УДК 004.031.6

В.В. ЛИТВИНОВ, И.В. БОГДАН

ТЕСТИРОВАНИЕ МОДЕЛЕЙ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Анотація. *У статті вказані причини, що призводять до необхідності проведення тестування моделей об΄єктно-орієнтованих програм, а також описуються найбільш популярні серед існуючих підходи до проведення тестування основних видів UML-діаграм, серед яких діаграма варіантів використання, діаграма класів, діаграма станів та діаграма послідовностей.*
Ключові слова: *тестування, об΄єктно-орієнтована програма, UML-діаграма, модель, коректність, клас.*

Аннотация. *В статье указываются причины, которые приводят к необходимости проведения тестирования моделей объектно-ориентированных программ, а также описываются наиболее популярные из существующих подходы к проведению тестирования основных видов UML-диаграмм, среди которых диаграмма вариантов использования, диаграмма классов, диаграмма состояний и диаграмма последовательностей.*
Ключевые слова: *тестирование, объектно-ориентированная программа, UML-диаграмма, модель, корректность, класс.*

Abstract. *The article presents the causes that lead to the necessity of models testing of object-oriented programs, and describes the most popular among the existing approaches to testing the main types of UML-diagrams such as the use case diagram, the class diagram, the statechart diagram and the sequence diagram.*
Keywords: *testing, object-oriented program, UML-diagram, model, correctness, class.*

1. Введение

Тестирование является одним из важнейших этапов разработки объектно-ориентированного программного обеспечения [1] и представляет собой процесс исследования программного обеспечения с целью получения информации о качестве продукта. Тестирование программного обеспечения проходит в несколько этапов:

1. Тестирование модели программного обеспечения.
2. Тестирование его классов.
3. Проверка взаимодействия компонентов.
4. Системное тестирование.
5. Приемочные испытания.
6. Проверка развертывания системы.

Сразу же после окончательной формулировки технического задания к проекту начинается этап проектирования, то есть создания модели разрабатываемого программного обеспечения. Для этого проводится анализ предметной области, который позволяет учесть ошибки других систем, подобных разрабатываемой, и уточнить сомнительные нюансы разрабатываемой системы, а также анализ приложения, то есть анализ задачи и требований к ее решению. Результатом проектирования является проектная модель системы [2], чаще всего представляемая в виде UML-диаграмм [3], которая и подлежит тестированию. Тестированию подлежат все входящие в состав модели UML-диаграммы, среди которых диаграммы вариантов использования, диаграммы классов, диаграммы последовательности, диаграммы состояний и другие.

Под тестированием проектной модели объектно-ориентированного программного обеспечения подразумевается проверка ее корректности [4], основными характеристиками которой являются последовательность и конечная выполнимость.

© Литвинов В.В., Богдан И.В., 2012

Рассмотрим особенности тестирования основных типов UML-диаграмм.

2. Тестирование диаграммы вариантов использования

При приемочных испытаниях объектно-ориентированного программного обеспечения часто обнаруживают ошибки, источником которых является неправильная формулировка требований [5]. Это одна из типичных проблем, которая обусловлена отсутствием необходимых требований (неполная модель требований), требованиями, которые противоречат друг другу (противоречивая модель), и сценариями, в условиях которых система функционирует не так, как требуется клиенту (некорректная модель).

Многие из этих проблем могут быть выявлены на более ранних, нежели приемочные испытания, стадиях путем тестирования диаграммы вариантов использования. Однако случаи использования, отображенные на диаграмме вариантов использования, не являются представлениями программного обеспечения. Они представляют требования, которым должно соответствовать программное обеспечение. Случаи использования чаще всего формулируются на естественном языке, поэтому и их тестирование практически сводится к проверке синтаксиса.

3. Тестирование диаграммы классов

Тестирование классов выполняется по мере того, как оно становится целесообразным или необходимым. Тестирование классов имеет смысл выполнять в процессе написания программных кодов, когда разработчику нужно узнать, какие детали упущены, и убедиться в правильности некоторой части реализации [2]. Тестирование классов также становится необходимым, когда некоторый компонент программного обеспечения нужно добавить к базовому программному коду. При этом полностью разработка класса может быть не завершена, но поведение, которое он представляет, должно быть правильным и представленным в завершенном виде.

Комплексные испытания обычно планируется проводить через заданные интервалы времени, чаще всего в конце крупных итераций, которые означают завершение очередного приращения, и\или перед выпуском очередной версии разрабатываемого программного обеспечения.

Несмотря на то, что тестирование класса может проводиться на разных этапах его разработки, оно всегда должно выполняться до того, как возникает необходимость использования этого класса в других компонентах программного обеспечения.

Тестирование классов охватывает все виды деятельности, ассоциированные с проверкой реализации класса на точное соответствие спецификации этого класса [6]. Если реализация корректна, то каждый экземпляр этого класса ведет себя подобающим образом. Тестирование классов производится путем построения экземпляров этих классов и тестирования поведения этих экземпляров.

Таким образом, одним из распространенных подходов к тестированию классов является тестирование путем разработки тестового драйвера, который создает экземпляры классов и окружает эти экземпляры соответствующей средой, то есть экземплярами тех классов, с которыми взаимодействует данный класс и которые по умолчанию считаются правильными. Только в таком случае становится возможным прогон соответствующего тестового случая.

Для второго подхода к тестированию классов используется стандарт OCL 2.0, согласно которому семантика диаграммы классов определяется на базе объектной модели – определена объектная модель (множество классов, атрибутов, операций, ассоциаций и т.д), семантика объектной модели определяется традиционным способом при помощи алгебраической системы.

Данный подход носит название метод линейных неравенств и был создан учеными Lenzerini и Nobili (1990), а также Thalheim (1993) [7]. Изначально метод линейных неравенств был определен для диаграммы сущность-связь, которая включает типы сущностей (классов), N-арные типы отношений (объединений) между ними и множество ограничений. Метод заключается в преобразовании кратных ограничений в набор линейных неравенств, для которых переменными являются размер сущности (класса) и возможные типы отношений. Бинарная ассоциация на рис. 1 порождает следующие неравенства:

$$\begin{aligned} & C_1 \succ 0, C_2 \succ 0, r \geq 0, \\ & for\min_1 \neq 0 : r \geq \min_1 * C_1; for\max_1 \neq \infty : r \leq \max_1 * C_1, \\ & for\min_2 \neq 0 : r \geq \min_2 * C_2; for\max_2 \neq \infty : r \leq \max_2 * C_2. \end{aligned} \tag{1}$$

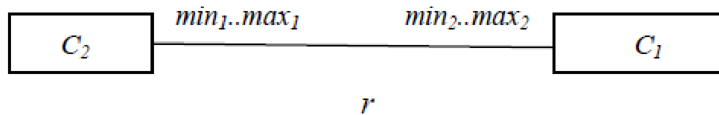


Рис. 1. Бинарная ассоциация

Обоснованием для этих неравенств является то, что для удовлетворения указанных ограничений должно быть не менее $\min_2 * C_1$ и $\min_1 * C_2$ и не более $\max_2 * C_1$ и $\max_1 * C_2$ ссылок в отношениях r, так как каждый объект C_1 связан, по крайней мере, с $\min_1$ и в большинстве случаев с $\max_1$ объектами C_2, и наоборот для C_2. Кроме того, для каждой сущности C или отношения типа R неравенства $C > 0$, $r \geq 0$ присутствуют. Размер системы неравенств является многочленом от размера диаграммы. Основным результатом является то, что диаграмма сущность-связь корректна тогда и только тогда, когда система неравенств имеет решение.

В 1994 году Calvanese и Lenzerini [7] расширили неравенства в методе линейных неравенств для применения к диаграммам классов. В результате была введена переменная для каждого возможного класса между подклассами супер-класса, а отношения были разбиты соответствующим образом. Итак, размер полученной системы неравенств стал экспоненциально зависимым от размера диаграммы классов.

Еще один подход к тестированию классов основан на построении графов и предназначен для выявления противоречивых ограничений. Этот метод был предложен Hartmann (2001), Lenzerini & Nobili (1990) и Thalheim (1993) [7]. Метод основан на построении ориентированного графа, который называется идентификационным графом, узлы которого обозначают классы и ассоциации, а его дуги соединяют ассоциативные узлы с классовыми узлами. Вес дуг соответствует множественным ограничениям, что показано на рис. 2.

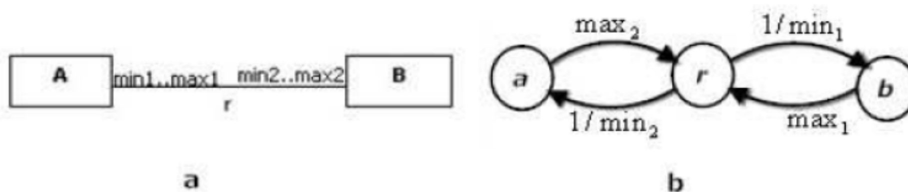
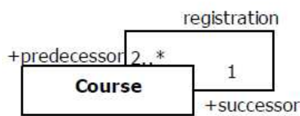


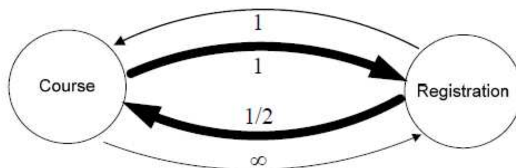
Рис. 2. Идентификационный граф для бинарной ассоциации

Вес пути является произведением весов ребер, которые его составляют.

Идентификационный граф используется для установления корректности диаграммы классов. Циклы, вес которых меньше 1, называются критическими циклами, точками неполной осуществимости. Именно критический цикл выделяет невыполнимый (противоречивый) набор ограничений. На рис. 3b показан идентификационный граф для классов, изображенных на рис. 3a, критический цикл выделяет предварительно последовательные ассоциации как причину конечной невыполнимости.



a



b

Рис. 3. Идентификационный граф с критическими циклами

Совсем иной подход предложила в 2004 году Mira Balaban [7]. Данный подход описывает шаблоны некорректности, которые характеризуют типичные ошибки в проектах. Каждый шаблон описывает проблему корректности, вызванную взаимодействием противоречивых ограничений, которая может быть идентифицирована характерными структурами в пределах диаграммы классов.

Каждый шаблон включает доказательства, описывающие проблему и советы по ее устранению.

Согласно данному подходу, выделяются четыре типа некорректности: противоречивость, отсутствие конечной выполнимости, избыточность и незаконченный проект, которые представляют два различных аспекта корректности.

Первые два типа относятся к проблеме формальной корректности, в то время как последние два – к проблеме качества проекта, которые являются другой формой корректности.

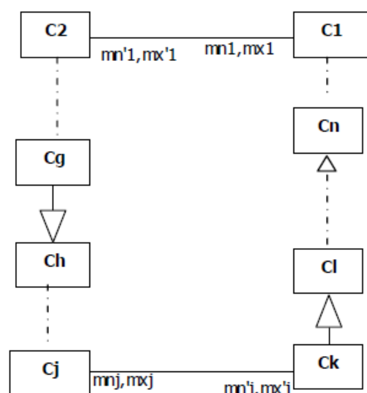
Низкокачественный проект формально корректен, но не соответствует некоторым критериям, требуемым проектом. Такие критерии включают, например, дублирование и недостающие спецификации.

Далее в качестве примера приведен один из наиболее часто используемых шаблонов.

Название шаблона: Цикл множественной иерархии.

Описание шаблона: Цикл ассоциаций с множественными ограничениями и ограничениями иерархии классов может создать проблему конечной удовлетворимости.

Идентификационная структура шаблона. Минимальный цикл ассоциаций и ограничений иерархии классов, в которой все ограничения множественности отличаются от (0,*) и все ограничения иерархии классов расположены в одном направлении.



1. Все классы различны.
2. Все ограничения множественности отличаются от (0,*)

Рис. 4. Пример шаблона: Цикл множественной иерархии

На рис. 4 показана такая идентификационная структура шаблона.

Шаблон описывает возможно чередующиеся цепи ассоциаций и ограничений иерархии классов (в одинаковом направлении). Шаблон может быть точно описан регулярным выражением, которое поддерживает чередование и последовательность.

Во-первых, выражение, которое отображает ограничение иерархии классов, описывается следующим образом: C ‹ D, где C есть подкласс D.

Выражение $[C_{t_{\min_t \max_t}}, C_{t+1_{\min_{t+1} \max_{t+1}}}] + C_t < C_{t+1}$ обозначает ограничение ассоциации или иерархии классов.

Выражение $\{[C_{t_{\min_t \max_t}}, C_{t+1_{\min_{t+1} \max_{t+1}}}] + C_t < C_{t+1}\}^{1..n}$ обозначает последовательность до n чередующихся ограничений ассоциативности или иерархии классов.

Весь шаблон описывается следующим выражением: $\{[C_{t_{\min_t \max_t}}, C_{t+1_{\min_{t+1} \max_{t+1}}}] + C_t < C_{t+1}\}^{1..n-1}, [C_{n_{\min_n \max n}}, C_{1_{\min_{t+n} \max_{t+n}}}]$.

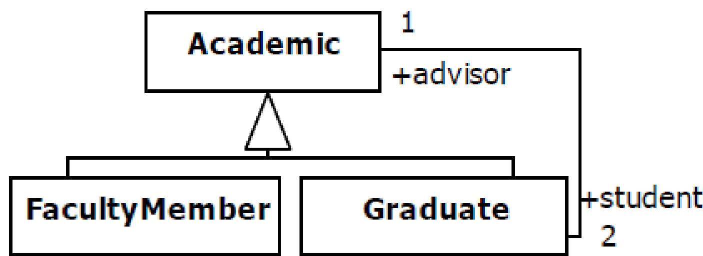


Рис. 5. Отсутствие конечной выполнимости благодаря множественным ограничениям и ограничениям иерархии классов

Конкретный пример: рис. 5 представляет цикл множественных ограничений, который включает класс Graduate и который является подклассом класса Academic и экземпляры которого должны быть связаны с экземплярами класса Academic. Следовательно, предполагая, что G и A – количество Graduates и Academics соответственно, количество связей student-advisor в каждом допустимом экземпляре должно быть G*1 и A*2, подразумевая, что G=A*2. В добавление расширения Graduate и Academic должны удовлетворять G≤A, поскольку Graduate является подклассом класса Academic. Эти ограничения могут быть удовлетворены только при пустых или неопределенных расширениях.

Обоснование шаблона: предыдущий пример показывает, что ассоциативно-классово-иерархический цикл может стать причиной возникновения проблем конечной выполнимости. Остается показать, что минимальность цикла и отсутствие множественного ограничения (0,*) являются необходимыми условиями.

Идентификационный граф конструируется в два шага:

1. Во-первых, каждое ограничение иерархии классов в цикле заменяется ассоциацией с множественными ограничениями (0,1) и (1,1) на подклассовой и суперклассовой сторонах соответственно. Интуитивно видно, что каждый объект подкласса является также объектом суперкласса, но не обязательно наоборот. Maree и Balaban еще в 2007 году показали, что преобразование сохраняет свойства непротиворечивости и конечной выполнимости

2. Замена образует простой ассоциативный цикл, для которого может быть построен идентификационный граф.

Поскольку преобразование иерархии классов в ассоциации не включает ограничение множественности (0,*), то можно сделать вывод, что если исходный ассоциативный цикл минимален и не включает (0,*) ограничение множественности, цикл может вызвать проблему конечной реализуемости, в то время как присутствие (0,*) ограничения множественности гарантирует, что проблемы конечной реализуемости не могут быть вызваны циклом.

Советы по исправлению: различные советы могут быть использованы для переключения направления ограничений иерархии классов в цикле. Дальнейшие действия исправляют конечную выполнимость, как показано в следующем утверждении.

Утверждение: минимальный цикл ассоциаций с множественными ограничениями и ограничениями иерархии классов в обратном направлении не могут создать проблемы конечной реализуемости.

Доказательство: каждое ограничение иерархии классов включает окончание дуги, помеченное ∞ в идентификационном графе, направленной от подкласса к суперклассу. Таким образом, здесь не может быть критических циклов.

4. Тестирование диаграммы состояний

Диаграмма состояний описывает поведение объектов класса в терминах наблюдаемых состояний этого объекта и то, как объект меняет свои состояния в результате событий, оказывающих воздействие на объект.

Каждая диаграмма состояний представляет некоторый автомат, описывающий поведение отдельного объекта в форме последовательности состояний, которые охватывают все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Таким образом, тестирование диаграммы состояний происходит путем проверки следующих условий, вытекающих из теории автоматов и специальной семантики языка UML:

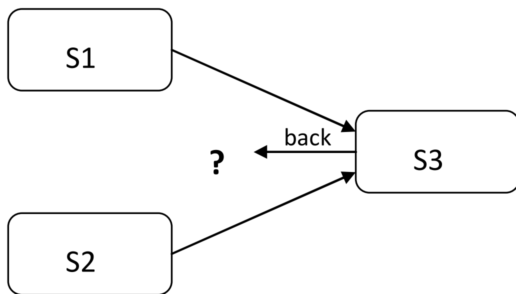


Рис. 6. Свойство диаграммы состояний не запоминать историю перемещения из состояния в состояние

1. Диаграмма состояний, как и автомат, не запоминает историю перемещения из состояния в состояние (рис. 6). С точки зрения моделируемого поведения, определяющим является сам факт нахождения объекта в том или ином состоянии, но никак не последовательность состояний, в результате которой объект перешел в текущее состояние. Другими словами, объект «забывает» все состояния, которые предшествовали текущему в данный момент времени.

2. В каждый момент времени объект на диаграмме состояний может находиться в одном и только в одном из своих состояний (рис. 7). Это означает, что диаграмма состояний предназначена для моделирования последовательного поведения, когда объект в течение своего жизненного цикла последовательно проходит через все свои состояния. При этом объект может находиться в отдельном состоянии как угодно долго, если не происходит никаких событий.

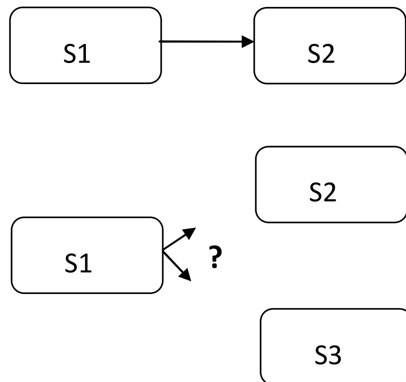
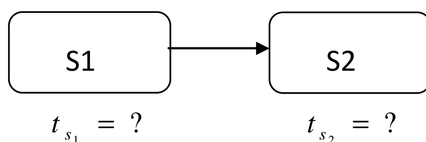


Рис. 7. Свойство объектов диаграммы состояний в каждый момент времени находиться только в одном состоянии

3. Несмотря на то, что процесс изменения состояний объекта происходит во времени, явно концепция времени на диаграмме состояний не отображена (рис. 8). Это означает, что длительность нахождения автомата в том или ином состоянии, а также время достижения того или иного состояния никак не специфицируются. Другими словами, время на диаграмме состояний присутствует в неявном виде, хотя для отдельных событий может быть указан интервал времени и в явном виде.



$t_{s_1} = ?$ $t_{s_2} = ?$

Рис. 8. Отсутствие концепции времени на диаграмме состояний

4. Количество состояний объекта на диаграмме состояний должно быть обязательно конечным (в языке UML рассматриваются только конечные автоматы), и все они должны быть специфицированы явным образом (рис. 9).

При этом отдельные псевдосостояния могут не иметь спецификаций (начальное и конечное состояния). В этом случае их назначение и семантика полностью определяются из контекста модели и рассматриваемой диаграммы состояний.

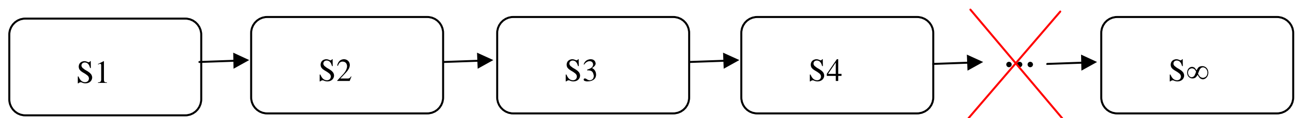


Рис. 9. Свойство диаграммы состояний иметь конечное количество состояний

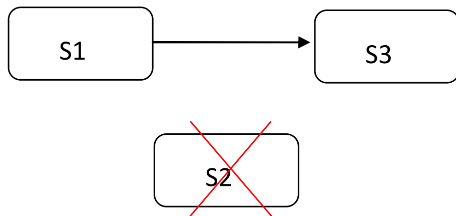


Рис. 10. Свойство диаграммы состояний, состоящее в отсутствии изолированных состояний и переходов

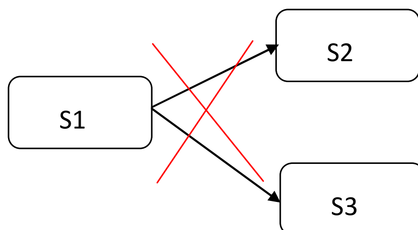


Рис. 11. Свойство диаграммы состояний, состоящее в отсутствии конфликтующих переходов

5. Диаграмма состояний не должна содержать изолированные состояния и переходы (рис. 10). Это условие означает, что для каждого из состояний, за исключением начального, должно быть определено предшествующее состояние. Каждый переход должен обязательно соединять два состояния автомата. Допускается переход из состояния в себя. Такой переход еще называют «петлей».

6. Диаграмма состояний не должна содержать конфликтующих переходов (рис. 11), т.е. таких переходов из одного и того же состояния, когда объект одновременно может перейти в два и более последующих состояния (кроме случая параллельных подавтоматов). В языке UML исключение конфликтов возможно на основе введения так называемых сторожевых условий (триггерных переходов).

5. Тестирование диаграммы последовательностей

Диаграмма последовательностей отображает обмен сообщениями между объектами, создание объектов и ответы на сообщения. Диаграммы последовательностей используются для описания того, какие объекты участвуют в обработке данных с целью нахождения того или иного аспекта решения и как эти объекты взаимодействуют между собой с целью оказания влияния на эту обработку.

Основное назначение тестирования диаграмм последовательностей состоит в том, чтобы убедиться, что происходит правильный обмен сообщениями между объектами, классы которых уже прошли тестирование ранее.

Так как некоторые объекты на диаграмме последовательности будут существовать постоянно, а некоторые временно (только в период выполнения ими требуемых действий), то, прежде всего, следует убедиться в том, что для уничтожения объектов, которые создаются на время выполнения своих действий, предусмотрены явные сообщения. Для этого используются коллекции объектов, представляющие собой наборы объектов, каждый из которых использует в своей спецификации объекты, но фактически не взаимодействует с ними, а вместо этого выполняет одно или несколько из следующих действий [2]:

– сохраняет ссылки на такие объекты, которые обычно представляют отношения один-ко-многим между объектами программы;

– создает экземпляры этих объектов;

– удаляет экземпляры этих объектов.

Тестовый драйвер создает экземпляры, которые передаются как параметры сообщений в тестируемые коллекции. Задача тестовых случаев заключается главным образом в том, чтобы убедиться, что эти экземпляры успешно включаются в коллекцию и удаляются из нее. При этом точный класс каждого из объектов, используемых в процессе тестирования, не имеет значения при проверке корректности функционирования коллекций, поскольку взаимодействие экземпляра коллекции с объектами, содержащимися в коллекции, отсутствует.

Далее следует выполнять непосредственное тестирование взаимодействия между объектами. Основное внимание во время тестирования взаимодействий в условиях контактного подхода уделяется проверке, выполнены ли объектом-отправителем предусловия методов получающего объекта. Не допускается построение тестовых случаев, нарушающих эти предусловия. Обычно практикуется перевод объекта-получателя в некоторое заданное состояние. После чего инициируется выполнение тестового драйвера, по условиям которого объект-отправитель требует, чтобы объект-получатель находился в другом состоянии. Смысл подобной проверки заключается в том, чтоб установить, выполняет ли объект-отправитель проверку предусловий объекта-получателя, прежде чем отправлять заранее неприемлемое сообщение. Одновременно с этим проверяется, корректно ли объект-отправитель прекращает свою деятельность, возможно, генерируя при этом исключение.

Тестирование диаграмм последовательности также можно выполнять путем тестирования протоколов. По мере того, как некоторый объект вступает во взаимодействие с другими объектами, увеличивается количество сообщений, которые он получает. Все эти сообщения упорядочиваются в определенную последовательность. Тестирование по протоколу проверяет реализацию на соответствие этой последовательности. Различные протоколы, в которых принимает участие тот или иной объект, могут быть логически выведены из пред- и постусловий, регламентирующих выполнение отдельных операций, объявленных в классе этого объекта. Идентифицирующая последовательность вызовов методов, в которую объединяются методы, чьи постусловия удовлетворяют предусловиям следующего метода, образуют протокол. Такие последовательности намного легче обнаружить на диаграмме состояний конкретного класса, нежели искать их, сопоставляя письменные формулировки пред- и постусловий. При данном подходе тестовый набор, предназначенный для тестирования взаимодействий, содержит тесты каждого протокола.

6. Согласованность UML-диаграмм

После завершения тестирования проектной модели объектно-ориентированного программного обеспечения составляющие ее диаграммы необходимо проверить на согласованность. Существуют следующие уровни согласованности:

– горизонтальная согласованность – согласованность в середине модели – между предварительно оттестированными диаграммами;

– эволюционная согласованность – непротиворечивость между различными версиями одной модели;

– вертикальная согласованность – непротиворечивость между разными моделями.

7. Выводы

Самым первым и одним из важнейших этапов тестирования объектно-ориентированного программного обеспечения является тестирование его проектной модели. На данном этапе происходит тестирование всех UML-диаграмм, которые составляют модель. К таким диаграммам чаще всего принадлежат основные виды UML-диаграмм, среди которых диаграмма вариантов использования, диаграмма классов, диаграмма состояний и диаграмма

последовательностей. В статье описаны наиболее популярные и эффективные из существующих методов выполнения тестирования основных видов UML-диаграмм. После тестирования каждая диаграмма проходит проверку на корректность. Только прошедшую проверку на корректность и согласованность проектную модель объектно-ориентированного программного обеспечения можно использовать для эффективной разработки программного обеспечения.

СПИСОК ЛИТЕРАТУРЫ

1. Тамре Л. Введение в тестирование программного обеспечения / Тамре Л.; пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 368 с.
2. Макгрегор Дж. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Дж. Макгрегор, Д. Сайкс; пер. с англ. – К.: ООО «ТИД «ДС»», 2002. – 432 с.
3. Леоненков А. Самоучитель UML. Эффективный инструмент моделирования информационных систем / Леоненков А. – Спб.: BHV-Санкт-Петербург, 2001. – 304 с.
4. Об'єктно-орієнтоване моделювання при проектуванні вбудованих систем і систем реального часу: навч. пос. з дисципліни: «Системний аналіз та проектування комп'ютерних інформаційних систем» / В.В. Литвинов, С.В. Голуб, К.М. Григор'єв [та ін.]. – Черкаси: Черкаський національний університет ім. Б. Хмельницького, 2010. – 379 с.
5. Page-Jones M. Fundamentals of Object-Oriented Design in UML / Page-Jones M. – New York: Addison-Wesley, 2000. – 435 p.
6. Zhou X. Auto-generation of Class Diagram from Free-text Functional Specificftions and Domain Ontology / X. Zhou, N. Zhou. – 2004. – 20 p.
7. Balaban M. Management of Correctness Problems in UML Class Diagrams – Towards a Pattern-based Approach / Balaban M., Maraee A., Stur A. – Beer Sheva: Department of Computer Science, Ben-Gurion University of the Negev, 2002. – 33 p.

Стаття надійшла до редакції 16.03.2012