS

This article discusses the basic rules of writing the HTML code for text blocks phase content or line, that means blocks which are mixed within a paragraph of text phrases with different directions of the letter. Article is devoted to the use of markup in HTML, but most of the concepts can also be used for other languages.

Information model, adaptation, users, especially writing and perceptions, interfaces, internet-browsers

УДК 20.53.19, 28.23.13

А. В. Смирнов
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Мутационное тестирование программного обеспечения

Рассматриваются преимущества и недостатки набирающего в последнее время популярность метода тестирования программного обеспечения – мутационного тестирования и возможность его применения в качестве одной из основных характеристик при оценке качества программного обеспечения.

Тестирование, разработка через тестирование, программное обеспечение, мутационное тестирование, покрытие кода, качество программного обеспечения

Многие современные методики разработки программного обеспечения (ПО) не только уделяют большое внимание автоматическому тестированию, но и основываются на нем.

Главным примером сложившейся ситуации является ответвившаяся от «экстремального» программирования *разработка через тестирование* (англ. test-driven development, TDD) – техника, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Одной из основных метрик, которые отражают качество созданного программного обеспечения, является *покрытие кода* (англ. code coverage), показывающее процент тестирования исходного кода программы.

Однако сам по себе даже высокий процент покрытия кода тестами не может свидетельствовать о качестве кода, если не существует метода, позволяющего оценить достоверность и качество самих тестов. Один из таких методов и описывается в данной статье.

Мутационное тестирование (мутационный анализ) – метод тестирования программного обеспечения, который включает небольшие изменения

кода программы [1]. Если набор тестов не в состоянии обнаружить такие изменения, то он рассматривается как недостаточный. Эти изменения называются мутациями и основываются на мутационных операторах, которые или имитируют типичные ошибки программистов (например, использование неправильной операции или имени переменной), или требуют создания полезных тестов.

Мутационное тестирование было предложено Р. Липтоном в 1971 г. [2], а впервые разработано и опубликовано ДеМиллом, Липтоном и Сейвардом [3]. Первая реализация инструмента для мутационного тестирования была создана Тимоти Баддом из Йельского университета в его диссертации (названной «Мутационный анализ») в 1980 г.

Мутационное тестирование производится выбором мутационных операторов и применением их одного за другим к каждому фрагменту исходного кода программы. Результат одного применения мутационного оператора к программе называется мутантом. Если набор тестов способен обнаружить изменение (т. е. один из тестов не проходит), то мутант называется убитым. На рис. 1 графически изображен традиционный процесс мутационного тестирования, где сплошные фигуры показывают шаги, которые автоматизированы компьютерными системами, а штриховые – шаги, которые выполняются вручную (*T* – условие выполняется (True), *F* – условие не выполняется (False)).

На данный момент исследован и доступен широкий спектр мутационных операторов, которые по предназначению делятся:

1) на операторы императивных языков программирования;

2) операторы объектно-ориентированных языков [4];

3) операторы для параллельного программирования [5];

4) операторы для структур данных (контейнеры и др.) [6].

Например, для императивных языков могут быть использованы следующие операторы:

– инвертация условия (например, == превращается в !=);

– замена каждого логического выражения на логическую константу «истина» или «ложь»;

– замена каждой арифметической операции на другую (например, + на *, – или /);

– замена каждой логической операции на другую (например, > на > =, == или < =);

– замена каждой переменной на другую (из той же области видимости).

Обе переменные должны иметь одинаковые типы.

Для объектно-ориентированных языков также существуют следующие операторы:

– подмена операторов доступа (public, protected, private);

– смена порядка вызова переопределенных методов при наследовании;

– скрытие/удаление переопределенных переменных и методов при наследовании;

– удаление ключевых слов (для JAVA: super, this, static);

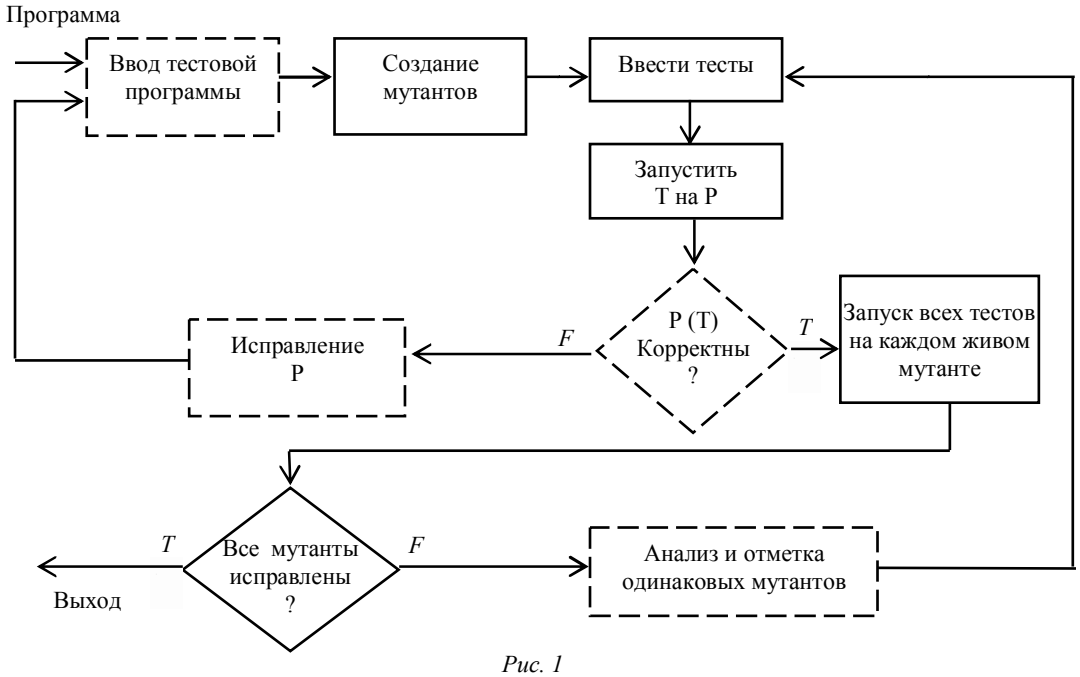

Рис. 1

– вызов new метода с типом класса наследника;

– изменение порядка аргументов при перегрузке (overloading).

Одна из основных проблем мутационного тестирования, из-за которой оно долгое время не пользовалось популярностью, – необходимость больших вычислительных затрат на создание мутантов и дальнейший запуск всех тестов на каждом живом мутанте.

Чтобы решить проблему быстродействия, предлагались различные модификации данного метода [2]:

• Слабые мутации (англ. Weak Mutation) – это аппроксимационный метод, который сравнивает внутренние состояния мутанта и оригинальной программы сразу после исполнения мутантной части программы. Экспериментальные опыты показали, что этот метод может сократить количество запусков не менее чем на 50 % (а обычно более) без серьезного ухудшения достоверности показателя качества тестов.

• Мутации, основанные на схемах (англ. Schema-based Mutation Analysis) – данный метод использует программные «схемы», кодирующие всех мутантов в одну метапрограмму, которая впоследствии компилируется и работает значительно быстрее, чем возможно при использовании стандартного метода интерпретации промежуточных мутационных форм. Предварительное повышение производительности – более чем на 300 % [7], [8]. Этот метод имеет дополнительные преимущества: простота реализации; легкий перенос между программными платформами; использование того же компилятора и среды исполнения, что и при разработке.

• Выборочные мутации (англ. Selective Mutation) – это аппроксимационный метод, при котором из всего множества мутантов, доступных на данный момент, выбираются только те, которые действительно отличаются от других (неизбыточные) и при этом позволяют обеспечить высокую оценку качества тестов. Исследования показали, что селективная мутация обеспечивает почти такой же охват, как и при использовании неселективных мутаций, с сокращением расходов по крайней мере в 4 раза с небольшими программами и до 50 раз с более крупными программами.

• Оптимистичные мутации (англ. Optimistic Mutation) – метод, при котором программная система сама определяет, какие из мутантов эквивалентны, что позволяет значительно сократить время работы тестировщика и соответственно всей системы в целом.

В последнее время, в силу высокого роста производительности вычислительных машин и распространения параллельного программирования, этот метод снова вызывает интерес исследователей в области информатики.

Сейчас доступны библиотеки для проведения мутационного тестирования на современных языках программирования. В случае с Java есть несколько готовых решений, наиболее примечательные из которых – Javalanche и Jumble, Pitest. Однако и они, и другие библиотеки (за исключением Pitest) не особо активно развиваются или практически не имеют интеграции с многими современными системами сборки и другими библиотеками, так что к оптимальному выбору нужно подходить достаточно строго.

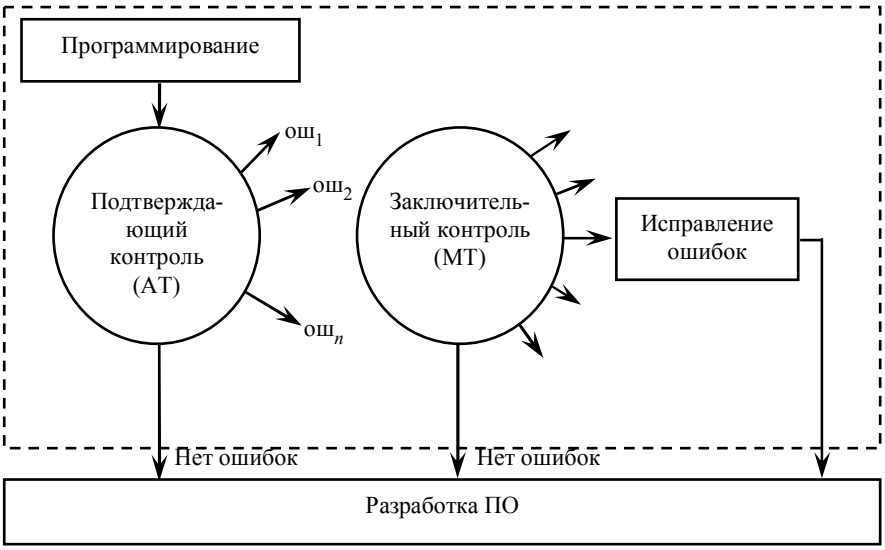

Рис. 2

В целом метод позволяет не только повысить качество создаваемого программного продукта, но и существенно сократить риски и время перехода на другие платформы.

Используя обобщенный структурный метод (ОСМ) и его развитие – функционально-структурную теорию (ФСЕ) [9], процесс автоматизированного тестирования (АТ) и мутационного тестирования (МТ) программного обеспечения может быть представлен с помощью функциональной структуры с подтверждающим (в данном случае им будет являться АТ) и заключительным контролем (МТ) (рис. 2) [10]. Имея множество характеристик, собранных с предыдущих проектов компании, включая индивидуальные показатели программистов, соблюдение сроков, соответствие уровня покрытия тестами кода качеству и затратам на дальнейшую поддержку, данную структуру можно агрегировать в единую укрупненную рабочую операцию [9], тем самым позволяя уже на раннем этапе составления программного проекта иметь возможность прогнозировать уровень качества программного обеспечения, а также требуемые временные и денежные ресурсы.

Сведение многих современных методик программирования в схожие представленной ранее функциональные структуры позволит менеджеру программного проекта на основе данных, собранных с предыдущих проектов, и опыта других фирм делать осознанный выбор, какая из методик будет выгодна для конкретного проекта.

СПИСОК ЛИТЕРАТУРЫ

1. Miller J. C., Clifford J. Maloney. Systematic mistake analysis of digital computer programs // Communications of The ACM – CACM. 1963. Vol. 6, № 2. P. 58–63.

2. Jefferson Offutt A. A Practical System for Mutation Testing: Help for the Common Programmer // Proc. of the Intern. conf. on Test: The Next 25 Years, Washington. DC, 1994. P. 824–830.

3. DeMillo R. A., Lipton R. J., Sayward F. G. Hints on test data selection: Help for the practicing programmer // IEEE Computer. 1978. Vol. 11, № 4. P. 34–41.

4. Yu-Seung Ma, Offutt J., Yong Rae Kwo. MuJava: An Automated Class Mutation System // Software Testing, Verification and Reliability. 2005. № 15(2). P. 97–133.

5. Bradbury J. S., Cordy J. R., Dingel J. Mutation operators for concurrent Java (J2SE 5.0) // 2nd Workshop on Mutation Analysis. Raleigh NC. 2006. P. 83–92.

6. Mutation of Java objects / R. T. Alexander, J. M. Bieman, S. Ghosh, J. Bixia // 13th Intern. Symp. on Software Reliability Engineering. Fort Collins CO, USA, 2002. P. 341–351.

7. Untch R. Mutation-based software testing using program schemata // Proc. of the 30th ACM Southeast Regional Conf. Raleigh NC, April, 1992. P. 285–291.

8. Untch R., Offutt A. J., Harrold M. J. Mutation analysis using program schemata // Proc. of the Intern. Symp. on Software Testing, and Analysis. Cambridge MA, June, 1993. P. 139–148.

9. Информационно-управляющие человекомашинные системы. Исследование, проектирование, испытания: справ. / под общ. ред. А. И. Губинского, В. Г. Евграфова. М.: Машиностроение, 1993.

10. Падерно П. И., Смирнов А. В. Оценка безошибочности выполнения фрагментов алгоритмов при различных видах ошибок // Изв. СПбГЭТУ «ЛЭТИ». 2012. № 2. С. 38–45.

A. V. Smirnov
Saint-Petersburg state electrotechnical university «LETI»

MUTATION TESTING OF SOFTWARE

One of the popular modern methods of software automation testing (mutation testing) is reviewed in this article and performed detail analysis of the possibility to use it as one of major feature during the evaluation of software quality.

Software testing, test-driven development, software, mutation testing, code coverage rate, software quality