Необходимость и важность тестирования ПО трудно переоценить. Вместе с тем следует отметить, что тестирование является сложной и трудоемкой деятельностью. Об этом свидетельствует содержание глав 6, 7 и 8 глав, в которых описывались классические основы тестирования, разработанные в расчете (в основном) на процедурное ПО. В этой главе рассматриваются вопросы объектно-ориентированного тестирования [17], [18], [42], [50], [51]. Существует мнение, что объектно-ориентированное тестирование мало чем отличается от процедурно-ориентированного тестирования. Конечно, многие понятия, подходы и способы тестирования у них общие, но в целом это мнение ошибочно. Напротив, особенности объектно-ориентированных систем должны вносить и вносят существенные изменения как в последовательность этапов, так и в содержание этапов тестирования. Сгруппируем эти изменения по трем направлениям:

* расширение области применения тестирования;
* изменение методики тестирования;
* учет особенностей объектно-ориентированного ПО при проектировании тестовых вариантов.

Обсудим каждое из выделенных направлений отдельно.

**Расширение области применения объектно-ориентированного тестирования**

Разработка объектно-ориентированного ПО начинается с создания визуальных моделей, отражающих статические и динамические характеристики будущей системы. Вначале эти модели фиксируют исходные требования заказчика, затем формализуют реализацию этих требований путем выделения объектов, которые взаимодействуют друг с другом посредством передачи сообщений. Исследование моделей взаимодействия приводит к построению моделей классов и их отношений, составляющих основу логического представления системы. При переходе к физическому представлению строятся модели компоновки и размещения системы.

На конструирование моделей приходится большая часть затрат объектно-ориентированного процесса разработки. Если к этому добавить, что цена устранения ошибки стремительно растет с каждой итерацией разработки, то совершенно логично требование тестировать объектно-ориентированные модели анализа и проектирования.

Критерии тестирования моделей: правильность, полнота, согласованность [18], [51].

О синтаксической правильности судят по правильности использования языка моделирования (например, UML). О семантической правильности судят по соответствию модели реальным проблемам. Для определения того, отражает ли модель реальный мир, она оценивается экспертами, имеющими знания и опыт в конкретной проблемной области. Эксперты анализируют содержание классов, наследование классов, выявляя пропуски и неоднозначности. Проверяется соответствие отношений классов реалиям физического мира.

О согласованности судят путем рассмотрения противоречий между элементами в модели. Несогласованная модель имеет в одной части представления, которые противоречат представлениям в других частях модели.

Для оценки согласованности нужно исследовать каждый класс и его взаимодействия с другими классами. Для упрощения такого исследования удобно использовать модель Класс— Обязанность— Сотрудничество CRC (Class— Responsibility — Collaboration). Основной элемент этой модели — CRC-карта [9], [76]. Кстати, CRC-карты — любимый инструмент проектирования в ХР-процессах.

По сути, CRC-карта — это расчерченная карточка размером 6 на 10 сантиметров. Она помогает установить задачи класса и выявить его окружение (классы-собеседники). Для каждого класса создается отдельная карта.

В каждой CRC-карте указывается имя класса, его обязанности (операции) и его сотрудничества (другие классы, в которые он посылает сообщения и от которых он зависит при выполнении своих обязанностей). Сотрудничества подразумевают наличие ассоциаций и зависимостей между классами. Они фиксируются в других моделях — диаграмме сотрудничества (последовательности) объектов и диаграмме классов.

CRC-карта намеренно сделана простой, даже ее ограниченный размер имеет определенный смысл: если список обязанностей и сотрудников не помещается на карте, то, наверное, данный класс надо разделить на несколько классов.

Для оценки модели (диаграммы) классов на основе CRC-карт рекомендуются следующие шаги [50].

1. Выполняется перекрестный просмотр CRC-карты и диаграммы сотрудничества (последовательности) объектов. Цель — проверить наличие сотрудников, согласованность информации в обеих моделях.
2. Исследуются обязанности CRC-карты. Цель — определить, предусмотрена ли в карте сотрудника обязанность, которая делегируется ему из данной карты. Например, в табл. 16.1 показана CRC-карта Банкомат. Для этой карты выясняем, выполняется ли обязанность Читать карту клиента, которая требует использования сотрудника Карта клиента. Это означает, что класс Карта клиента должен иметь операцию, которая позволяет ему быть прочитанным.

**Таблица 16.1.**CRC-карта Банкомат

|  |  |
| --- | --- |
| **Имя класса: Банкомат**  **Обязанности:** | **Сотрудники:** |
| Читать карту клиента  Идентификация клиента  Проверка счета  Выдача денег  Выдача квитанции  Захват карты | Карта клиента  База данных клиентов  База данных счетов  Блок денег  Блок квитанций  Блок карт |

1. Организуется проход по каждому соединению CRC-карты. Проверяется корректность запросов, выполняемых через соединения. Такая проверка гарантирует, что каждый сотрудник, предоставляющий услугу, получает обоснованный запрос. Например, если произошла ошибка и класс База данных клиентов получает от класса Банкомат запрос на Состояние счета, он не сможет его выполнить — ведь База данных клиентов не знает состояния их счетов.
2. Определяется, требуются ли другие классы, или правильно ли распределены обязанности по классам. Для этого используют проходы по соединениям, исследованные на шаге 3.
3. Определяется, нужно ли объединять часто запрашиваемые обязанности. Например, в любой ситуации используют пару обязанностей — Читать карту клиента и Идентификация клиента. Их можно объединить в новую обязанность Проверка клиента, которая подразумевает как чтение его карты, так и идентификацию клиента.
4. Шаги 1-5 применяются итеративно, к каждому классу и на каждом шаге эволюции объектно-ориентированной модели.

**Изменение методики при объектно-ориентированном тестировании**

В классической методике тестирования действия начинаются с тестирования элементов, а заканчиваются тестированием системы. Вначале тестируют модули, затем тестируют интеграцию модулей, проверяют правильность реализации требований, после чего тестируют взаимодействие всех блоков компьютерной системы.

**Особенности тестирования объектно-ориентированных «модулей»**

При рассмотрении объектно-ориентированного ПО меняется понятие модуля. Наименьшим тестируемым элементом теперь является класс (объект). Класс содержит несколько операций и свойств. Поэтому сильно изменяется содержание тестирования модулей.

В данном случае нельзя тестировать отдельную операцию изолированно, как это принято в стандартном подходе к тестированию модулей. Любую операцию приходится рассматривать как часть класса.

Например, представим иерархию классов, в которой операция ОР() определена для суперкласса и наследуется несколькими подклассами. Каждый подкласс использует операцию ОР(), но она применяется в контексте его приватных свойств и операций. Этот контекст меняется, поэтому операцию ОР() надо тестировать в контексте каждого подкласса. Таким образом, изолированное тестирование операции ОР(), являющееся традиционным подходом в тестировании модулей, не имеет смысла в объектно-ориентированной среде.

**Выводы:**

* тестированию модулей традиционного ПО соответствует тестирование классов объектно-ориентированного ПО;
* тестирование традиционных модулей ориентировано на поток управления внутри модуля и поток данных через интерфейс модуля;
* тестирование классов ориентировано на операции, инкапсулированные в классе, и состояния в пространстве поведения класса.

**Тестирование объектно-ориентированной интеграции**

Объектно-ориентированное ПО не имеет иерархической управляющей структуры, поэтому здесь неприменимы методики как восходящей, так и нисходящей интеграции. Мало того, классический прием интеграции (добавление по одной операции в класс) зачастую неосуществим.

Р. Байндер предлагает две методики интеграции объектно-ориентированных систем [16]:

* тестирование, основанное на потоках;
* тестирование, основанное на использовании.

В первой методике объектом интеграции является набор классов, обслуживающий единичный ввод данных в систему. Иными словами, средства обслуживания каждого потока интегрируются и тестируются отдельно. Для проверки отсутствия побочных эффектов применяют регрессионное тестирование. По второй методике вначале интегрируются и тестируются независимые классы. Далее работают с первым слоем зависимых классов (которые используют независимые классы), со вторым слоем и т. д. В отличие от стандартной интеграции, везде, где возможно, избегают драйверов и заглушек.

Д. МакГрегор считает, что одним из шагов объектно-ориентированного тестирования интеграции должно быть кластерное тестирование [50]. Кластер сотрудничающих классов определяется исследованием CRC-модели или диаграммы сотрудничества объектов. Тестовые варианты для кластера ориентированы на обнаружение ошибок сотрудничества.

**Объектно-ориентированное тестирование правильности**

При проверке правильности исчезают подробности отношений классов. Как и традиционное подтверждение правильности, подтверждение правильности объектно-ориентированного ПО ориентировано на видимые действия пользователя и распознаваемые пользователем выводы из системы.

Для упрощения разработки тестов используются элементы Use Case, являющиеся частью модели требований. Каждый элемент Use Case задает сценарий, который позволяет обнаруживать ошибки во взаимодействии пользователя с системой.

Для подтверждения правильности может проводиться обычное тестирование «черного ящика».

Полезную для формирования тестов правильности информацию содержат диаграммы взаимодействия, диаграммы деятельности, а также диаграммы схем состояний.

**Проектирование объектно-ориентированных тестовых вариантов**

Традиционные тестовые варианты ориентированы на проверку последовательности: ввод исходных данных — обработка — вывод результатов — или на проверку внутренней управляющей (информационной) структуры отдельных модулей. Объектно-ориентированные тестовые варианты проверяют состояния классов. Получение информации о состоянии затрудняют такие объектно-ориентированные характеристики, как инкапсуляция, полиморфизм и наследование.

**Инкапсуляция**

Информацию о состоянии класса можно получить только с помощью встроенных в него операций, которые возвращают значения свойств класса.

**Полиморфизм**

При вызове полиморфной операции трудно определить, какая реализация будет проверяться. Пусть нужно проверить вызов функции

y=functionA(x).

В стандартном ПО достаточно рассмотреть одну реализацию поведения, которая обеспечивает вычисление функции. В объектно-ориентированном ПО придется рассматривать поведение реализации Базовый\_класс :: functionA(х), Производный\_клacc :: functionA(x), Наследиик\_Производного\_класса :: functionA(х). Здесь двойным двоеточием от имени операции отделяется имя класса, в котором размещена операция (это обозначение UML). Таким образом, в объектно-ориентированном контексте каждый раз при вызове functionA(x) следует рассматривать набор различных поведений. Конечно, подход к тестированию базовых и производных классов в основном одинаков. Разница состоит только в учете используемых системных ресурсов.

**Наследование**

Наследование также может усложнить проектирование тестовых вариантов. Пусть Родительский\_класс содержит операции унаследована() и переопределена(). Дочерний\_класс переопределяет операцию переопределена() по-своему. Очевидно, что реализация Дочерний\_класс::переопределена() должна повторно тестироваться, ведь ее содержание изменилось. Но надо ли повторно тестировать операцию Дочерний\_класс::унаследована()?

Рассмотрим следующий случай. Положим, что операция Дочерний\_класс::унаследована() вызывает операцию переопределен(). К чему это приводит? Поскольку реализация операции переопределен() изменена, операция Дочерний\_класс::унаследована() может не соответствовать этой новой реализации. Поэтому нужны новые тесты, хотя содержание операции унаследована() не изменено.

Важно отметить, что для операции Дочерний\_класс::унаследована() может проводиться только подмножество тестов. Если часть операции унаследована() не зависит от операции переопределен(), то есть нет ее вызова или вызова любого кода, который косвенно ее вызывает, то ее не надо повторно тестировать в дочернем классе.

Родительский\_класс::переопределена() и Дочерний\_класс::переопределена() —это две разные операции с различными спецификациями и реализациями. Каждая из них проверяется самостоятельным набором тестов. Эти тесты нацелены на вероятные ошибки: ошибки интеграции, ошибки условий, граничные ошибки и т. д. Однако сами операции, как правило, похожи. Наборы их тестов будут перекрываться. Чем лучше качество объектно-ориентированного проектирования, тем больше перекрытие. Таким образом, новые тесты надо формировать только для тех требований к операции Дочерний\_класс::переопределена(), которые не покрываются тестами для операции Родительский\_класс::переопределена().

**Выводы:**

* тесты для операции Родительский\_класс::переопределена() частично применимы к операции Дочерний\_класс::переопределена();
* входные данные тестов подходят к операциям обоих классов;
* ожидаемые результаты для операции Родительского\_класса отличаются от ожидаемых результатов для операции Дочернего\_класса.

К операциям класса применимы классические способы тестирования «белого ящика», которые гарантируют проверку каждого оператора и их управляющих связей. При большом количестве операций от тестирования по принципу «белого ящика» приходится отказываться. Меньших затрат потребует тестирование на уровне классов.

Способы тестирования «черного ящика» также применимы к объектно-ориентированным системам. Полезную входную информацию для тестирования «черного ящика» и тестирования состояний обеспечивают элементы Use Case.

**Тестирование, основанное на ошибках**

Цель тестирования, основанного на ошибках, — проектирование тестов, ориентированных на обнаружение предполагаемых ошибок [46]. Разработчик выдвигает гипотезу о предполагаемых ошибках. Для проверки его предположений разрабатываются тестовые варианты.

В качестве примера рассмотрим булево выражение

if (*X*and not *Y*or *Z*).

Инженер по тестированию строит гипотезу о предполагаемых ошибках выражения:

* операция not сдвинулась влево от нужного места (она должна применяться к Z);
* вместо and должно быть or;
* вокруг not Y or Z должны быть круглые скобки.

Для каждой предполагаемой ошибки проектируются тестовые варианты, которые заставляют некорректное выражение отказать.

Обсудим первую предполагаемую ошибку. Значения *(X*= false, *Y* = false, *Z* = false) устанавливают указанное выражение в false. Если вместо not Y должно быть not Z,то выражение принимает неправильное значение, которое приводит к неправильному ветвлению. Аналогичные рассуждения применяются к генерации тестов по двум другим ошибкам.

Эффективность этой методики зависит от того, насколько правильно инженер выделяет предполагаемую ошибку. Если действительные ошибки объектно-ориентированной системы не воспринимаются как предполагаемые, то этот подход не лучше стохастического тестирования. Если же визуальные модели анализа и проектирования обеспечивают понимание ошибочных действий, то тестирование, основанное на ошибках, поможет найти подавляющее большинство ошибок при малых затратах.

При тестировании интеграции предполагаемые ошибки ищутся в пересылаемых сообщениях. В этом случае рассматривают три типа ошибок: неожиданный результат, вызов не той операции, некорректный вызов. Для определения предполагаемых ошибок (при вызове операций) нужно исследовать процесс выполнения операции.

Тестирование интеграции применяется как к свойствам, так и к операциям. Поведение объекта определяется значениями, которые получают его свойства. Поэтому проверка значений свойств обеспечивает проверку правильности поведения.

При тестировании интеграции ищутся ошибки в объектах-клиентах, запрашивающих услуги, а не в объектах-серверах, предоставляющих эти услуги. Тестирование интеграции определяет ошибки в вызывающем коде, а не в вызываемом коде. Вызов операции используют как средство, обеспечивающее тестирование вызывающего кода.

**Тестирование, основанное на сценариях**

Тестирование, основанное на ошибках, оставляет в стороне два важных типа ошибок:

* некорректные спецификации;
* взаимодействия между подсистемами.

Ошибка из-за неправильной спецификации означает, что продукт не выполняет то, чего хочет заказчик. Он делает что-то неправильно, или в нем пропущена важная функциональная возможность. В любом случае страдает качество — соответствие требованиям заказчика.

Ошибки, связанные с взаимодействием подсистем, происходят, когда поведение одной подсистемы создает предпосылки для отказа другой подсистемы.

Тестирование, основанное на сценариях, ориентировано на действия пользователя, а не на действия программной системы [47]. Это означает фиксацию задач, которые выполняет пользователь, а затем применение их в качестве тестовых вариантов. Задачи пользователя фиксируются с помощью элементов Use Case.

Сценарии элементов Use Case обнаруживают ошибки взаимодействия, каждая из которых может быть следствием многих причин. Поэтому соответствующие тестовые варианты более сложны и лучше отражают реальные проблемы, чем тесты, основанные на ошибках. С помощью одного теста, основанного на сценарии, проверяется множество подсистем.

Рассмотрим, например, проектирование тестов, основанных на сценариях, для текстового редактора.

Рабочие сценарии опишем *в*виде спецификаций элементов Use Case.

Элемент Use Case: Исправлять черновик.

*Предпосылки:*обычно печатают черновик, читают его и обнаруживают ошибки, которые не видны на экране. Данный элемент Use Case описывает события, которые при этом происходят.

1. Печатать весь документ.

2. Прочитать документ, изменить определенные страницы.

3. После внесения изменения страница перепечатывается.

4. Иногда перепечатываются несколько страниц.

Этот сценарий определяет как требования тестов, так и требования пользователя. Требования пользователя очевидны, ему нужны:

* метод для печати отдельной страницы;
* метод для печати диапазона страниц.

В ходе тестирования проверяется редакция текста как до печати, так и после печати. Разработчик теста может надеяться обнаружить, что функция печати вызывает ошибки в функции редактирования. Это будет означать, что в действительности две программные функции зависят друг от друга.

Элемент Use Case: Печатать новую копию.

*Предпосылки:*кто-то просит пользователя напечатать копию документа.

1. Открыть документ.

2. Напечатать документ.

3. Закрыть документ.

И в этом случае подход к тестированию почти очевиден. За исключением того, что не определено, откуда появился документ. Он был создан в ранней задаче. Означает ли это, что только эта задача влияет на сценарий?

Во многих современных редакторах запоминаются данные о последней печати документа. По умолчанию эту печать можно повторить. После сценария Исправлять черновик достаточно выбрать в меню Печать, а затем нажать кнопку Печать в диалоговом окне — в результате повторяется печать последней исправленной страницы. Таким образом, откорректированный сценарий примет вид:

Элемент Use Case: Печатать новую копию.

1. Открыть документ.
2. Выбрать в меню пункт Печать.
3. Проверить, что печаталось, и если печатался диапазон страниц, то выбрать опцию Печатать целый документ.
4. Нажать кнопку Печать.
5. Закрыть документ.

Этот сценарий указывает возможную ошибку спецификации: редактор не делает того, что пользователь ожидает от него. Заказчики часто забывают о проверке, предусмотренной шагом 3. Они раздражаются, когда рысью бегут к принтеру и находят одну страницу вместо ожидаемых 100 страниц. Раздраженные заказчики считают, что в спецификации допущена ошибка.

Разработчик может опустить эту зависимость в тестовом варианте, но, вероятно, проблема обнаружится в ходе тестирования. И тогда разработчик будет выкрикивать: «Я предполагал, я предполагал это учесть!!!».

**Тестирование поверхностной и глубинной структуры**

Поверхностная структура — это видимая извне структура объектно-ориентированной системы. Она отражает взгляд пользователя, который видит не функции, а объекты для обработки. Тестирование поверхностной структуры основывается на задачах пользователя. Главное — выяснить задачи пользователя. Для разработчика это нетривиальная проблема, поскольку требует отказа от своей точки зрения.

Глубинная структура отражает внутренние технические подробности объектно-ориентированной системы (на уровне проектных моделей и программного текста). Тесты глубинной структуры исследуют зависимости, поведение и механизмы взаимодействия, которые создаются в ходе проектирования подсистем и объектов.

В качестве базиса для тестирования глубинной структуры используются модели анализа и проектирования. Например, разработчик исследует диаграмму взаимодействия (невидимую извне) и спрашивает: «Проверяет ли тест сотрудничество, отмеченное на диаграмме?»

Диаграммы классов обеспечивают понимание структуры наследования, которая используется в тестах, основанных на ошибках. Рассмотрим операцию ОБРАБОТАТЬ (Ссылка\_на\_РодительскийКласс). Что произойдет, если в вызове этой операции указать ссылку на дочерний класс? Есть ли различия в поведении, которые должны отражаться в операции ОБРАБОТАТЬ? Эти вопросы инициируют создание конкретных тестов.

**Способы тестирования содержания класса**

Описываемые ниже способы ориентированы на отдельный класс и операции, которые инкапсулированы классом.

**Стохастическое тестирование класса**

При стохастическом тестировании исходные данные для тестовых вариантов генерируются случайным образом. Обсудим методику, предложенную С. Кирани и В.Тсай[43].

Рассмотрим класс Счет, который имеет следующие операции: Открыть, Установить, Положить, Снять, Остаток, Итог, ОграничитьКредит, Закрыть.

Каждая из этих операций применяется при определенных ограничениях:

* счет должен быть открыт перед применением других операций;
* счет должен быть закрыт после завершения всех операций.

Даже с этими ограничениями существует множество допустимых перестановок операций. Минимальная работа экземпляра Счета включает следующую последовательность операций:

Открыть ► Установить ►Положить ► Снять ► Закрыть.

Здесь стрелка обозначает операцию следования. Иначе говоря, здесь записано, что экземпляр Счета сначала выполняет операцию открытия, затем установки и т. д. Эта последовательность является минимальным тестовым вариантом для Счета. Впрочем, в эту последовательность можно встроить группировку, обеспечивающую создание других вариантов поведения:

Открыть ► Установить ► Положить ► [Остаток●Снять●Итог●ОграничитьКредит●Положить]n ►Снять ► Закрыть.

Здесь приняты дополнительные обозначения: точка означает операцию И/ИЛИ, пара квадратных скобок — группировку, а показатель степени — количество повторений группировки.

Набор различных последовательностей может генерироваться случайным образом:

*Тестовый вариант N:*

Открыть ►Установить ►Положить ►Остаток ► Снять ►Итог ►Снять ►Закрыть.

*Тестовый вариант М:*

Открыть ►Установить ► Положить ►Итог ►ОграничитьКредит ►Снять ►Остаток ►Снять ►Закрыть.

Эти и другие тесты случайных последовательностей проводятся для проверки различных вариантов жизни объектов.

**Тестирование разбиений на уровне классов**

Тестирование разбиений уменьшает количество тестовых вариантов, требуемых для проверки классов (тем же способом, что и разбиение по эквивалентности для стандартного ПО). Области ввода и вывода разбивают на категории, а тестовые Варианты разрабатываются для проверки каждой категории.

Обычно используют одну из трех категории разбиения [43]. Категории образуются операциями класса.

Первый способ — *разбиение на категории по состояниям.*Основывается на способности операций изменять состояние класса. Обратимся к классу Счет. Операции Снять, Положить изменяют его состояние и образуют первую категорию. Операции Остаток, Итог, ОграничитьКредит не меняют состояние Счета и образуют вторую категорию. Проектируемые тесты отдельно проверяют операции, которые изменяют состояние, а также те операции, которые не изменяют состояние. Таким образом, для нашего примера:

*Тестовый вариант 1:*

Открыть ►Установить ►Положить *►*Положить *►*Снять ►Снять ►Закрыть.

*Тестовый вариант 2:*

Открыть *►*Установить ►Положить *►*Остаток ►Итог ►ОграничитьКредит *►*Снять *►*Закрыть.

ТВ1 изменяет состояние объекта, в то время как ТВ2 проверяет операции, которые не меняют состояние. Правда, в ТВ2 пришлось включить операции минимальной тестовой последовательности, поэтому для нейтрализации влияния операций Снять и Положить их аргументы должны иметь одинаковые значения.

Второй способ —*разбиение на категории по свойствам.*Основывается на свойствах, которые используются операциями. В классе Счет для определения разбиений можно использовать свойства остаток и ограничение кредита. Например, на основе свойства ограничение кредита операции подразделяются на три категории:

1) операции, которые используют ограничение кредита;

2) операции, которые изменяют ограничение кредита;

3) операции, которые не используют и не изменяют ограничение кредита.

Для каждой категории создается тестовая последовательность.

Третий способ — *разбиение на категории по функциональности.*Основывается на общности функций, которые выполняют операции. Например, операции в классе Счет могут быть разбиты на категории:

* операции инициализации (Открыть, Установить);
* вычислительные операции (Положить, Снять);
* запросы (Остаток, Итог, ОграничитьКредит);
* операции завершения (Закрыть).

**Способы тестирования взаимодействия классов**

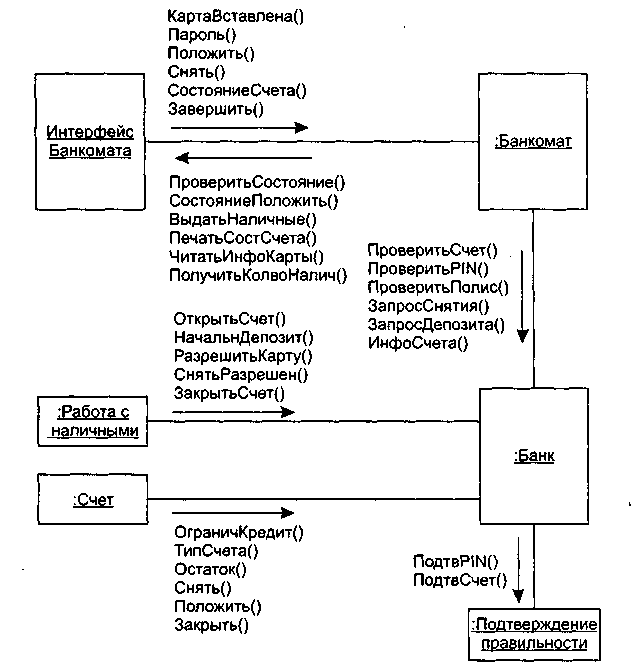
Для тестирования сотрудничества классов могут использоваться различные способы [43]:

* стохастическое тестирование;
* тестирование разбиений;
* тестирование на основе сценариев;
* тестирование на основе состояний.

В качестве примера рассмотрим программную модель банковской системы, в состав которой входят классы Банк, Банкомат, ИнтерфейсБанкомата, Счет, Работа с наличными, ПодтверждениеПравильности, имеющие следующие операции:

|  |  |  |
| --- | --- | --- |
| Банк: |  |  |
| ПроверитьСчет( ); | ЗапросДепозита ( ); | РазрешитьКарту( ); |
| ПроверитьРIN( ); | ИнфоСчета( ); | СнятьРазрешен( ); |
| ПроверитьПолис( ); | ОткрытьСчет( ); | ЗакрытьСчет( ). |
| ЗапросСнятия( ); | НачальнДепозит( ); |  |
| Банкомат: |  |  |
| КартаВставлена( ); | Положить( ); | СостояниеСчета( ); |
| Пароль( ); | Снять( ); | Завершить( ). |
| ИнтерфейсБанкомата: |  |  |
| ПроверитьСостояние( ); | ВыдатьНаличные( ); | ЧитатьИнфоКарты( ); |
| СостояниеПоложить( ); | ПечатьСостСчета( ); | ПолучитьКолвоНалич( ). |
| Счет: |  |  |
| ОграничКредит( ); | Остаток) ); | Положить( ); |
| ТипСчета( ); | Снять( ); | Закрыть( ). |
| ПодтверждениеПравильности: | | |
| ПодтвРIN( ); | ПодтвСчет( ). |  |

Диаграмма сотрудничества объектов банковской системы представлена на рис. 16.1. На этой диаграмме отображены связи между объектами, стрелки передачи сообщений подписаны именами вызываемых операций.



**Рис. 16.1.**Диаграмма сотрудничества банковской системы

**Стохастическое тестирование**

Стохастические тестовые варианты генерируются следующей последовательностью шагов.

1. Для создания тестов используют списки операций каждого класса-клиента. Операции будут посылать сообщения в классы-серверы.
2. Для каждого созданного сообщения определяется класс-сотрудник и соответствующая операция в классе-сервере.
3. Для каждой операции в классе-сервере, которая вызывается сообщением из класса-клиента, определяются сообщения, которые она, в свою очередь, посылает.
4. Для каждого из сообщений определяется следующий уровень вызываемых операций; они вставляются в тестовую последовательность.

В качестве примера приведем последовательность операций для класса Банк, вызываемых классом Банкомат:

ПроверитьСчет *►*ПроверитьРIN ►[[ПроверитьПолис ►

ЗапросСнятия]●ЗапросДепозита●ИнфоСчета]n.

**ПРИМЕЧАНИЕ**

Здесь приняты следующие обозначения: стрелка означает операцию следования, точка — операцию И/ИЛИ, пара квадратных скобок — группировку операций классов, показатель степени — количество повторений группировки из операций классов.

Случайный тестовый вариант для класса Банк может иметь вид

*Тестовый вариант N:*ПроверитьСчет *►*ПроверитьРШ *►*ЗапросДепозита.

Для выявления сотрудников, включенных в этот тест, рассматриваются сообщения, связанные с каждой операцией, записанной в ТВ *N.*Для выполнения заданий ПроверитьСчет и ПроверитьРТМ Банк должен сотрудничать с классом ПодтверждениеПравильности. Для выполнения задания ЗапросДепозита Банк должен сотрудничать с классом Счет. Отсюда новый ТВ, который проверяет отмеченные сотрудничества:

*Тестовый вариант М:*ПроверитьСчетБанк*►*(ПодтвСчетПодтвПрав) ►ПроверитьРINБанк*►*(ПодтвРШПодтвПрав) ►ЗапросДепозитаБанк ►(ПоложитьСчет).

В этой последовательности операции классов-сотрудников Банка помещены в круглые скобки, индексы отображают принадлежность операций к конкретным классам.

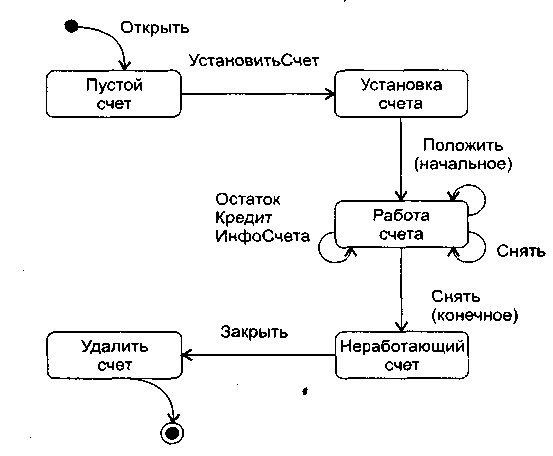
**Тестирование разбиений**

В основу этого метода положен тот же подход, который применялся к отдельному классу. Отличие в том, что тестовая последовательность расширяется для включения тех операций, которые вызываются с помощью сообщений для сотрудничающих классов.

Другой подход к тестированию разбиений основан на взаимодействиях с конкретным классом. Как показано на рис. 16.1, Банк получает сообщения от Банкомата и класса Работа с наличными. Поэтому операции внутри Банка тестируются разбиением их на те, которые обслуживают класс Банкомат, и на те, которые обслуживают класс Работа с наличными. Для дальнейшего уточнения может быть использовано разбиение на категории по состояниям.

**Тестирование на основе состояний**

В качестве источника исходной информации используют диаграммы схем состояний, фиксирующие динамику поведения класса. Данный способ позволяет получить набор тестов, проверяющих поведение класса и тех классов, которые сотрудничают с ним [43]. В качестве примера на рис. 16.2 показана диаграмма схем состояний класса Счет.



**Рис. 16.2.**Диаграмма схем состояний класса Счет

Видим, что объект Счета начинает свою жизнь в состоянии Пустой счет, а заканчивает жизнь в состоянии Удалить счет. Наибольшее количество событий (и действий) связано *с*состоянием Работа счета. Для упрощения рисунка здесь принято, что имена событий совпадают с именами действий (поэтому действия не показаны).

Проектируемые тесты должны обеспечить покрытие всех состояний. Это значит, что тестовые варианты должны инициировать переходы через все состояния объекта:

*Тестовый вариант 1:*Открыть ►УстановитьСчет *►*Положить (начальное) ►Снять (конечное) *►*Закрыть.

Отметим, что эта последовательность аналогична минимальной тестовой последовательности. Добавим к минимальной последовательности дополнительные тестовые варианты:

*Тестовый вариант 2:*Открыть *►*УстановитьСчет *►*Положить (начальное) ►Положить *►*Остаток *►*Кредит ►Снять (конечное) ►Закрыть

*Тестовый вариант 3:*Открыть *►*Установить ►Положить (начальное) ►Положить *►*Снять *►*ИнфоСчета *►*Снять (конечное) ►Закрыть

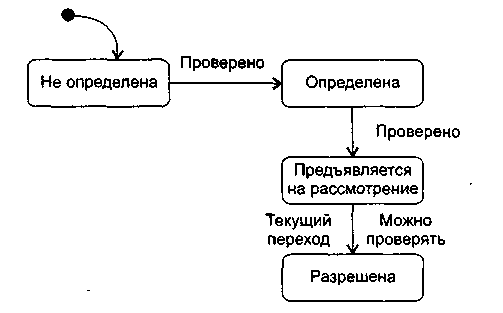
Для гарантии проверки всех вариантов поведения количество тестовых вариантов может быть увеличено. Когда поведение класса определяется в сотрудничестве с несколькими классами, для отслеживания «потока поведения» используют набор диаграмм схем состояний, характеризующих смену состояний других классов.

Возможна другая методика исследования состояний.— *«преимущественно в ширину».*

В этой методике:

* каждый тестовый вариант проверяет один новый переход;
* новый переход можно проверять, если полностью проверены все предшествующие переходы, то есть переходы между предыдущими состояниями.

Рассмотрим объект Карта клиента (рис. 16.3). Начальное состояние карты Не определена, то есть не установлен номер карты. После чтения карты (в ходе диалога с банкоматом) объект переходит в состояние Определена. Это означает, что определены банковские идентификаторы Номер Карты и Дата Истечения Срока. Карта клиента переходит в состояние Предъявляется на рассмотрение, когда проводится ее авторизация, и в состояние Разрешена, когда авторизация подтверждается. Переход карты клиента из одного состояния в другое проверяется отдельным тестовым вариантом.



**Рис. 16.3.**Тестирование «преимущественно в ширину»

Подход «преимущественно в ширину» требует: нельзя проверять Разрешена перед проверкой Не определена, Определена и Предъявляется на рассмотрение. В противном случае нарушается условие этого подхода: перед тестированием текущего перехода должны быть протестированы все переходы, ведущие к нему.

**Предваряющее тестирование при экстремальной разработке**

Предваряющее тестирование и рефакторинг (реорганизация) — основной способ разработки при экстремальном программировании.

Обычно рефакторингом называют внесение в код небольших изменений, сохраняющих функциональность и улучшающих структуру программы. Более широко рефакторинг определяют как технику разработки ПО через множество изменений кода, направленных на добавление функциональности и улучшение структуры.

Предваряющее (test-first) тестирование и рефакторинг — это способ создания и последующего улучшения ПО, при котором сначала пишутся тесты, а затем программируется код, который будет подвергаться этим тестам. Программист выбирает задачу, затем пишет тестовые варианты, которые приводят к отказу программы, так как программа еще не выполняет данную задачу. Далее он модифицирует программу так, чтобы тесты проходили и задача выполнялась. Программист продолжает писать новые тестовые варианты и модифицировать программу (для их выполнения) до тех пор, пока программа не будет исполнять все свои обязанности. После этого программист небольшими шагами улучшает ее структуру (проводит рефакторинг), после каждого из шагов запускает все тесты, чтобы убедиться, что программа по-прежнему работает.

Для демонстрации такого подхода рассмотрим пример конкретной разработки. Будем опираться на технику, описанную Робертом Мартином (с любезного разрешения автора)\*.

\* Robert C. Martin. RUP/XP Guidelines: Test-first Design and Refactoring. - Rational Software White Paper, 2000.

Создадим программу для регистрации посещений кафе-кондитерской. Каждый раз, когда лакомка посещает кафе, вводится количество купленных булочек, их стоимость и текущий вес любителя (любительницы) сладостей. Система отслеживает эти значения и выдает отчеты. Программу будем писать на языке Java.

Для экстремального тестирования удобно использовать среду Junit, авторами которой являются Кент Бек и Эрик Гамма (Kent Beck и Erich Gamma). Прежде всего создадим среду для хранения тестов модулей. Это очень важно для предваряющего тестирования: вначале пишется тестовый вариант, а только потом — код программы. Необходимый код имеет следующий вид:

**Листинг 16.1.**ТестЛакомки. java

import junit.framework.\*;

public class ТестЛакомки extends TestCase

{

public ТестЛакомки (String name)

{

super(name);

}

}

Видно, что при использовании среды Junit класс-контейнер тестовых вариантов должен быть наследником от класса TestCase. Кстати, условимся весь новый код выделять полужирным шрифтом.

Создадим первый тестовый вариант. Одна из целей создаваемой программы — запись количества посещений кафе. Поэтому должен существовать объект ПосещениеКафе, содержащий нужные данные. Следовательно, надо написать тест, создающий этот объект и опрашивающий его свойства. Тесты будем записывать как тестовые функции (их имена должны начинаться с префикса тест). Введем тестовую функцию тестСоздатьПосещениеКафе (листинг 16.2).

**Листинг 16.2.**ТестЛакомки.jауа

import junit.framework.\*;

public class ТестЛакомки extends TestCase

{

public ТестЛакомки (String name)

{

super(name);

}

public void тестСоздатьПосещениеКафе()

{

ПосещениеКафе v = new-ПосещениеКафе();

}

}

Для компиляции этого фрагмента подключим класс ПосещениеКафе.

**Листинг 16.3.**ТестЛакомки.jаvа и ПосещениеКафе.jаvа

ТестЛакомки.jаvа

import junit. framework.\*;

**import ПосещениеКафе;**

public class ТестЛакомки extends TestCase

{

public ТестЛакомки (String name)

{

super(name);

}

public void тестСоздатьПосещениеКафе()

{

ПосещениеКафе v = new ПосещениеКафе();

}

}

ПосещениеКафе.java

**public class ПосещениеКафе**

**{**

**}**

Этот код компилируется, тест проходит, и мы готовы добавить необходимую функциональность.

**Листинг 16.4.**ТестЛакомки.jауа и ПосещениеКафе.jауа

ТестЛакомки.java

import junit.framework.\*;

import ПосещениеКафе;

**import java.util.Date**

public class ТестЛакомки extends TestCase

{

public TecтЛакомки(String name)

{

super(name):

}

public void тестСоздатьПосещениеКафе()

{

**Date дата = new Date();**

**double булочки = 7.0;** // 7 булочек

**double стоимость = 12.5 \* 7;**

// цена 1 булочки - 12.5 руб.

**double вес = 60.0;** // взвешивание лакомки

**double дельта = 0.0001;**// точность

ПосещениеКафе v =

new ПосещениеКафе(**дата, булочки, стоимость, вес**);

**assertEquals(дата, v.получитьДату( ));**

**assertEquals(12.5 \* 7, v.получитьСтоииость(), дельта);**

**assertEquals(7.0, v.получитьБулочки(), дельта);**

**assertEquals(60.0, v.получитьВес(), дельта);**

**assertEquals(12.5, v.получитьЦену(). дельта);**

}

}

ПосещениеКафе.java

**import Java.uti1.Date;**

public class ПосещениеКафе

{

**private Date егоДата;**

**private double егоБулочки;**

**private double егоСтоимость;**

**private double eroBec;**

**public ПосещениеКафе(Date дата, double булочки,**

**double стоимость, double вес)**

**{**

**егоДата = дата;**

**егоБулочки = булочки;**

**егоСтоимость = стоимость;**