Функциональное тестирование списочной структуры с использованием JavaTESK

Содержание

Общее описание технологии UniTESK....2
Постановка целей проекта и определение его рамок....3
Разработка спецификации....7
- add....10
- indexOf....10
- remove....10

Уточнение требований к качеству тестирования....16
Разработка тестовых сценариев....22
Разработка медиаторов....25
Разработка сценарных классов....27
Работа с тестовым набором....31
- Сборка тестового набора....31
- Выполнение теста....31
- Получение тестовых отчетов и анализ результатов тестирования....31

Общее описание технологии UniTESK

При разработке программного обеспечения (ПО) мы обычно хотим, чтобы оно работало правильно и надежно. Наиболее известный и широко используемый способ проконтролировать правильность его работы и надежность — это тестирование. Однако взрывной рост функциональности и сложности реально используемого ПО в последнее время приводит к проблемам при применении традиционных подходов к тестированию. Протестировать обычными методами небольшую или среднюю (~< 10^4 строк кода) программу возможно, но при дальнейшем росте её размера, а также при необходимости ее долгого сопровождения и многократного внесения изменений, трудоемкость использования традиционных подходов к тестированию возрастает во много раз, а их эффективность значительно падает.

Технология разработки тестов UniTESK призвана разрешить эти проблемы. Она предназначена для снижения трудоемкости и повышения качества тестирования современного сложного ПО.

Основные элементы технологии UniTESK таковы:

- Тесты разрабатываются исходя из некоторой модели поведения ПО, называемой *формальными спецификациями*, а не на основе структуры его кода. Это позволяет начать разработку тестов задолго до завершения реализации целевого ПО.
- Из формальных спецификаций автоматически строятся *критерии тестового покрытия*, определяющие на основе структуры требований критерии полноты тестов. Эти критерии применяются в дальнейшем для контроля качества проведенного тестирования.
- На основе спецификаций и выбранного критерия покрытия строятся *тестовые сценарии*, которые нацелены на достижение максимально возможного покрытия по этому критерию. Тестовые сценарии UniTESK представляют собой аналог широко используемых тестовых скриптов, однако они более выразительны и позволяют при тех же усилиях на разработку провести гораздо более качественное тестирование.
- UniTESK поддерживает тестирование «черным ящиком», без учета особенностей конкретной реализации целевого ПО.
- Формальные спецификации можно использовать для тестирования разных версий одной системы, даже если их интерфейсы различаются, экономя на разработке специфических тестов для них. Эта возможность поддерживается технологией за счет дополнительной прослойки между тестами и реализацией, состоящей из программных компонентов, называемых *медиаторами*.
- Формальные спецификации, медиаторы и тестовые сценарии оформляются на расширении языка программирования, на котором разрабатывается тестируемое ПО. Это позволяет значительно упростить освоение технологии тестировщиками и сделать максимально понятной связь тестов с целевым ПО. Инструмент, поддерживающий разработку тестов по технологии UniTESK, имеется для языка Java.

Основные действия, которые нужно выполнить для разработки теста по технологии UniTESK:

1. *Подготовка формальных спецификаций*. Для этого следует проанализировать функциональные требования к целевому ПО на основе имеющихся документов или

знаний участников проекта.

2. *Выделение критериев покрытия*. Попытаться сформулировать требования к качеству тестов (что будет означать «достаточно полное тестирование», когда можно его прекратить) на основе пожеланий заказчика, знаний о предметной области и имеющихся ресурсов проекта. Используя структуру полученных спецификаций, переформулировать эти требования в виде требований к уровням покрытия по определенным критериям покрытия.
3. *Разработка тестовых сценариев*. Сценарии разрабатываются на основе спецификаций и не привязаны к конкретной реализации целевого ПО или его конкретной версии. Набор тестовых сценариев должен обеспечивать достижение нужного уровня покрытия.
4. *Разработка медиаторов*. Для привязки полученных тестов к конкретной реализации целевой системы разработать набор медиаторов. Для этого нужно знать точный интерфейс выбранной реализации, хотя сама она еще может быть не готова к тестированию.
5. *Компиляция тестовой системы*. Для того, чтобы получить готовые тесты, нужно оттранслировать спецификации, медиаторы и сценарии с расширения языка программирования в целостную тестовую систему на самом этом языке.
6. *Выполнение тестов*. После трансляции становится возможным выполнение тестов. При выполнении тестов на первых порах обычно приходится их отлаживать, удаляя ошибки из спецификаций, сценариев и медиаторов. После отладки нужно выполнить все тесты «начисто» и получить их результаты в виде отчетов для дальнейшего анализа.
7. *Анализ результатов тестирования*. Следует определить, какие ошибки обнаружены, а также, достигнут ли нужный уровень покрытия или надо разрабатывать дополнительные сценарии.

Шаги по разработке тестовых сценариев и медиаторов почти не зависят друг от друга и часто могут выполняться параллельно. Кроме того, недочеты и ошибки, обнаруживаемые в самой тестовой системе на этапах сборки тестов, выполнения тестов и анализа результатов тестирования, могут приводить к пересмотру результатов любого ранее сделанного шага, и, соответственно, к возвращению на этот шаг.

Постановка целей проекта и определение его рамок

Прежде чем приступать непосредственно к разработке тестов, надо определить общие цели и рамки этой разработки. Для этого нужно ответить на следующие два вопроса:

1. Какие части, подсистемы, компоненты целевого ПО мы будем тестировать?
2. На соответствие каким требованиям они будут тестироваться?

Ответы на эти вопросы зависят обычно от следующих 4-х видов факторов:

- Контекст предметной области: какие требования выдвигаются к системам такого рода, имеющиеся документы и знания о предметной области, о контексте использования целевой системы, о потребностях ее пользователей и других лиц, о требованиях контролирующих организаций и стандартов, о решаемых системой задачах, возможных методах решения этих задач и возможном устройстве соответствующих компонентов системы.

- Архитектура целевой системы, насколько она известна на момент начала работ: разбиение системы на компоненты, задачи, решаемые различными ее компонентами, и возможные взаимодействия между ними.
- Контекст текущего проекта: какие ресурсы (люди, время, деньги, аппаратное и программное обеспечение) 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.).
- Контекст связанных проектов: 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 качеству целевой системы и к качеству ее тестирования.

Дальнейшая часть документа посвящена разработке тестов для стандартной библиотеки коллекций платформы Java 5. Рассматриваемые классы и интерфейсы определены в пакете `java.util`.

Сведения о целевой системе и требованиях к ее поведению можно подчерпнуть из следующих источников:

- Документация на систему
- Эксперты в данной предметной области (эксперты по библиотекам JDK)
- Стандарты, относящиеся к данной предметной области (требования, накладываемые спецификацией языка Java, принятыми шаблонами разработки)
- Архитекторы, проектировщики и разработчики данной системы

В нашем примере имеется стандартная документация к JDK 1.5. Ниже приведены краткие описания классов и интерфейсов из пакета `java.util`, относящихся к работе с коллекциями:

Interface Summary	
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
Queue<E>	A collection designed for holding elements prior to processing.
RandomAccess	Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access.
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time.
SortedSet<E>	A set that further guarantees that its iterator will traverse the set in ascending element order,

	sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

Class Summary	
AbstractCollection<E>	This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.
AbstractList<E>	This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
AbstractMap<K,V>	This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface.
AbstractQueue<E>	This class provides skeletal implementations of some `Queue` operations.
AbstractSequentialList<E>	This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
AbstractSet<E>	This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.
ArrayList<E>	Resizable-array implementation of the `List` interface.
Arrays	This class contains various methods for manipulating arrays (such as sorting and searching).
Collections	This class consists exclusively of static methods that operate on or return collections.
Dictionary<K,V>	The `Dictionary` class is the abstract parent of any class, such as `Hashtable`, which maps keys to values.
EnumMap<K extends Enum<K>,V>	A specialized `Map` implementation for use with enum type keys.
EnumSet<E extends Enum<E>>	A specialized `Set` implementation for use with enum types.
HashMap<K,V>	Hash table based implementation of the `Map` interface.
HashSet<E>	This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance).
Hashtable<K,V>	This class implements a hashtable, which maps keys to values.
IdentityHashMap<K,V>	This class implements the `Map` interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
LinkedHashMap<K,V>	Hash table and linked list implementation of the `Map` interface, with predictable iteration order.
LinkedHashSet<E>	Hash table and linked list implementation of the `Set` interface, with predictable iteration order.
LinkedList<E>	Linked list implementation of the `List` interface.
PriorityQueue<E>	An unbounded priority queue based on a priority heap.
Stack<E>	The `Stack` class represents a last-in-first-out (LIFO) stack of objects.
TreeMap<K,V>	Red-Black tree based implementation of the `SortedMap` interface.
TreeSet<E>	This class implements the `Set` interface, backed by a `TreeMap` instance.
Vector<E>	The `Vector` class implements a growable array of objects.
WeakHashMap<K,V>	A hashtable-based `Map` implementation with *weak keys*.

Exception Summary	
ConcurrentModificationException	This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.
EmptyStackException	Thrown by methods in the `Stack` class to indicate that the stack is empty.
NoSuchElementException	Thrown by the `nextElement` method of an `Enumeration` to indicate that there are no more elements in the enumeration.

Итого, имеется 13 интерфейсов, 25 классов и 3 класса исключений.

Еще раз повторим два вопроса, на которые надо дать ответ.

1. Какие компоненты системы будут тестироваться.
2. Какие требования при этом будут проверяться.

При ответе на второй вопрос (о проверяемых требованиях) нужно использовать *абстракцию* или *обобщение* как способ снижения затрат на разработку тестов путем выявления общей функциональности у разных компонентов, общих требований к ним. Для тестирования общей функциональности зачастую можно разработать один тест, который может связываться с различными целевыми компонентами через простые промежуточные компоненты — *медиаторы*.

- Для тестирования большинства классов исключений (за исключением `MissingResourceException`) надо проверить только правильность их построения при помощи конструкторов (при помощи конструктора без параметров и при помощи конструктора с параметром-строкой) и работу методов, унаследованных от `java.lang.Throwable`. Для этого достаточно использовать один и тот же тест, подключая его к разным целевым классам при помощи разных медиаторов. Если у нас уже имеется такой тест для класса `java.lang.Throwable`, мы можем использовать его и не делать отдельных тестов для указанных 3 классов.
- Некоторые классы среди указанных выше имеют практически одинаковую функциональность (`HashMap` и `Hashtable`, `ArrayList` и `Vector`), отличаясь только поведением в многопоточном окружении. Тесты на последовательные обращения к их методам могут быть одинаковыми. Делать отдельные тесты на их поведение в многопоточном окружении также не имеет смысла, поскольку первые классы из этих пар для такого использования не предназначены, а правильное поведение вторых обеспечивается средствами самого языка Java.
- Не имеет смысла разрабатывать отдельные тесты для абстрактных классов, предоставляющих некоторую базовую реализацию для интерфейсов (`AbstractCollection`, `AbstractList`, `AbstractMap`, `AbstractSet`). Вся их функциональность можт быть проверена при помощи тестов для интерфейсов.

При детальном расмотрении методов отдельных классов могут выясниться дополнительные подробности. Например, может оказаться, что в рамках данного проекта не надо тестировать в классе `ArrayList` функциональность, связанную с изменением вместимости такого списка (конструктор `ArrayList(int)` и методы `void ensureCapacity(int)` и `void trimToSize()`). При этом оставшаяся функциональность `ArrayList` сводится к методам работы со списком, которые должны тестироваться тестами для интерфейса `List`.

Таким образом, мы можем более четко определить, для каких именно классов надо разрабатывать тесты и как, примерно, они должны работать.

Более детальное определение требований к качеству тестирования мы пока отложим до точного выяснения того, что же именно тесты должны проверять, т.е. до разработки спецификаций.

Разработка спецификации

Пусть мы включили класс `ArrayList` в список классов, для которых нужно разработать тесты. Пусть также требуется протестировать в нем только методы, реализующие интерфейс `List`. В соответствии с технологией UniTESK надо начать с точного описания функциональности интерфейса, т.е. с разработки формальных спецификаций.

Спецификации, используемые в UniTESK, описывают структуры данных целевых компонентов при помощи описания их полей и *инвариантов* – дополнительных ограничений на данные, которые должы быть выполнены для корректно построенного компонента. При этом описываются только те поля, которые действительно необходимы для объяснения того, как этот компонент работает. Кроме того для всех методов компонента описываются их *предусловия* и *постусловия*. Предусловие определяет ситуации, в которых этот метод может быть вызван. Постусловие определяет ограничения на результаты его работы, которые должны быть выполнены. Посмотрим на описание методов интерфейса `List`.

Method Summary	
`boolean`	**`add`**`(E o)` Appends the specified element to the end of this list (optional operation).
`void`	**`add`**`(int index, E element)` Inserts the specified element at the specified position in this list (optional operation).
`boolean`	**`addAll`**`(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
`boolean`	**`addAll`**`(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
`void`	**`clear`**`()` Removes all of the elements from this list (optional operation).
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the specified element.
`boolean`	**`containsAll`**`(Collection<?> c)` Returns `true` if this list contains all of the elements of the specified collection.
`boolean`	**`equals`**`(Object o)` Compares the specified object with this list for equality.
`E`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`hashCode`**`()` Returns the hash code value for this list.
`int`	**`indexOf`**`(Object o)` Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
`boolean`	**`isEmpty`**`()` Returns `true` if this list contains no elements.
`Iterator<E>`	**`iterator`**`()`

	Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
ListIterator<E>	**listIterator**() Returns a list iterator of the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence in this list of the specified element (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all the elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

Далее считаем целевой системой выбранную для тестирования группу методов, в данном случае все методы `java.util.List`.

Для разработки спецификаций набора методов нужно определить следующее:

1. Состояние системы, позволяющее описать ее реакцию на вызов любого метода.
2. Для каждого метода определить его сигнатуру: понятное имя, типы параметров, тип результата, типы создаваемых исключений, объекты, к которым метод имеет доступ.
3. Для каждого метода определить его предусловие, показывающее в каких ситуациях его поведение определено, когда его можно вызывать.
4. Для каждого метода определить постусловие, описывающее ограничения на результаты правильной работы метода.

Рассмотрим эти вопросы более детально.

1. Состояние системы определяется тем, какая информация об истории работы системы требуется, чтобы полностью определить результат вызова любого метода с любыми

значениями параметров. Если эта информация и методы разбиваются на несколько частей таким образом, что каждая группа методов зависит только от данных своей части и не зависит от остальных, нужно исходную группу методов разделить эти части и описывать каждую из них как отдельную целевую систему. В нашем случае для полного описания работы всех методов `java.util.List` достаточно знать все элементы списка и их порядок следования в списке.

2. Для каждого метода целевой системы нужно определить его сигнатуру, которой он будет описываться в спецификациях. При этом следует помнить о следующем:

    a. Типы параметров, результата и создаваемых спецификацией метода исключений могут отличаться от соответсвующих типов в реализации. В том случае если в реализации эти типы являются классами, которые мы тоже специфицируем в рамках данного проекта (или ранее проведенных), надо использовать в качестве типов параметров и результата спецификационные аналоги реализационных типов. Например, в нашем примере метод `subList()` возвращает объект типа список. При спецификации тип результата этого метода надо определить как спецификационный список.

    b. При указании типов возможных исключений надо перечислять все типы исключений, которые могут возникнуть при работе данного метода в нормальных условиях, в том числе и типы непроверяемых исключений Java, наследников классов `RunitmeException` и `Error`. Под «нормальными условиями» имеется в виду выполнение предусловия метода (см. ниже) и, может быть, дополнительных условий на доступные ресурсы, связанных с предполагаемым режимом работы тестов, например, наличие достаточно большого объема свободной памяти и пр.

3. Для каждого метода целевой системы нужно определить предусловие данного метода. Для этого надо выделить ситуации (состояния системы и значения параметров метода, с которым он был вызван), для работы в которых этот метод не предназначен. В таких ситуациях поведение метода не определено и его вызов может привести к непредсказуемым последствиям, вплоть до разрушения целевой системы. Для компонентов библиотек, доступных для широкого использования, чаще всего вызов метода в произвольной ситуации не должен приводить к разрушению системы, т.е. его предусловие должно быть всегда выполнено. О некорректных значениях своих аргументов, которые он не может обработать по общему правилу, такой метод может сообщить, вернув особое значение результата или создав исключение. Не для каждой операции предусловие должно быть всегда выполнено. Например, в некоторой системе управления памятью операция освобождения памяти может для обеспечения высокой эффективности работы требовать, чтобы ей в качестве аргумента всегда давалась ссылка на объект, память под который выделялась ранее при помощи определенной операции. Такие ограничения особо подчеркиваются в документации и присутствуют обычно только для таких операций, пользоваться которыми может ограниченный круг разработчиков. В нашем случае у всех методов `java.util.List` предусловие должно быть выполнено всегда, поскольку это библиотечный интерфейс общего пользования.

Для каждого метода целевой системы нужно определить постусловие данного метода. Постусловие описывает ограничения на результаты корректной работы метода и может быть извлечено из документации или других источников, содержащих требования к его работе. В нашем случае имеется достаточно полная документация для методов интерфейса `java.util.List`. Рассмотрим несколько из них.

add

```
public void add(int index,
                E element)
```

Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Parameters:

`index` - index at which the specified element is to be inserted.

`element` - element to be inserted.

Throws:

`UnsupportedOperationException` - if the `add` method is not supported by this list.

`ClassCastException` - if the class of the specified element prevents it from being added to this list.

`NullPointerException` - if the specified element is null and this list does not support null elements.

`IllegalArgumentException` - if some aspect of the specified element prevents it from being added to this list.

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size()).

---

indexOf

```
public int indexOf(Object o)
```

Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element. More formally, returns the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))`, or -1 if there is no such index.

Parameters:

`o` - element to search for.

Returns:

the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.

Throws:

`ClassCastException` - if the type of the specified element is incompatible with this list (optional).

`NullPointerException` - if the specified element is null and this list does not support null elements (optional).

---

remove

```
public E remove(int index)
```

Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

Parameters:

`index` - the index of the element to removed.

Returns:

the element previously at the specified position.

Throws:

`UnsupportedOperationException` - if the `remove` method is not supported by this list.

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size()).

Предположим, что наши тесты будут расчитаны на реализацию общей функциональности списка, не имеющей ограничений на классы объектов и сами объекты, которые могут в нем хранится. Кроме того, предположим, что все операции списка ею поддерживаются.

Это означает, что ситуаций, соответствующих созданию исключений `UnsupportedOperationException`, `IllegalArgumentException`, при использовании такой реализации не возникает. Подобные предположения незначительно сужают область применимости тестов, которые будут получены в результате.

Перейдем теперь к собственно разработке спецификаций.

Декларация спецификационного класса, который будет содержать спецификацию списка, выглядит следующим образом. Спецификационный класс имеет типовой параметр, поскольку спецификация предназначена для списков элементов одного типа.

```
package jatva.examples.list;

specification class ListSpecification<T>
{
  ...
}
```

Состояние списка, как мы уже замечали, полностью описывается перечислением его элементов в той последовательности, в которой они идут в списке. Для того, чтобы иметь возможность использовать состояние списка в описании поведения его методов, его нужно оформить в виде данных – одного или нескольких полей в спецификационном классе. Используем для этого массив объектов, дополнив его общедоступными методами для просмотра находящихся в нем объектов.

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

При разработке спецификаций и тестов будем придерживаться следующего правила: при использовании объектов спецификационных классов обычно не следует изменять их состояние, за исключением синхронизации состояния спецификационных объектов и целевой системы, о которой речь пойдет при разработке медиаторов.

Рассмотрим создание спецификации для метода `add()` списка.

В спецификации используем то же имя метода. Поскольку типы параметров и результата метода, а также создаваемых им исключений, не входят в специфицируемую систему, мы не будем их заменять. При своей работе метод `add()` использует значения своих параметров и может изменять состояние списка, поэтому сигнатура его спецификации выглядит так:

```
specification class ListSpecification<T>
{
  public T[] items;

  specification void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    ...
  }
```

```
}
```

Как мы уже упоминали, предусловие всех методов списка должно быть всегда выполнено. Описать это можно следующим образом:

```
specification class ListSpecification<T>
{
  public T[] items;

  specification void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    pre { return true; }
    ...
  }
}
```

Однако проще всего вообще опустить предусловие, что также интерпретируется как его истинность в любой ситуации.

При разработке постусловия сначала опишем ограничения на результаты работы метода в нормальной ситуации, при которой не должно возникать исключений. При этом указанный в качестве второго аргумента объект должен встать на i-ю позицию в списке, все элементы списка до этой позиции должы остаться на своих местах, а все элементы, стоящие на позициях с номером, большим i, должны сдвинуться на одну позицию вправо. Для того, чтобы это проверить, создадим временную копию исходного списка, а затем, используя ее, проверим, что результирующий список устроен так, как нужно. Для сравнения интервалов массивов создадим вспомогательный метод `arrayCompare`:

```
  public static <T> boolean objectsAreEqual(T o1, T o2)
  {
    return    o1 == null && o2 == null
           || o1 != null && o1.equals(o2);
  }

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

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

      return    thrown    == null
             && items[i] == o
             && items.length == oldItems.length + 1
             && arrayCompare(items, 0,   oldItems, 0, i)
             && arrayCompare(items, i+1, oldItems, i, oldItems.length-i);
    }
  }
```

Оператор **pre** употребляется здесь для получения копии массива `items` до вызова целевого метода. Поскольку `items` является ссылкой на объект, а не значением, вызван метод `clone()`.Оператор **pre** имеет меньший приоритет, чем вызов метода, поэтому результатом вызова `clone()` является копия массива, взятая до обращения к целевому методу. В том случае, если выражение `a` имеет необъектный тип, для получения его значения до вызова метода достаточно просто взять `pre a`.

При вычислении результата постусловия при помощи выражения `thrown == null` мы проверяем, что не было создано исключений. Ключевое слово **thrown** обозначает ссылку на созданное исключение.

Дополним теперь постусловие ограничениями, которые упоминаются в требованиях в связи с исключительными ситуациями. В ситуациях, в которых указанный номер позиции i не попадает в интервал корректных позиций, должно быть создано исключение типа `IndexOutOfBoundsException` и список не должен измениться.

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

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

Действуя аналогичным образом, можно специфицировать и метод `indexOf()`.
Результирующая спецификация выглядит так:

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

specification int indexOf(T o)
{
  post
  {
    T oldItems[] = pre (T[]) items.clone();
```

```
      if(arrayContains(items, 0, items.length, o))
      {
        return    objectsAreEqual(oldItems[indexOf], o)
               && !arrayContains(oldItems, 0, indexOf + 1, o)
               && items.length == oldItems.length
               && arrayCompare(items, 0, oldItems, 0, items.length);
      }
      else
      {
        return    indexOf == -1
               && items.length == oldItems.length
               && arrayCompare(items, 0, oldItems, 0, items.length);
      }
    }
  }
```

В приведенной спецификации введено ограничение на результат, возвращаемый методом. На этот результат можно сослаться с помощью имени метода.

Приведем теперь спецификации всех выбранных выше методов. Мы выделим также вспомогательный метод для сравнения объектов, каждый из которых может быть равен `null`.

```
package jatva.examples.list;

specification class ListSpecification<T>
{
  public T[] items;

  public static <T> boolean objectsAreEqual(T o1, T o2)
  {
    return    o1 == null && o2 == null
           || o1 != null && o1.equals(o2);
  }

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

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

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

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

specification int indexOf(T o)
{
  post
  {
    T oldItems[] = pre (T[]) items.clone();

    if(arrayContains(items, 0, items.length, o))
    {
      return     objectsAreEqual(oldItems[indexOf], o)
              && !arrayContains(oldItems, 0, indexOf + 1, o)
              && items.length == oldItems.length
              && arrayCompare(items, 0, oldItems, 0, items.length);
    }
    else
    {
      return     indexOf == -1
              && items.length == oldItems.length
              && arrayCompare(items, 0, oldItems, 0, items.length);
    }
  }
}

specification T remove(int i)
  throws IndexOutOfBoundsException
{
  post
  {
    T oldItems[] = pre (T[]) items.clone();

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

```
                && arrayCompare(items, i, oldItems, i+1, items.length - i);
        }
      }
    }
}
```

Осталось написать конструктор, инициализирующий состояние модельного объекта (поля спецификационного класса). В данном случае нужно в поле `items` поместить пустой массив, что можно сделать следующим образом:

```
  @SuppressWarnings("unchecked")
  public ListSpecification()
  {
    items = (T[])new Object[0];
  }
```

Теперь у нас имеются спецификации, представляющие требования к целевой системе в формализованном виде, удобном для автоматизированной разработки тестов.

Уточнение требований к качеству тестирования

На данном этапе разработки тестов нужно попытаться для каждого выбранного для тестирования компонента, подсистемы, класса и метода определить качество тестирования, которое надо для них провести.

Обычно качество тестирования измеряется на основе обеспечиваемого им *покрытия* различных ситуаций, возникающих в ходе работы системы (они называются *целями покрытия*). На основе некоторого *критерия покрытия* выделяется полный набор ситуаций, во время выполнения тестов определяется, какие из этих ситуаций реализовались, и отношения количества реализовавшихся ситуаций к общему их числу — достигнутый процент покрытия — считается показателем качества проведенного тестирования.

Для того, чтобы разработать тесты высокого качества, надо сразу стараться делать их так, чтобы они обеспечивали как можно более высокий процент покрытия по выбранному критерию.

Критерии покрытия делятся на две больших группы:

- Критерии первой группы определяют набор ситуаций на основе структуры самой целевой системы — на основе ее архитектуры, внутреннего устройства компонентов, структуры кода. Например, можно считать целями покрытия вызовы всех методов, декларированных в классах целевой системы, и измерять качество тестирования процентом методов, которые были вызваны во время теста. Другой пример — покрытие строк кода, при этом целями покрытия являются выполнения всех строк кода целевой системы. Такие критерии покрытия называются *структурными* и используются при *структурном тестировании*.

- Критерии второй группы определяют набор целей покрытия на основе структуры требований к системе. Например, целями покрытия могут быть отдельные требования, и качество тестирование измеряется при этом отношением количества протестированных требований к общему их числу. Такие критерии называются *функциональными* и используются при *функциональном тестировании*.

Технология UniTESK предназначена для разработки функциональных тестов, нацеленных на достижение высокого качества по функциональным критериям покрытия. При этом эти критерии устанавливаются на основе структуры спецификаций, разработанных на предыдущем шаге. Это возможно, поскольку спецификации разрабатываются исходя из требований и, по сути, являются формализованным их представлением. Кроме того,

разработку таких тестов легко автоматизировать, потому что формальные спецификации могут быть обработаны автоматически.

Рассмотрим еще раз спецификацию метода `add()` списка.

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

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

Постусловие этого метода имеет две ветви, в которых описаны существенно разные ограничения на результаты его работы. Поэтому логично считать, что стоит протестировать поведение метода в обоих этих ситациях. Такие разветвления в постусловии, которое определяют разные наборы ограничений на результаты целевого метода, называются *ветвями функциональности*. Их нужно выделять при помощи оператора **branch**. Кроме того, условия попадания в первую или вторую ветвь зависят только от входных данных метода — состоянияи списка при его вызове и значений параметров.

UniTESK требует, чтобы условия попадания в разные ветви функциональности зависели только от входных данных. При этом в постусловии считается, что все, что происходит до операторов **branch**, выполняется до вызова целевого метода, а все, что после — после вызова целевого метода. Поэтому все идентификаторы до операторов **branch** обозначают значения соответствующих полей, параметров и т.п. до вызова метода. Оператор **pre** для получения пре-значения выражений тоже можно использовать только после операторов **branch**. Снабдим каждый оператор **branch** описанием ситуации. В связи с этим, спецификация метода `add()` переписывается следующим образом:

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

    if(i < 0 || i > items.length)
    {
      branch ExceptionalCase ( “Exceptional case” );
      return    thrown != null
             && thrown instanceof IndexOutOfBoundsException
             && items.length == oldItems.length
```

```
              && arrayCompare(items, 0, oldItems, 0, items.length);
      }
      else
      {
        branch NormalCase ( "Normal case" );
        return     thrown    == null
              && items[i] == o
              && items.length == oldItems.length + 1
              && arrayCompare(items, 0,   oldItems, 0, i)
              && arrayCompare(items, i+1, oldItems, i, oldItems.length-i);
      }
    }
  }
```

При этом ветви функциональности естественным образом задают критерий покрытия, основанный на структуре спецификаций.

Переработанная с учетом приведенных требований спецификация трех методов списка выглядит так:

```
package jatva.examples.list;

specification class ListSpecification<T>
{
  public T[] items;

  @SuppressWarnings("unchecked")
  public ListSpecification()
  {
    items = (T[])new Object[0];
  }

  public static <T> boolean objectsAreEqual(T o1, T o2)
  {
    return     o1 == null && o2 == null
           || o1 != null && o1.equals(o2);
  }

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

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

  specification void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    post
    {
```

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

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

specification int indexOf(T o)
{
  post
  {

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

    if(arrayContains(items, 0, items.length, o))
    {
      branch ListContainsTheObject ( "List contains the object" );
      return     objectsAreEqual(oldItems[indexOf], o)
              && !arrayContains(oldItems, 0, indexOf + 1, o)
              && items.length == oldItems.length
              && arrayCompare(items, 0, oldItems, 0, items.length);
    }
    else
    {
      branch ListDoesNotContainTheObject("List does not contain the object");
      return     indexOf == -1
              && items.length == oldItems.length
              && arrayCompare(items, 0, oldItems, 0, items.length);
    }
  }
}

specification T remove(int i)
  throws IndexOutOfBoundsException
{
  post
  {
    T oldItems[] = (T[])items.clone();

    if(i < 0 || i >= items.length)
    {
      branch ExceptionalCase ( "Exceptional case" );
      return     thrown != null
              && thrown instanceof IndexOutOfBoundsException
              && items.length == oldItems.length
              && arrayCompare(items, 0, oldItems, 0, items.length);
```

```
      }
      else
      {
        branch NormalCase ( “Normal case” );
        return     thrown == null
                && remove == oldItems[i]
                && items.length == oldItems.length - 1
                && arrayCompare(items, 0, oldItems, 0, i)
                && arrayCompare(items, i, oldItems, i+1, items.length - i);
      }
    }
  }
}
```

Теперь мы можем потребовать, чтобы тесты обеспечивали определенный процент покрытия ветвей функциональности, например покрывали бы их все (обеспечивали 100%-е покрытие). Все ветви функциональности методов одного класса задают *критерий покрытия ветвей функциональности*.

Определив ветви функциональности, нужно проанализировать, насколько их 100%-е покрытие соответствует тем требованиям к качеству тестирования, в которых в данном проекте заинтересованы заказчики, разработчики и другие лица, имеющие к нему отношение. Если требуется покрытие большего числа ситуаций, эти ситуации можно явно определить в спецификациях, связав с ними некоторые метки. Пусть, например, в нашем проекте требуется проверить работоспособность всех методов для пустого списка, а работоспособность методов `indexOf()` и `remove()` еще и для списков, содержащих один элемент. Каждую такую дополнительную ситуацию определим с помощью оператора **mark**. Дополним им пост-условие, поместив этот оператор в самое начало и сохранив весь оставшийся код пост-условия:

```
specification class ListSpecification<T>
{
  ...
  specification void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    post
    {
      if (items.length == 0) mark "Empty list";
      ...
    }
  }

  specification int indexOf(T o)
  {
    post
    {
      if      (items.length == 0) mark "Empty list";
      else if(items.length == 1) mark "List with single element";
      ...
    }
  }

  specification T remove(int i)
    throws IndexOutOfBoundsException
  {
    post
    {
      if      (items.length == 0) mark "Empty list";
      else if(items.length == 1) mark "List with single element";
```

```
        ...
      }
    }
}
```

Расставленные при помощи операторов **mark** метки определяют *критерий покрытия помеченных путей*, при использовании которого различные ситуации соответствуют различным комбинациям меток операторов **mark** и **branch**, встречающихся при проверке постусловия.

В нашем примере эти ситуации можно свести в следующую таблицу.

Метод	**add()**	**indexOf()**	**remove()**
Пустой список	Empty list Exceptional case	Empty list List contains the object	Empty list Exceptional case
	Empty list Normal case	Empty list List does not contain the object	Empty list Normal case
Список с одним элементом	Exceptional case	List with single element List contains the object	List with single element Exceptional case
	Normal case	List with single element List does not contain the object	List with single element Normal case
Список с несколькими элементами	Exceptional case	List contains the object	Exceptional case
	Normal case	List does not contain the object	Normal case

Таблица 1: Тестовые ситуации, описываемые критерием покрытия маркированных путей

Заметим, что выделенные ячейки таблицы соответствуют ситуациям, которые никогда не могут реализоваться — пустой список не может содержать элементов и для пустого списка нельзя подобрать значение параметра, большее или равное 0, но меньшее его длины, которая тоже равна 0.

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 понятна только человеку: `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. Окончательный вид спецификации метода `indexOf()` выглядит так:

```
specification int indexOf(T o)
{
  post
  {
    if      (items.length == 0) mark "Empty list";
    else if(items.length == 1) mark "List with single element";

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

    tautology items.length != 0 || !arrayContains(items, 0, items.length, o);

    if(arrayContains(items, 0, items.length, o))
    {
```

```
      branch ListContainsTheObject ( "List contains the object" );
      return    objectsAreEqual(oldItems[indexOf], o)
             && !arrayContains(oldItems, 0, indexOf + 1, o)
             && items.length == oldItems.length
             && arrayCompare(items, 0, oldItems, 0, items.length);
    }
    else
    {
      branch ListDoesNotContainTheObject("List does not contain the object");
      return    indexOf == -1
             && items.length == oldItems.length
             && arrayCompare(items, 0, oldItems, 0, items.length);
    }
  }
}
```

Теперь разработка спецификации полностью закончена и точно определена цель построения тестов — достижение полного покрытия по критерию помеченных путей.

Разработка тестовых сценариев

Имея спецификации и требования к качеству тестирования в терминах покрытия функциональных ветвей и меток, можно переходить к разработке тестового сценария.

Тестовый сценарий в технологии UniTESK задает способ построения последовательности тестов. Для систем, поведение операций которых зависит только от аргументов, с которыми к ним обращаются, тестовый сценарий описывает набор таких обращений, обеспечивающий требуемую полноту тестирования — высокий процент покрытия по выбранному критерию.

Для более сложных систем, поведение операций которых зависит не только от их аргументов, но и от ранее сделанных обращений к системе, для обеспечения полноты тестирования во время теста требуется попадать в различные состояния системы. Для решения этой задачи тестовый сценарий кратко описывает *конечный автомат*, моделирующий поведение целевой системы. При выполнении теста для этого конечного автомата будет построен обход всех его переходов, т.е. для каждого состояния, обнаруженного во время тестирования, будут выполнены все операции, которые можно в нем выполнить. И в этом случае сценарий должен обеспечить нам высокий процент покрытия по выбранному критерию, в идеале — полное покрытие.

Рассмотрим различные ситуации, используя критерий маркированных путей в спецификационном классе `ListSpecification`. Они приведены в Таблице 1. Для того, чтобы построить покрывающий их тестовый сценарий, надо определить конечный автомат, обход которого покроет все эти ситуации. Для определения конечного автомата надо задать

1. Состояния автомата, называемые *обобщенными состояниями сценария*. В нашем примере можно в качестве состояния автомата использовать ту часть условий возникновения различных ситуаций, которая не зависит от параметров методов. Таким образом получается три состояния: пустой список, список с одним элементом и список с несколькими элементами.

2. Переходы автомата. Для задания переходов автомата достаточно указать, какие методы и с какими значениями параметров вызывать в каждом из состояний. В нашем примере в каждом из обобщенных состояний для каждого из методов мы должны покрыть две ситуации: для методов `add()` и `remove()` это ситуации, соответствующие нормальной работе и созданию исключения, для метода `indexOf()` эти ситуации различаются тем, содержит ли список объект, передаваемый в качестве

параметра, или нет. Поэтому, достаточно в каждом состоянии для каждого метода определить ровно два перехода, использовав разные значения параметров. Для метода `add()` в качестве аргумента, дающего ситуацию `Normal case` достаточно всегда использовать 0, а в качестве аргумента, дающего ситуацию `Exceptional case` — -1. Для метода `remove()` можно использовать те же значения для тех же случаев. Для метода `indexOf()` можно использовать значение `items[0]` в качестве аргумента, реализующего ситуацию `List contains the object`, за исключением случая, когда список пуст. Чтобы реализовать ситуацию `List does not contain the object` можно в качестве аргумента взять значение `new Object()`. Граф состояний полученного автомата представлен на рисунке 1.

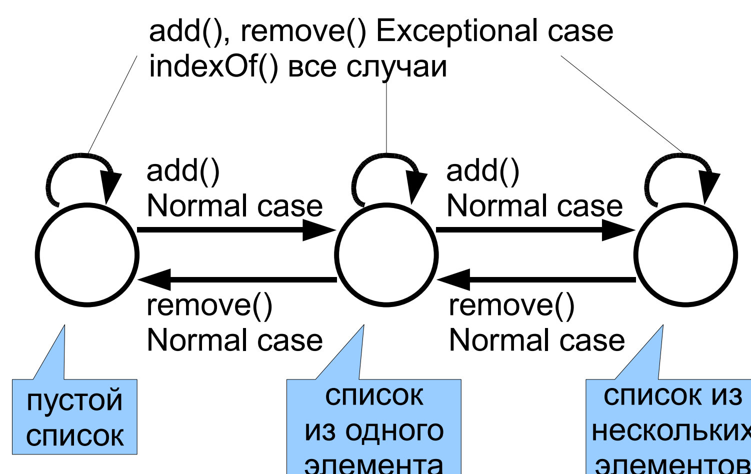


Рисунок 1: Граф состояний конечного автомата, моделирующего список

Помимо описания автомата при разработке тестового сценария нужно определить алгоритм, используемый для построения обхода этого автомата во время теста. Такой алгоритм реализуется одним их *обходчиков*, библиотечных компонентов, прилагаемых к инструменту JavaTESK. Нужно позаботиться о том, чтобы выбранный обходчик смог работать с построенным автоматом.

Для нашего примера выберем обходчик `jatva.engines.DFSMExplorer` который может работать с детерминированными сильно связными автоматам.

Сильная связность автомата означает, что из любого его состояния можно попасть в любое другое, пройдя по нескольким переходам. В нашем примере это так.

Детерминированность автомата означает, что вызов любой операции в произвольном состоянии определяет однозначно переход из этого состояния в некоторое другое, причем это другое состояние должно быть одним и тем же, несмотря на то, каким путем мы попали в исходное состояние.

В нашем примере автомат недетерминирован — при вызове метода `remove()` с параметром, лежащим в границах списка, в состоянии «список из нескольких элементов» мы можем как остаться в том же состоянии (если в списке больше 2-х элементов), так и перейти в состояние «список из одного элемента» (если в исходном списке их ровно 2).

Технология UniTESK требует в таких случаях доработать автомат так, чтобы он стал детерминированным. Проще всего это сделать, «расщепляя» состояния, в которых есть недетерминизм, до тех пор, пока не получится детерминированный автомат.

В нашем примере надо «расщепить» состояние «список из нескольких элементов» на два:

«список из 2-х элементов» и «список с более чем 2-мя элементами». При этом недетерминизм останется при вызове метода `remove()` в состоянии «список с более чем 2-мя элементами». «Расщепляя» его далее, мы получим «список из 3-х элементов», «список из 4-х элементов», и.т.д. — *бесконечный* автомат, в котором состояния соответствуют числу элементов в списке.

Для задания теста нам теперь надо как-то ограничить автомат, например, запретив добавлять новые элементы в список, если в нем уже больше 8-ми элементов. Граф состояний полученного автомата показан на рисунке 2.

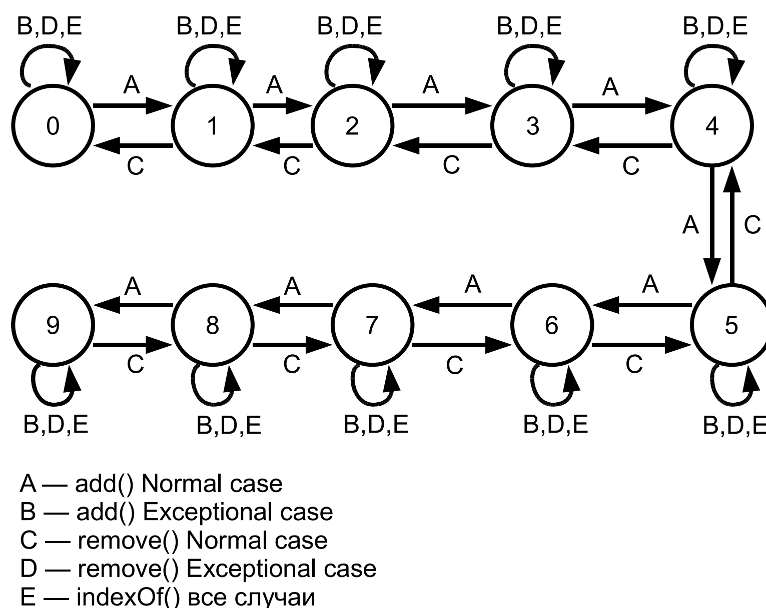


Рисунок 2: Граф состояний детерминированного конечного автомата, моделирующего список

Теперь мы имеем всю информацию, необходимую для определения тестового сценария.

1. В качестве основы теста используем спецификацию, записанную в виде спецификационного класса `ListSpecification`.
2. Для построения обхода используется обходчик `jatva.engines.DFSMExplorer`.
3. На основе спецификации и ситуаций, заданных в ней критерием помеченных путей, построен конечный автомат, обход которого гарантирует достижение полного покрытия по этому критерию.
    a. Состояние автомата (обобщенное состояние сценария) соответствует числу элементов в списке и может быть задано целым числом.
    b. Переходы автомата соответствуют вызовам методов класса `List`.

Более точно, переходы автомата соответствуют вызовам:

- метода `add()` с аргументами:
    - 0 и `new Object()` (для списка, в котором не более 8-ми элементов)
    - -1 и `new Object()`
- метода `remove()` с аргументами:
    - 0
    - -1

- метода `indexOf()` с аргументами:
  - `items[0]` (для непустого списка)
  - `new Object()`

Хотя все необходимое, чтобы написать сценарий, у нас имеется, для его разработки нам понадобится еще *медиатор*, разработка которого описана в следующем разделе. Поэтому здесь мы прервем рассказ о разработке сценария, который будет продолжен через один раздел.

Разработка медиаторов

Для того, чтобы спецификации можно было использовать в тестах, проверяя по ним поведение целевой системы, надо установить связь между специкационными классами и классами целевой системы.

Мы разработали класс `ListSpecification`, описывающий функциональность методов списка, представляемого в целевой системе любыми объектами интерфейса `List`. Если мы хотим протестировать поведение какого-то такого объекта на соответствие спецификациям, нужно связать его с объектом спецификационного класса. Для этого служат *медиаторы*.

Декларация медиаторного класса (назовем его `ListMediator`) содержит указание соответствующего спецификационного класса. Поместим медиаторный класс в тот же пакет, что и спецификационный. Поскольку спецификационный класс параметризован, то и медиаторный класс будет параметризован той же типовой переменной. В результате получается такая декларация:

```
package jatva.examples.list;

mediator class ListMediator<T> implements ListSpecification<T>
{
  ...
}
```

Медиаторный класс должен иметь ссылку на объект целевой системы. Именно над ним будут совершаться вызовы реализационных методов из операторов **branch** пост-условий спецификационных методов:

```
mediator class ListMediator<T> implements ListSpecification<T>

{
  implementation java.util.List<T> targetObject = null;
  ...
}
```

Медиаторный класс должен перегружать все спецификационные методы `ListSpecification`. В данном случае достаточно вызвать методы целевого объекта с теми же параметрами, с которыми вызывается и модельный метод. При этом не надо забывать про исключительные ситуации — они также должны присутствовать в сигнатуре медиаторных методов. Вызовы методов целевого объекта следует поместить в **implementation**-блок следующим образом:

```
mediator class ListMediator<T> implements ListSpecification<T>
{
  implementation java.util.List<T> targetObject = null;

  mediator void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    implementation
    {
```

```
      targetObject.add(i, o);
    }
  }

  mediator int indexOf(T o)
  {
    implementation
    {
      return targetObject.indexOf(o);
    }
  }

  mediator T remove(int i)
    throws IndexOutOfBoundsException
  {
    implementation
    {
      return targetObject.remove(i);
    }
  }
  ...
}
```

И, наконец, надо определить *процедуру синхронизации модельного состояния*, т.е. алгоритм построения состояния спецификационного объекта после того, как был вызван один из целевых методов.

В нашем примере состояние спецификационного объекта задается массивом `items[]`. Для задания процедуры синхронизации этого состояния с реализацией воспользуемся методами `get(int)` и `size()` интерфейса `java.util.List`, служащего нам в качестве типа реализационного объекта. Заметим, что при тестировании с помощью полученного медиатора мы полагаемся на корректность работы используемых при синхронизации состояний методов. Для проверки их правильности нужно подготовить отдельный тест.

Используя эти методы, код синхронизации состояния можно записать следующим образом.

```
items = (T[])new Object[targetObject.size()];
for(int i = 0; i < items.length; i++)
    items[i] = targetObject.get(i);
```

Каждый раз при выполнении этого кода, т.е. после обращения к любому целевому методу, массив `items` будет создаваться заново и заполняться теми объектами, которые возвращает метод `get(int)`. В медиаторном классе следует поместить код синхронизации в **`update`**-блоке так, как это приведено в итоговом коде медиаторного класса:

```
package jatva.examples.list;
mediator class ListMediator<T> implements ListSpecification<T>
{
  implementation java.util.List<T> targetObject = null;

  mediator void add(int i, T o)
    throws IndexOutOfBoundsException
  {
    implementation
    {
      targetObject.add(i, o);
    }
  }

  mediator int indexOf(T o)
  {
```

```
    implementation
    {
      return targetObject.indexOf(o);
    }
  }

  mediator T remove(int i)
    throws IndexOutOfBoundsException
  {
    implementation
    {
      return targetObject.remove(i);
    }
  }

  @SuppressWarnings("unchecked")
  update
  {
    items = (T[])new Object[targetObject.size()];
    for(int i = 0; i < items.length; i++)
      items[i] = targetObject.get(i);
  }
}
```

Теперь можно продолжить разработку сценария.

Разработка сценарных классов

Сценарные классы предназначены для задания тестовых сценариев, то есть для описания *тестовой последовательности*, выполнение которой обеспечивает решение некоторой задачи тестирования. Сценарные классы помечается модификатором **scenario**, который указывается в их объявлении после всех других модификаторов и до ключевого слова **class**. Назовем сценарный класс — `ListTestScenario` — и поместим его в тот же пакет, где находятся медиаторный и спецификационный классы. В результате получается следующая декларация:

```
package jatva.examples.list;
scenario class ListTestScenario
{
  ...
}
```

Внутри сценарных классов могут находиться объявления полей данных и методов. Но обязательно должно быть объявлено поле с типом спецификационного класса (в данном случае `ListSpecification`). Назовем его `objectUnderTest`. Поскольку спецификационный класс параметризован, нужно выбрать тип для его типового параметра. Пусть это будет `Object`:

```
scenario class ListTestScenario
{
  public ListSpecification<Object> objectUnderTest;
  ...
}
```

В качестве обобщенного состояния при построении конечного автомата было выбрано количество элементов массива `items` объекта `objectUnderTest`. Запишем это в виде метода , вычисляющего обобщенного состояние, но оформим специальным образом, а именно, в виде state-блока следующим образом:

```
scenario class ListTestScenario
```

```
{
  public ListSpecification<Object> objectUnderTest;
  state { return objectUnderTest.items.length; }
  ...
}
```

Сценарные методы предназначены для описания набора обращений к *целевой системе*, которые необходимо осуществить в каждом *обобщенном состоянии*, достижимом во время тестирования. Сценарные методы представляются как методы сценарного класса, не имеющие параметров, помеченные модификатором `scenario` . Тип возвращаемого значения указать нельзя — он предопределен и равен `boolean`: `true` – если проверка показывает, что целевая система ведет себя корректно, `false` – иначе. По сути сценарный метод позволяет написать дополнительную проверку к той, которую осуществляет постусловие спецификационного метода. В случае когда такая проверка не требуется, достаточно возвращать `true`. Назовем сценарные методы так же, как и спецификационные (тем самым будет достаточно просто отследить, какой спецификационный метод запускается в обобщенном состоянии). Начнем с метода `add()`. Согласно построенному конечному автомату в каждом обобщенном состоянии этот метод нужно вызвать дважды. В первый раз с параметрами -1 и `new Object()` и во второй раз — с параметрами 0 и `new Object()`. Иными словами, нужно вызывать метод `add()` с параметрами `i` и `new Object()` для всех `i` из интервала [-1, 1). Это можно записать с использованием оператора `iterate` следующим образом:

```
scenario class ListTestScenario
{
  public ListSpecification<Object> objectUnderTest;
  state { return objectUnderTest.items.length; }

  scenario add()
  {
    iterate(int i = -1; i < 1; i++)
    {
      objectUnderTest.add(i, new Object());
    }
    return true;
  }
  ...
}
```

Теперь учтем возможные исключительные ситуации при выполнении спецификационного метода. Для метода `add()` это `IndexOutOfBoundsException`. В сценарном методе его возникновение можно просто игнорировать, потому как проверка правильности его возникновения уже сделана в спецификационном методе:

```
  scenario add()
  {
    iterate(int i = -1; i < 1; i++)
    {
      try
      {
        objectUnderTest.add(i, new Object());
      }
      catch(IndexOutOfBoundsException e) { }
    }
    return true;
  }
```

Если этот сценарный метод вызовется в обобщенном состоянии 9, то добавление нового объекта будет сделано, хотя согласно конечному автомату на этом обобщенном состоянии добавлять элементы уже не надо. Добавим эту проверку в сценарный метод. Тем самым он

получит требуемый вид:

```
scenario add()
{
  if (objectUnderTest.items.length <= 9)
    iterate(int i = -1; i < 1; i++)
    {
      try
      {
        objectUnderTest.add(i, new Object());
      }
      catch(IndexOutOfBoundsException e) { }
    }
  return true;
}
```

Аналогично напишем код сценарного метода `remove()`. В нем проверять размер массива `items` не нужно, так как во время выполнения сценарного метода этот размер не может увеличиться.

```
scenario remove()
{
  iterate(int i = -1; i < 1; i++)
  {
    try
    {
      objectUnderTest.remove(i);
    }
    catch(IndexOutOfBoundsException e) { }
  }
  return true;
}
```

Перейдем к написанию сценарного метода `indexOf()`. Согласно конечному автомату в каждом обобщенном состоянии необходимо вызвать этот метод с аргументом `items[0]`, если массив `items` не пуст, и с аргументом **`new`** `Object()`, если пуст. Эти два случая можно записать в одном сценарном методе с использованием оператора **`iterate`**. В нем итерационная переменная будет принимать значения 1 и 2, что обозначает нужный вид параметра:

```
scenario indexOf()
{
  iterate(int i = 1; i < 3; i++)
  {
    objectUnderTest.indexOf
    (
      (i == 1 && objectUnderTest.items.length > 0)
        ? objectUnderTest.items[0]
        : new Object()
    );
  }
  return true;
}
```

Теперь напишем конструктор сценарного класса. Его цель — выполнить все операции по инициации выполнения сценария. К таковым операциям относятся фиксация тестового обходчика (в данном случае `jatva.engines.DFSMExplorer`) и создание начального состояния модельного объекта. Тестовый обходчик фиксируется вызовом метода `setTestEngine`. В качестве параметра ему следует передать объект тестового обходчика. Для создания начального состояния модельного объекта нужно задать медиаторный объект. Для задания медиаторного объекта нужно задать все его реализационные ссылки. В данном случае

реализационная ссылка одна - `targetObject` в классе `ListMediator`. Таким образом, конструктор сценарного класса будет иметь следующий вид:

```
public ListTestScenario()
{
  objectUnderTest = mediator ListMediator<Object>(
                          targetObject = new java.util.ArrayList<Object>()
                     );
  setTestEngine(new jatva.engines.DFSMExplorer());
}
```

Для возможности запуска тестового сценария необходимо написать метод `main`. Его можно написать как в отдельном классе, так и прямо в сценарном классе, что мы и сделаем. Следует создать объект сценарного класса и вызвать его метод `run` для запуска:

```
public static void main(String[] args)
{
  ListTestScenario myScenario = new ListTestScenario();
  myScenario.run();
}
```

Код готового сценарного класса имеет следующий вид:

```
package jatva.examples.list;

scenario class ListTestScenario
{
  public ListSpecification<Object> objectUnderTest;
  state { return objectUnderTest.items.length; }

  public ListTestScenario()
  {
    objectUnderTest = mediator ListMediator<Object>(
                            targetObject = new java.util.ArrayList<Object>()
                       );
    setTestEngine(new jatva.engines.DFSMExplorer());
  }

  scenario add()
  {
    if (objectUnderTest.items.length <= 9)
      iterate(int i = -1; i < 1; i++)
      {
        try
        {
          objectUnderTest.add(i, new Object());
        }
        catch(IndexOutOfBoundsException e) { }
      }
    return true;
  }

  scenario remove()
  {
    iterate(int i = -1; i < 1; i++)
    {
      try
      {
        objectUnderTest.remove(i);
      }
      catch(IndexOutOfBoundsException e) { }
    }
    return true;
  }
```

```
  scenario indexOf()
  {
    iterate(int i = 1; i < 3; i++)
    {
      objectUnderTest.indexOf
      (
        (i == 1 && objectUnderTest.items.length > 0)
          ? objectUnderTest.items[0]
          : new Object()
      );
    }
    return true;
  }

  public static void main(String[] args)
  {
    ListTestScenario myScenario = new ListTestScenario();
    myScenario.run();
  }
}
```

Теперь мы подготовили готовый к использованию тестовый набор, состоящий из спецификации, медиатора и сценария.

Работа с тестовым набором

Сборка тестового набора

Для полной сборки тестового набора из спецификаций, медиаторов и сценариев достаточно в среде программирования Eclipse выполнить команду **Build Project** для проекта, в котором располагается наш набор сценариев. При этом спецификации, медиаторы и сценарии будут оттранслированы в соответствующие компоненты тестового набора на Java.

В нашем примере достаточно выполнить эту команду для сценария `ListTestScenario`.

Выполнение теста

Для выполнения теста, входящего в тестовый набор, надо в среде программирования Eclipse выполнить команду **RunAs > JavaTESK Test** для тестового сценария, соответствующего этому тесту.

В нашем примере надо выполнить сценарий `ListTestScenario`.

Получение тестовых отчетов и анализ результатов тестирования

После окончания работы теста на одном уровне с узлом, соответвующим сценарию в окне среды Eclipse **Package Explorer**, появляется файл, соответствующий трассе выполненного теста. Для получения отчета с информацией об обнаруженных ошибках и достигнутом тестовом покрытии нужно выполнить команду **Generate Report** из контекстного меню для этого файла.

Полученный отчет говорит, что во время теста было обнаружено 9 нарушений.

Package/Namespace Summary Report - Mozilla Firefox

Package/Namespace Summary Report
generated: 12.12.2008 16:43:22

Overview
Failures (9)
j.examples.list
ListTestScenario
ListSpecification

jatva.examples.list

Summary | Failures

scenarios	failures	states/transitions			
ListTestScenario	9	10/58			
	9	**10/58**			
specifications	**failures**	**branches**	**marks**	**predicates**	**disjuncts**
ListSpecification	9	100% (6/6)	100% (14/14)	100% (14/14)	78% (15/19)
	9	**100% (6/6)**	**100% (14/14)**	**100% (14/14)**	**78% (15/19)**

Готово

Посмотрим на информацию о первой обнаруженной ошибке, выбрав в меню отчета пункты **Failures/failure 7**.

Failure Report - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us

Failure Report

- Overview
- Failures
  - failure 1
  - failure 2
  - failure 3
  - failure 4
  - failure 5
  - failure 6
  - failure 7
  - failure 8
  - failure 9
- j.examples.list
  - ListTestScenario
  - ListSpecification
    - add(int, java.lang.Object)
    - indexOf(java.lang.Object)
    - remove(int)

failure 7

Info

Location	
trace	ListTestScenario.1.utt, line 9473
Exception	
info	Postcondition violation
where	
Occurence	
scenario	j.e.list.ListTestScenario
state	3
transition	**jatva.examples.list.ListTestScenario.indexOf(** int i = 0 **)**
model object	[-]jatva.examples.list.ListMediator@197d257 { items : java.lang.Object[]@754fc [java.lang.Object@ea5461 { } , java.lang.Object@5c98f3 { } , java.lang.Object@cffc79 { }] }
specification method	j.e.l.ListSpecification.indexOf(java.lang.Object)
parameter value	java.lang.Object o = java.lang.Object@ea5461 { }
return value	int result = 0
oracle	int j.e.l.ListSpecification.indexOf(java.lang.Object)
after precondition	true
implementation object	java.util.ArrayList@1690726
implementation method	int j.u.List.indexOf(java.lang.Object)
parameter value	java.lang.Object ? = java.lang.Object@ea5461
return value	int result = 0
Test situation	
oracle	int j.e.l.ListSpecification.indexOf(java.lang.Object)
branch	ListContainsTheObject
mark	ListContainsTheObject
predicate	!(items.length == 0) && !(items.length == 1) && arrayContains(items, 0, items.length, o)

disjunct	pre	post	
	false		items.length == 0
	false		items.length == 1
	true		arrayContains(items, 0, items.length, o)
		true	objectsAreEqual(oldItems[indexOf], o)
		true	arrayContains(oldItems, 0, indexOf + 1, o)
			items.length == oldItems.length

Из описания можно увидеть, что данное нарушение является нарушением постусловия при вызове метода `indexOf()` с аргументом, который уже содержится в списке. При этом, согласно спецификации, ожидается, что формула `arrayContains(oldItems, 0, indexOf+1, o)` примет значение **`false`**, но в ходе теста она оказалась равна **`true`**. Просмотр описаний остальных обнаруженных нарушений показывает, что все они соответствуют в точности такой же ситуации. Более того, как показывает отчет о покрытии, все вызовы `indexOf()` с аргументом, который содержится в списке, приводят к такой ошибке.

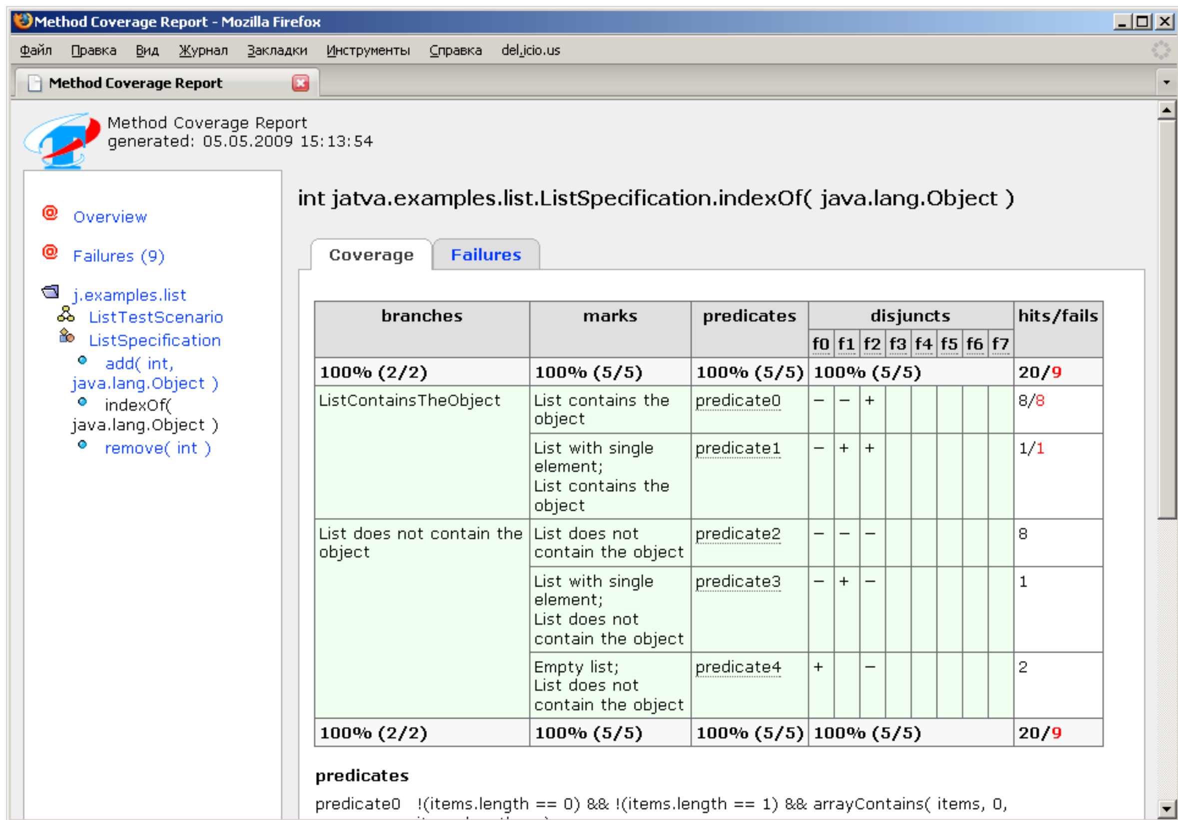


Рисунок 3: Покрытие метода indexOf() с данными о количестве нарушений

Несложный анализ выражения `arrayContains(oldItems, 0, indexOf+1, o)` показывает, что оно действительно должно всегда оказываться равно **true** в данной ситуации, поскольку при этом проверяется наличие объекта `o` в списке на позициях от 0 до результата вызова `indexOf()` *включительно*. Таким образом, обнаруженные нарушения выявляют ошибку в спецификациях — часть постусловия метода `indexOf()` в случае наличия аргумента этого метода в списке должна выглядеть следующим образом:

```
if(arrayContains(items, 0, items.length, o))
{
  branch ListContainsTheObject ( “List contains the object” );
  return    objectsAreEqual(oldItems[indexOf], o)
         && !arrayContains(oldItems, 0, indexOf - 1, o)
         && items.length == oldItems.length
         && arrayCompare(items, 0, oldItems, 0, items.length);
}
```

Исправив спецификации, скомпилировав их заново и запустив снова тест, мы получим результаты, уже не содержащие нарушений. В частности, отчет о покрытии метода `indexOf()` теперь выглядит так:

Method Coverage Report - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us

Method Coverage Report

- Overview
- j.examples.list
  - ListTestScenario
  - ListSpecification
    - add(int, java.lang.Object)
    - indexOf(java.lang.Object)
    - remove(int)

int jatva.examples.list.ListSpecification.indexOf(java.lang.Object)

Coverage

branches	marks	predicates	disjuncts								hits/fails
			f0	f1	f2	f3	f4	f5	f6	f7	
100% (2/2)	**100% (5/5)**	**100% (5/5)**	**100% (5/5)**								**20**
ListContainsTheObject	List contains the object	predicate0	–	–	+						8
	List with single element; List contains the object	predicate1	–	+	+						1
List does not contain the object	List does not contain the object	predicate2	–	–	–						8
	List with single element; List does not contain the object	predicate3	–	+	–						1
	Empty list; List does not contain the object	predicate4	+		–						2
100% (2/2)	**100% (5/5)**	**100% (5/5)**	**100% (5/5)**								**20**

predicates

predicate0 !(items.length == 0) && !(items.length == 1) && arrayContains(items, 0,

Method Coverage Report - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us

Method Coverage Report

- Overview
- j.examples.list
  - ListTestScenario
  - ListSpecification
    - add(int, java.lang.Object)
    - indexOf(java.lang.Object)
    - remove(int)

void jatva.examples.list.ListSpecification.add(int, java.lang.Object)

Coverage

branches	marks	predicates	disjuncts											hits/fails
			f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	
100% (2/2)	**100% (4/4)**	**100% (4/4)**	**66% (4/6)**											**99**
Exceptional case	Exceptional case	predicate0	–	–	+									0
			–	+										8
	Empty list; Exceptional case	predicate1	+	–	+									0
			+	+										1
NormalCase	Normal case	predicate2	–	–	–									88
	Empty list; Normal case	predicate3	+	–	–									2
100% (2/2)	**100% (4/4)**	**100% (4/4)**	**66% (4/6)**											**99**

predicates

predicate0 !(items.length == 0) && (i < 0 || items.length < i)

predicate1 items.length == 0 && (i < 0 || items.length < i)

predicate2 !(items.length == 0) && !((i < 0 || items.length < i))

predicate3 items.length == 0 && !((i < 0 || items.length < i))

prime formulas

f0 items.length == 0

f1 i < 0

f2 items.length < i

f3 thrown == null

Отчет о достигнутом покрытии для метода `add()` содержит указание на то, что во время тестирования было достигнуто 100%-покрытие по критерию маркированных путей (столбец marks содержит «100%»). Столбец disjuncts означает критерий покрытия по дизъюнктам. По этому критерию было достигнуто всего лишь 66%-покрытие, о чем свидетельствуют две строки с розовыми квадратами из шести. Каждая розовая часть строки разбита на столбцы, соответствующие простейшим формулам (не содержащие конъюнкций и дизъюнкций). Простейшие формулы, встречающиеся на маркированном пути, помечены «+» (если формула истинна) или «-» (если формула ложна). Одному маркированному пути могут соответствовать разные варианты «+» и «-»: например, при вычислении конъюнкции нет смысла вычислять второй конъюнкт, если первый имеет ложное значение — конъюнкция при любом втором конъюнкте будет также ложной. На странице отчета выписаны все возможные варианты вычисления простейших формул, за исключением невозможных случаев. Розовая строка означает, что данный вариант вычисления простейших формул не был реализован при тестировании. А именно в данном сценарии не были реализованы следующие случаи:

- !f0 && !f1 && f2
- f0 && !f1 && f2

Оба случая можно объединить в один – !f1 && f2, поскольку для f0 значение должно быть любым. Судя по определениям простейших формул, в сценарии для метода `add()` дополнительно нужно реализовать ситуацию `!(i < 0) && (items.length < i)`. Иными словами, необходимо в сценарии вызвать метод с параметром, большим длины списка. Это можно сделать с помощью такого сценарного метода:

```
scenario add()
{
  if ( objectUnderTest.items.length <= 8 )
    iterate(int i = -1; i <= objectUnderTest.items.length + 1; i++ )
    {
      try
      {
        objectUnderTest.add( i, new Object() );
      }
      catch( IndexOutOfBoundsException  e ) { }
    }
  return true;
}
```

После запуска обновленного теста отчет о покрытии метода `add()` является полным по всем критериям:

Method Coverage Report - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us

Method Coverage Report

Overview
j.examples.list
ListTestScenario
ListSpecification
add(int, java.lang.Object)
indexOf(java.lang.Object)
remove(int)

void jatva.examples.list.ListSpecification.add(int, java.lang.Object)

Coverage

branches	marks	predicates	disjuncts											hits/fails
			f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	
100% (2/2)	**100% (4/4)**	**100% (4/4)**	**100% (6/6)**											**108**
Exceptional case	Exceptional case	predicate0	–	–	+									8
			–	+										8
	Empty list; Exceptional case	predicate1	+	–	+									1
			+	+										1
NormalCase	Normal case	predicate2	–	–	–									88
	Empty list; Normal case	predicate3	+	–	–									2
100% (2/2)	**100% (4/4)**	**100% (4/4)**	**100% (6/6)**											**108**

predicates

Аналогичные изменения следует сделать со сценарием для метода `remove()` для получения полного покрытия по всем критериям:

Method Coverage Report - Mozilla Firefox

Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us

Method Coverage Report

Overview
j.examples.list
ListTestScenario
ListSpecification
add(int, java.lang.Object)
indexOf(java.lang.Object)
remove(int)

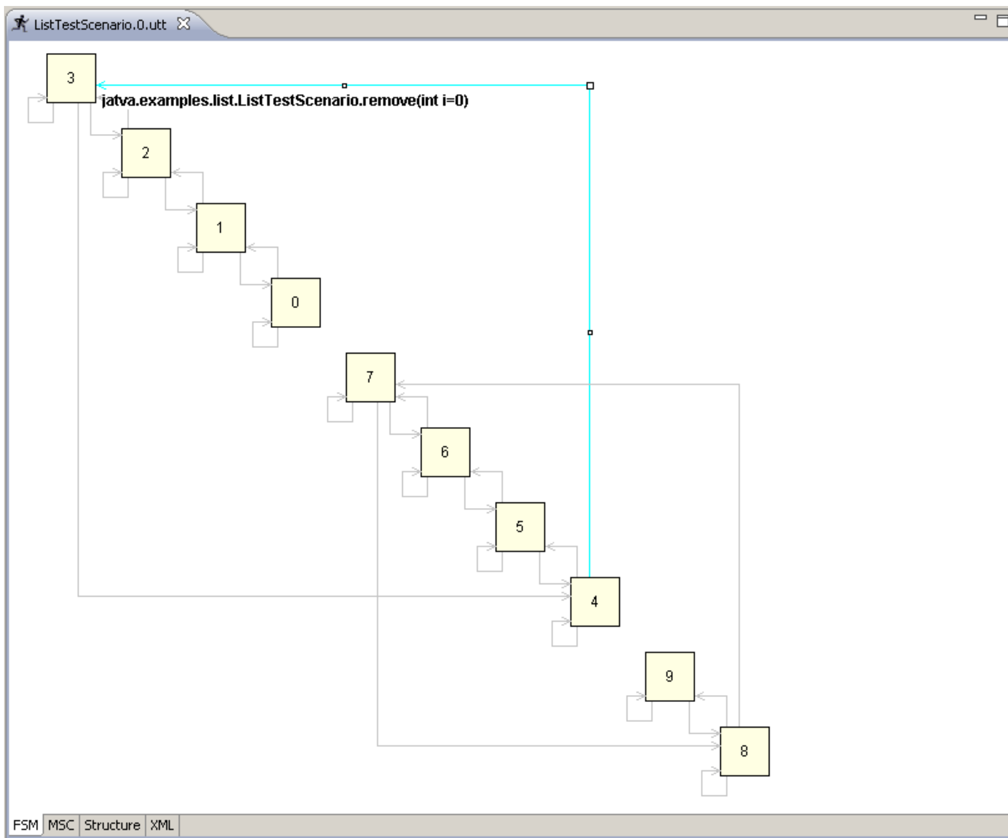
java.lang.Object jatva.examples.list.ListSpecification.remove(int)

Coverage

branches	marks	predicates	disjuncts												hits/fails
			f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	
100% (2/2)	**100% (5/5)**	**100% (5/5)**	**100% (8/8)**												**110**
Exceptional case	Exceptional case	predicate0	–	–	–	–									8
			–	–	+										8
	List with single element; Exceptional case	predicate1	–	+	–	–									1
			–	+	+										1
	Empty list; Exceptional case	predicate2	+		–	–									1
			+		+										1
NormalCase	Normal case	predicate3	–	–	–	+									88
	List with single element; Normal case	predicate4	–	+	–	+									2
100% (2/2)	**100% (5/5)**	**100% (5/5)**	**100% (8/8)**												**110**

predicates

Помимо отчета о нарушениях, обнаруженных в ходе теста, и достигнутом покрытии мы можем, воспользовавшись пунктом контекстного меню **Open** для узла, соответствующего трассе теста, открыть графический отчет о ходе теста.


ListTestScenario.0.utt
jatva.examples.list.ListTestScenario.remove(int i=0)
3
2
1
0
7
6
5
4
9
8
FSM
MSC
Structure
XML