Тестирование на основе моделей

В. В. Кулямин

Лекция 8. Основы технологии разработки тестов UniTESK

Разработка тестов и тестирование на основе моделей предполагают, что используемые модели формулируются явно. Одной из технологий такого типа является разработанная в Институте системного программирования РАН в 1995-2002 технология UniTESK.

Назначение технологии UniTESK — разработка и развитие наборов тестов для сложных развивающихся систем. Сложность системы означает достаточно большое количество функций, сложный интерфейс (от нескольких десятков операций), достаточно большой размер кода (от $5 \cdot 10^4$ строк). Развитие системы предполагает, что время от времени появляются ее новые версии, и что набор тестов придется поддерживать в рабочем состоянии и пополнять тестами новых функций в течение значительного периода времени (начиная от 2-х лет, для более чем 2-х версий). Сказанное не означает, что при невыполнении этих условий технологию нельзя применить, просто при этом разработка тестов традиционными методами может быть более эффективной с точки зрения затрат на определенное качество тестов.

Концептуальная основа технологии UniTESK такова.

- Тесты разрабатываются исходя из некоторой модели поведения тестируемой системы. Критерии полноты тестирования определяются, чаще всего, в терминах этой же модели. Все это позволяет вести разработку тестов независимо от разработки тестируемых компонентов. Модель поведения служит источником для автоматического построения тестовых оракулов.
- Используемая модель поведения обычно создается на основе требований и желательных свойств системы, а не на основе использованных в ней проектных решений. За счет этого тестовый набор необходимо модифицировать, в основном, из-за изменений в требованиях, а не в коде системы. Полученные в результате тесты могут без модификаций или с небольшими изменениями использоваться для тестирования разных версий системы и различных систем, реализующих одни и те же функции, один и тот же стандарт. Различия в интерфейсах между разными системами при этом локализуются в небольшой части тестового набора — тестовых адаптерах.
- Для оформления моделей используются языки, насколько это возможно, близкие к языкам, используемым при разработке системы, с тем чтобы ее разработчики и архитекторы тратили как можно меньше усилий на их понимание. Обычно используются широко распространенные языки программирования или их небольшие расширения.
- Тесты строятся на основе модели поведения и критериев полноты тестирования с помощью нескольких различных техник. Выбор этих техник зависит от требуемого вида тестирования (проверка базовой функциональности, основных сценариев использования, сочетаний различных условий и ограничений и пр.), необходимой полноты, сложности входных данных и состояния системы, наличия параллелизма и асинхронности в ее поведении и других факторов. При этом сочетаются нацеленное тестирование, комбинаторное построение тестовых данных и автоматные техники для проверки работы системы в различных состояниях.

Технология UniTESK состоит из следующих частей.

- Метод разработки тестов, определяющий набор видов деятельности и решаемых ими задач, последовательность их выполнения и правила применения различных техник в зависимости от складывающейся в проекте ситуации.
- Архитектура тестового набора, определяющая основные виды компонентов тестов, связи и способы взаимодействия между ними и правила расширения и модификации созданных тестовых наборов.
- Набор техник построения и организации тестов.
- Набор языков, на которых разрабатываются модели. Большинство таких языков являются расширениями языков программирования, построенными по общим правилам.
- Инструменты, автоматизирующие работу с моделями на определенных языках и построение тестов из них.

Далее дается описание метода разработки тестов. Оно сопровождается примерами построения моделей и тестов на их основе, в основном на расширении языка Java.

Метод

Используемый метод построения тестов включает в себя следующие виды деятельности.

1. Определение целей и рамок проекта.
2. Определение и анализ требований к тестируемой системе.
3. Определение и анализ требований к полноте тестирования.
4. Разработка и выполнение тестов.
5. Анализ результатов тестирования.

Эти виды деятельности обычно выполняются примерно в той последовательности, в которой перечислены выше. Однако при необходимости используется итеративная разработка — т.е. возможны (неоднократные) возвращения от следующих видов деятельности к предыдущим для выполнения каких-то доработок или изменений. Кроме того, после проведения декомпозиции системы на отдельные компоненты в рамках первой или второй деятельности, дальнейшая разработка тестов для каждого компонента может идти независимо от остальных, поэтому разные действия для различных компонентов часто выполняются одновременно и параллельно.

Далее каждый из видов деятельности рассматривается более подробно.

Определение целей и рамок проекта

В ходе выполнения этого вида деятельности принимаются основные стратегические решения, касающиеся проекта. Выявляются границы тестируемой системы, основные проверяемые функции и свойства, интерфейс, которым можно пользоваться, основные риски, на преодоление которых нацелено тестирование. Определяются компоненты системы, которые можно тестировать независимо, набор видов создаваемых тестов и используемые при этом техники.

Принимаемые на этом этапе решения зависят от 3-х видов факторов.

- Контекст предметной области: какие требования выдвигаются к системам такого рода, какие есть документы и знания о предметной области, какие стандарты действуют в данной предметной области, что известно о возможностях использования целевой системы, о потребностях ее пользователей, заказчиков и третьих лиц, о решаемых системой задачах, возможных методах решения этих задач и возможном устройстве соответствующих компонентов системы,.
- Контекст текущего проекта: какие ресурсы (люди, время, деньги, аппаратное и программное обеспечение, другое оборудование) имеются в нашем распоряжении, какие требования к данной системе и ее тестированию выдвигают заказчик и

другие заинтересованные лица (конечные пользователи системы, ее разработчики, контролирующие и лицензирующие организации и пр.), какие другие проекты зависят или, вероятно, будут зависеть от целевой системы и от результатов этого проекта, какие требования предъявляются или, вероятно, будут предъявляться ими к целевой системе и к ее тестам.

- Архитектура целевой системы, насколько она известна на момент начала работ: разбиение системы на компоненты, задачи, решаемые различными ее компонентами, возможные сценарии взаимодействия между ними, а также правила внесения модификаций в имеющиеся компоненты и добавления новых.

Определение рамок проекта включает в себя решение следующих задач.

1. **Определение задач и рамок тестируемой системы.**
   В рамках этой задачи нужно определить, что представляет собой тестируемая система (system under test, SUT), для чего она предназначена — какие основные задачи решает. Также важно, из каких частей она состоит — что входит, а что не входит в нее, какие именно части нужно тестировать, можно ли их отделить от других частей/других систем при проведении тестирования, и как.
2. **Определение набора проверяемых функций и свойств**.
   Нужно выяснить, полный список функций (функция, feature — это некоторая услуга, предоставляемая системой), которые необходимо протестировать, а также список других свойств, которые необходимо проверять. На этом этапе выявляется только общий набор свойств и функций, без детализации.
3. **Определение используемого при тестировании интерфейса**.
   Необходимо определить набор операций или действий, с помощью которых можно воздействовать на систему, и набор ее возможных реакций и событий, создаваемых ею. Нужно стремиться найти такой интерфейс, который позволит как можно точнее оценить проверяемые свойства, с минимальными искажениями от других частей системы или других систем. Иногда, однако, можно пользоваться только таким интерфейсом, который вынуждает работать не только проверяемую функциональность, но и множество других компонентов, вносящих определенные возмущения в наблюдаемые реакции, которые нужно учитывать при тестировании.
4. **Оценка важности различных характеристик, функций и элементов системы**.
   Наборы проверяемых функций и свойств, а также компонентов системы необходимо проранжировать по их важности для основных заинтересованных лиц проекта. При этом необходимо не только использовать пожелания руководителя проекта или пользователей, но и провести аккуратный анализ зависимостей функций и анализ рисков их некорректной работы в различных ситуациях — иногда представления самих разработчиков и пользователей системы о значимости возможных ошибок для оценки работы системы в целом не являются вполне адекватными.
5. **Первичная декомпозиция интерфейса на группы связанных операций**.
   Весь набор операций, действий или событий, с помощью которых можно воздействовать на систему и получать от нее информацию и реакции, нужно разбить на группы логически связанных операций и событий. Каждая такая группа совместно реализует одну или несколько тесно связанных функций. Обычно прямые и обратные операции (создать/уничтожить, добавить/удалить, записать/стереть, войти/выйти и пр.) помещаются в одну группу.
   На этом этапе производится только первичная декомпозиция, предназначенная для предварительного планирования и оценки трудоемкости дальнейших работ. В дальнейшем эта декомпозиция может изменяться и уточняться, поскольку иногда выявляются неявные, но важные, связи и зависимости между функциями и операциями.

Помимо собственно декомпозиции на группы в рамках этой деятельности определяются зависимости между отдельными группами, позволяющие упорядочить разработку модели поведения от базовых понятий и операций к более сложным, использующим эти базовые.

6. **Выбор методов тестирования.**
   Для каждой из выделенных групп операций выявляются ее существенные с точки зрения организации тестирования свойства. К таким свойствам относятся требуемый вид тестирования (проверяется ли только базовая функциональность, основные способы использования или сложные сценарии работы, используются ли некорректные входные данные или нет), наличие или отсутствие работы с внутренним состоянием, высокая или низкая сложность входных данных и части состояния системы, связанной с данной группой, наличие или отсутствие зависимостей от других групп, важность или несущественность (отсутствие возможности) асинхронных взаимодействий с системой в рамках данной группы. В соответствии с выявленными особенностями выбирается наиболее подходящий и эффективный набор техник построения тестов в рамках технологии и видов тестов, которые будут разрабатываться для данной группы.
7. **Выбор аспектов и детальности формализации (уровня абстракции)**.
   В соответствии с выявленными особенностями данной группы и важностью проверки отдельных свойств входящих в нее операций определяется уровень абстракции выполняемой формализации требований — аспекты и детали функциональности и др. свойств, которые должны быть описаны и проверятся, и те аспекты и детали, которые будут проигнорированы.

В дальнейшем различные виды деятельности выполняются уже не по отношению ко всей тестируемой системе, а для выделенных групп интерфейсных операций в соответствии с порядком, устанавливаемым их зависимостями друг от друга — от независящих ни от чего к тем, которые зависят от ранее рассмотренных. Тесты для группы, которые не зависят друг от друга даже косвенно, могут разрабатываться параллельно. При необходимости проводится уточнение или исправление первичной декомпозиции, после чего часть уже выполненных работ во определению требований, критериев полноты или разработке тестов бывает необходимо переделать в соответствии с внесенными изменениями.

Рассмотрим небольшой пример. Пусть необходимо разработать тесты для классов, реализующих коллекции в пакете java.util библиотеки JDK. Всего в этом пакете 19 интерфейсов, 59 классов (14 из них — абстрактные, т.е. предназначенные для создания классов-наследников на их основе), 1 перечислимый тип и 21 класс исключений. Помимо коллекций, эти классы реализуют работу с датами, временем и таймерами, с географическими и национальными настройками системы, с наборами конфигурационных свойств, простейшие операции с регулярными выражениями, а также предоставляют базовые классы для образца «Подписчик». Непосредственно к коллекциям имеют отношение 16 интерфейсов, 27 классов и 3 класса исключений.

Анализ имеющейся документации (http://java.sun.com/javase/6/docs/api/) дает следующую дополнительную информацию.

- Вместо интерфейса Enumeration рекомендуется использовать Iterator, хотя последний более специфичен. По-видимому, первый интерфейс оставлен, в основном, для обеспечения совместимости со старыми программами, написанными до появления Java 1.2.
- Устаревшим также считается классы Dictionary (вместо него нужно использовать интерфейс Map).
- Классы Hastable и Vector очень похожи по выполняемым функциям на HashMap и ArrayList. Даже отличающиеся по имени их методы имеют аналогичную

функциональность. Единственное важное отличие — все методы первых двух классов синхронизованы. Это означает, что они рассчитаны на корректную работу и при асинхронных обращениях к их объектам из различных потоков. Однако, если такие свойства не важны, можно совсем не использовать эти классы.

Рассмотрим теперь отдельно разработку тестов для списков. Список — линейно упорядоченная коллекция объектов, предоставляющая доступ ко всем своим элементам по их индексам. Из классов и интерфейсов коллекций в java.util к спискам относятся один интерфейс List и 5 классов, реализующих его. Два класса — AbstractList и AbstractSequentialList — являются абстрактными, их методы доступны только через наследников (например, ArrayList и LinkedList). Класс Vector считается уже устаревшим, поэтому тесты для него могут не понадобиться.

Класс ArrayList в дополнение к методам интерфейса List предоставляет только методы управления вместимостью (capacity) массива, на основе которого построен данный список. Класс LinkedList содержит дополнительные методы работы с первым и последним элементами. Если тестировать эти функции не нужно, то для обоих классов можно разработать унифицированный набор тестов, проверяющих только методы общего интерфейса. Этот набор тестов можно будет использовать для тестирования соответствующей функциональности любого класса, реализующего интерфейс List.

Определение и анализ требований к тестируемой системе

В рамках данной деятельности определяются требования к тестируемой системе — все ограничения и свойства, которые нужно проверять в ходе тестирования. Кроме того, эти требования систематизируются и оформляются в виде модели поведения системы.

Определение требований и их формализация разбиваются на следующие задачи.

1. **Определение источников требований.**
   Прежде**,** чем начинать выделять отдельные требования, необходимо выявить все возможные их источники. К таким источникам относятся следующие.
   - Проектная и пользовательская документация на систему.
   - Архитекторы, проектировщики и разработчики данной системы.
   - Бизнес-аналитики, эксперты в данной предметной области, опытные пользователи и руководители, технологи в организациях-заказчиках и контролирующих организациях.
   - Стандарты, относящиеся к данной предметной области.
   - Другие аналогичные системы и их документация.
2. **Выделение и сбор требований.**
   После определения источников требований можно начать собирать отдельные требования — все существенные ограничения на работу системы, которые можно найти в документах или выделить из общения с экспертами в предметной области. Каждое такое утверждение должно хранить ссылку на источник, из которого оно получено.
3. **Систематизация требований.**
   После выделения набора требований их необходимо превратить в некоторую систему — каждое требование должно иметь уникальный идентификатор, чтобы на него можно было ссылаться в дальнейшем, должны быть выявлены связи между отдельными требованиями — одни из них могут уточнять другие или являться их следствием. В дальнейшем эти связи помогут аккуратно вносить изменения в систему требований, сразу выявляя нарушения ее целостности и возможные противоречия. Требования должны быть классифицированы по обязательности их выполнения — как минимум нужно отделять обязательные от опциональных.

Нужно стремиться привести требования в проверяемый вид — чтобы обладающий определенной квалификацией человек мог по некоторым результатам работы системы уверенно утверждать, выполнены они или нет. Это не всегда удается, но, по крайне мере, нужно отделить проверяемые требования от прочих.

4. **Уточнение, согласование и устранение противоречий и неполноты**.
   При систематизации и формализации требований выявляется множество дефектов — недостаточно ясные и точные формулировки, противоречия между разными требованиями или пожеланиями различных заинтересованных лиц, неполнота — отсутствие необходимой информации о свойствах системы в каком-либо аспекте. Все эти дефекты необходимо устранить, часто привлекая для этого различных заинтересованных лиц, чтобы выяснить точное значение некоторой фразы или согласовать их противоречивые интересы и привести к разумным компромиссам относительно свойств системы.
5. **Финальная декомпозиция.**
   После приведения требований в систему необходимо проанализировать первичную декомпозицию интерфейса системы и внести в нее необходимые изменения, связанные с полученной более полной информацией о работе различных функций и операций и об их зависимостях друг от друга.
6. **Формализация требований в виде модели поведения**.
   Часто параллельно систематизации и согласованию проводится формализация требований — построение формально описанной модели поведения системы, включающей все ее аспекты, важные с точки зрения данного проекта. В рамках UniTESK используются модели в виде расширенных автоматов, программных контрактов. Для описания сложных структур данных используются иерархические структурные модели (по сути — описания типов данных в виде наборов их полей, также имеющих некоторые типы или являющихся ссылками) или грамматики, пополненные дополнительными атрибутами (но не атрибутные грамматики в смысле Кнута).

Рассмотрим в качестве примера разработку модели поведения в виде программного контракта для интерфейса списка. Его методы можно условно разделить на следующие группы: работа со списком как с целым (equals, hashCode, toString, toArray, clear, size, isEmpty) , методы перебора (iterator, listIterator), методы для работы с отдельными элементами списка (add, contains, get, indexOf, lastIndexOf, remove, set), работа с коллекциями элементов (addAll, containsAll, removeAll, retainAll), работа с подсписками (subList).

Для определения требований достаточно использовать имеющуюся документацию на интерфейс List (http://java.sun.com/javase/6/docs/api/). Она дана в достаточно систематизированном виде, поэтому задачи по выделению и систематизации требований можно считать уже решенными.

Чтобы построить модель поведения списка в виде программного контракта, необходимо определить структуру состояния, содержащую информацию достаточную для описания всех ограничений на его операции, а для каждой операции определить предусловие и постусловие (которые могут зависеть от текущего состояния).

Для полного описания поведения почти всех операций списка необходимо знать полный набор его элементов и их индексы. Поэтому в качестве структуры состояния нам придется использовать тоже список. Важно только, чтобы его реализация как-то отличалась от тех, которые нам предстоит тестировать — иначе при тестах будет проверяться только то, что примененные к двум различным одинаково реализованным объектам операции дают одинаковые результаты.

Метод subList существенно отличается от других — для описания его поведения нужно иметь дополнительную структуру подсписков, построенных на основе данного. В документации сказано, что такие подсписки являются только представлением, «другим

взглядом» на данные исходного списка, т.е. при их создании не создается новая коллекция, а при их модификации модифицируется и содержимое исходного списка. Оставим пока формализацию этой функциональности за рамками рассмотрения. С учетом этого упрощения описание структуры состояния списка может быть сделано так.

```
public specification class ListSpecification<E>
{
  protected E[] items;
  ...
}
```

Теперь опишем метод add(int index, E element). В документации на этот метод сказано следующее.

> Вставляет указанный объект в список на указанную позицию. Элементы, находившиеся в списке на этой позиции или после нее, сдвигаются на одну позицию вперед.
> **Параметры:**
> index – номер позиции, на которую нужно вставить указанный объект.
> element – объект, который нужно вставить.
> **Создаваемые исключения:**
> UnsupportedOperationException – создается, если этот метод не поддерживается данной реализацией списка.
> ClassCastException – создается, если класс указанного объекты препятствует его вставке в список.
> NullPointerException – создается, если указанный объект равен null и данная реализация списка не может хранить null в качестве элемента.
> IllegalArgumentException – создается, если указанный объект не может быть вставлен в данный список.
> IndexOutOfBoundsException – создается, если указана некорректная позиция (index < 0 || index > size()).

Поскольку обращаться к этому методу можно в произвольной ситуации, его предусловие должно быть всегда выполнено.

Кроме того, для простоты предположим, что исключения первых четырех видов не возникают — т.е. тестируемая реализация поддерживает операцию add, может иметь элементы любого типа, подходящего с точки зрения типа второго параметра этого метода и может иметь null в качестве элемента. Это уже не произвольная реализация интерфейса List.

Ниже приведена спецификация метода add, использующая два вспомогательных метода: для сравнения объектов, каждый из которых может быть равен null, и для сравнения участков массивов.

```
public static boolean equalObjects(E o1, E o2)
{
  return    o1 == null && o2 == null
         || o1 != null && o1.equals(o2);
}

public static boolean equalArrays(
     E[] first,  int firstStart
   , E[] second, int secondStart, int number
  )
{
  for(int i = 0; i < number; i++)
    if(!equalObjects(first[i+firstStart], second[i+secondStart]))
      return false;
  return true;
}

public specification void add(int i, E o)
  throws IndexOutOfBoundsException
{
  post
  {
    E[] oldItems = (E[])(pre items.clone());
```

```
        if(i < 0 || i > oldItems.length)
          return    thrown != null
                 && thrown instanceof IndexOutOfBoundsException
                 && items.length == oldItems.length
                 && equalArrays(items, 0, oldItems, 0, items.length);
        else
          return    thrown   == null
                 && items[i] == o
                 && items.length == oldItems.length + 1
                 && equalArrays(items, 0,   oldItems, 0, i)
                 && equalArrays(items, i+1, oldItems, i, oldItems.length-i);
      }
    }
```

В постусловии используется *оператор пре-выражения* pre, результатом применения которого к некоторому выражению является значение этого выражения непосредственно перед началом работы описываемой операции. В данном примере он использован для получения предшествовавшего операции набора элементов списка.

Определение и анализ требований к полноте тестирования

Помимо требований к проверяемой системе, определяющих, что проверять, для построения тестов необходимы требования к полноте тестирования, определяющие, в каких ситуациях нужно выполнять проверки.

В рамках этого вида деятельности нужно решить следующие задачи.

1. **Определение критериев полноты для данного проекта**.
   Критерии полноты тестирования определяются, прежде всего, на основе структуры требований. Кроме этого, учитываются наиболее важные риски, связанные с качеством тестируемой системы. Если известны наиболее критичные, а также наиболее рискованные, ее функции и компоненты, типы наиболее вероятных ошибок, эти функции и компоненты должны проверяться более тщательно, а ситуации, соответствующие наиболее рискованным действиям, должны входить в задаваемый критерий полноты отдельно.
2. **Формализация критериев полноты**.
   Выбранные критерии полноты тестирования нужно представить в виде правил выбора структурных элементов модели поведения, которые необходимо задействовать в тестах. Используется комбинация правил из некоторого общеупотребительного набора — критерий полноты задается в терминах покрытия различных способов поведения, указанных в спецификациях, покрытия комбинаций условий, касающихся выбираемого поведения или используемых данных.
   При использовании грамматик критерий покрытия может задаваться указанием того, что нужно покрыть все правила грамматики, все альтернативы, возможные комбинации альтернатив в рамках одного или нескольких правил, если в одних их них присутствуют нетерминалы, определяемые другими. Дополнительно указываются ограничения на число раскрытий неограниченных списков, используемых в правилах.

В рассмотренном выше примере постусловие метода add списка задает два сильно отличающихся поведения, две *ветви функциональности*. При одном из них создается исключение, а содержимое списка не изменяется, при другом исключения нет, а добавляемый элемент вставляется на указанное первым аргументом место. Различие между этими поведениями отражается в различных выражениях, использованных для их описания при вычислении результата постусловия.

Чтобы явно выделить различные поведения или ветви функциональности, используется оператор оператора **`branch`**. Чтобы сделать покрытие той или иной ветви более

управляемым, условия их выполнения должны зависеть только от входных данных метода — состояния списка при его вызове и значений параметров.

После добавления указания ветвей функциональности постусловие метода add приобретает следующий вид.

```
post
{
  E oldItems[] = (E[])items.clone();

  if(i < 0 || i > oldItems.length)
  {
    branch ExceptionalCase;
    return    thrown != null
           && thrown instanceof IndexOutOfBoundsException
           && items.length == oldItems.length
           && equalArrays(items, 0, oldItems, 0, items.length);
  }
  else
  {
    branch NormalCase;
    return    thrown   == null
           && items[i] == o
           && items.length == oldItems.length + 1
           && equalArrays(items, 0,   oldItems, 0, i)
           && equalArrays(items, i+1, oldItems, i, oldItems.length-i);
  }
```

Поскольку при выборе пути до оператора **branch** могут использоваться только пре-состояние объекта и значения входных параметров, удобно считать, что положение этого оператора соответствует точке вызова тестируемой операции. Все действия, выполняемые до одного из таких операторов считаются выполненными до вызова, и, соответственно, могут использовать только пре-состояние объекта и аргументы метода. Все действия, выполняемые после такого оператора, выполняются после вызова проверяемой операции, и, соответственно, могут использовать ее результат (обозначаемый в постусловии именем самой операции) и пост-состояние объекта. Для обращения после оператора **branch** к значениям в пре-состоянии должен использоваться оператор пре-выражения.

Различные поведения задают естественный *критерий покрытия ветвей функциональности* — полнота тестирования по нему определяется процентом задействованных ветвей функциональности от их общего числа.

Другие ситуации, которые нужно задействовать в ходе тестирования, можно описать в постусловиях соответствующих операций при помощи *операторов разметки ситуаций*. Допустим, в нашем примере нужно особо проверить работу метода add для пустых списков. Включить эту ситуацию в заданный спецификациями критерий покрытия можно следующим образом.

```
post
{
  if(items.length == 0) mark “Empty list”;
  E oldItems[] = (E[])items.clone();

  if(i < 0 || i > oldItems.length)
  {
    branch ExceptionalCase;
    ...
  }
```

```
    else
    {
      branch NormalCase;
      ...
    }
```

При этом возникает более мелкое разбиение всех ситуаций согласно *критерию помеченных путей* — каждая последовательность из выполненных операторов оператора **branch** и **mark** задает помеченный путь, полнота тестирования определяется процентом покрытых помеченных путей от общего числа достижимых.

Допустим, что помимо метода add мы также описали методы remove(int index) и indexOf(E element), отметив в remove в качестве различных поведений нормальное поведение и поведение с созданием исключения, а в indexOf — поведение в ситуации, когда искомый объект находится в списке, и когда его там нет. Кроме того, с помощью меток отметим отдельно случаи пустого списка и списка с единственным элементом. Соответствующие спецификации даны ниже.

```
public static boolean arrayContains(E[] a, int start, int number, E o)
{
  for(int i = 0; i < number; i++)
    if(equalObjects(a[i + start], o)) return true;
  return false;
}

public specification int indexOf(E o)
{
  post
  {
    if     (items.length == 0) mark “Empty list”;
    else if(items.length == 1) mark “List with single element”;

    E oldItems[] = (E[])items.clone();

    if(arrayContains(items, 0, items.length, o))
    {
      branch ObjectInList;
      return    thrown == null
             && objectsAreEqual(oldItems[indexOf], o)
             && !arrayContains(oldItems, 0, indexOf, o)
             && items.length == oldItems.length
             && equalArrays(items, 0, oldItems, 0, items.length);
    }
    else
    {
      branch NoObjectInList;
      return    thrown  == null
             && indexOf == -1
             && items.length == oldItems.length
             && equalArrays(items, 0, oldItems, 0, items.length);
    }
  }
}

public specification E remove(int i)
  throws IndexOutOfBoundsException
{
  post
  {
    if     (items.length == 0) mark “Empty list”;
    else if(items.length == 1) mark “List with single element”;

    E oldItems[] = (E[])items.clone();
```

```
        if(i < 0 || i >= items.length)
        {
          branch ExceptionalCase;
          return    thrown != null
                 && thrown instanceof IndexOutOfBoundsException
                 && items.length == oldItems.length
                 && equalArrays(items, 0, oldItems, 0, items.length);
        }
        else
        {
          branch NormalCase;
          return    thrown == null
                 && remove == oldItems[i]
                 && items.length == oldItems.length - 1
                 && equalArrays(items, 0, oldItems, 0, i)
                 && equalArrays(items, i, oldItems, i+1, items.length-i);
        }
      }
    }
  }
```

Набор возможных помеченных путей в этой спецификации выглядит так.

add()	indexOf()	remove()
Empty list ExceptionalCase	Empty list ObjectInList	Empty list ExceptionalCase
Empty list NormalCase	Empty list NoObjectInList	Empty list NormalCase
ExceptionalCase	List with single element ObjectInList	List with single element ExceptionalCase
	List with single element NoObjectInList	List with single element NormalCase
NormalCase	ObjectInList	ExceptionalCase
	NoObjectInList	NormalCase

Ситуации, выделенные серым цветом в таблице, недостижимы — пустой список не может содержать элементов и из пустого списка нельзя удалить элемент по неотрицательному индексу, меньшему длины списка.

Вторая ситуации автоматически будет отброшена инструментом, поскольку она описывается невозможной комбинацией равенств и неравенств между целыми числами: `items.length == 0`, `!(i < 0)`, `!(i >= items.length)`.

Первая же ситуация описывается комбинацией условий, связь между которыми инструменту, вообще говоря, непонятна: `items.length == 0`, `arrayContains(items, 0, items.length, o)`. Поэтому, чтобы выбросить ее из обрабатываемого инструментом множества ситуаций, надо написать в спецификации *тавтологию*, явно указывающую связь между входящими в эту комбинацию условиями. После этого постусловие метода `indexOf()` выглядит так.

```
      post
      {
        if     (items.length == 0) mark “Empty list”;
        else if(items.length == 1) mark “List with single element”;

        E oldItems[] = (E[])items.clone();

        tautology items.length == 0 =>
          !arrayContains(items, 0, items.length, o);
        ...
      }
```

Для еще более детальной оценки полноты тестирования может использоваться критерий покрытия коротких дизъюнктов, составленных из условий, встречающихся в условиях

операторов ветвления или участвующих в определении ветви оператора выбора до одного из операторов **`branch`**.

Литература