Functional testing of list using JavaTESK

Contents

UniTESK description

Developed software must be correct and reliably. Testing is the best-known and widely used way for assure it. But traditional methods of testing is not applicable for the modern software because it becomes huge. Testing of small program ($\sim< 10^4$ lines of code) is possible by regular ways but its efficiency is reduced when program grows. UniTESK is technology for test developing. It can solve this problem. It decreases laboriousness and increases the testing quality for modern complex software.

The main aspects of UniTESK are:

- Tests are developed on basis of functional model (*formal specification*), but not on structure of code. So test developing may be started earlier than software is developed.
- *Test coverage criteria* are built automatically from formal specifications. Criteria define *test fullness criteria* on basis of requirements structure. These criteria are used for evaluating of testing quality.
- *Test scenario* are built from specifications and chosen coverage criterion. Test scenario are targeted to the maximum coverage by the criterion. UniTESK test scenario are similar to the test scripts, but test scenario are more expressive and provide more qualitative testing with the same efforts.
- UniTESK supports “black-box” testing.
- Formal specifications can be used for testing of different versions of the same software, even these interfaces are different. It economizes efforts of developing specific tests for this software. Special layer between tests and implementation (*mediators*) provides this capability.
- Formal specifications, mediators and test scenario are developed on the extension of the programming language of tested software. It simplifies studying of technology by testers and makes more clear relations between tests and tested software. There is a tool supported UniTESK for Java.

The main actions for test developing according to UniTESK:

1. *Formal specification developing*. It bases on the analysis of functional requirements and knowledge of developers.
2. *Coverage criteria extraction*. Try to formulate requirements for test quality («enough full testing», when it may be stopped) based on customer wishes, knowledge about application domain and project resources. Reformulate these requirements as requirements for coverage according to coverage criteria using specification structure.
3. *Test scenario developing*. It bases on specifications and doesn't relate with software implementation or its specific version. Test scenario must provide achievement of the requiring coverage.
4. *Mediators developing*. The aim is binding test with specific system implementation. Interface is enough for this step, full implementation may not be ready yet.
5. *Test system compilation*. It is necessary to automatically compile specifications, mediators and scenario from programming language extension to programs on the same programming language.
6. *Test running*. This is possible only after compilation. First time will be spent to the test

debugging.

7. *Testing results analysis.* Aims are error detecting and collecting coverage (if it is not enough, new scenario can be useful).

Test scenario and mediators can be developed simultaneously since their dependencies are weak. In addition since errors of testing system any step may be done again.

Testing aims and bounds detection

It is necessary to answer the following questions:

1. What parts, subsystems, components of target system will be tested?
2. What are functional requirements of this testing?

Answers depend on the following aspects:

- Application domain context: what requirements are put forth these systems, available documents and knowledge about application domain, context of target system using, wants of users and other persons, requirements of governing organizations and standards, solved problems by system, possible ways of solution these problems and possible structure corresponding components of system.
- Target system architecture, how much is it known at the beginning of the work: system division into components, problems solved by different components, and possible interactions between components.
- Current project context: what resources (people, time, money, hardware and software) are available, what are customer's requirements to the system and its testing (also system users' requirements, developers' requirements, governing organizations' requirements, etc.).
- Related projects context: what another projects are related or will be possible related from the target system and results of this project, what requirements to the system quality or to the testing quality of the system.

The rest of the document is devoted to tests developing for collection framework of Java 5. Its classes are located in package `java.util`.

Sources of the target system and requirements to their functionality:

- documentation
- application domain experts (experts of JDK)
- standards concerning to the application domain (requirements of Java specification, patterns of programming)
- architectures, designers and developers of the system

We have standard documentation of JDK 1.5. The following consists of brief description of the classes and interfaces from `java.util`, dealing with collections:

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.

As result, there are 13 interfaces, 25 classes, and 3 exceptions.

Repeat two corner questions:

1. What components will be tested?
2. What requirements to its testing?

Abstraction (or *generalization*) is useful way for decreasing efforts of test developing. Abstraction is extraction common functionality of the different components, common requirements to them. Abstraction is useful for the answer of the second question. The common way of common functionality testing is developing one test. This test deals with different components of the target system through the simple intermediate components — *mediators*.

- Testing of exceptions is testing their constructors (without parameters and with one string-parameter) and methods inherited from `java.lang.Throwable`. One specification is enough, each exception will be tested with specific mediator. If there is such test for `java.lang.Throwable`, it can be used for testing without additional development.
- Few classes have almost similar functionality but their behaviors differ in multithreading environment. Tests for sequential requests to their methods can be similar. It is not necessary to develop special tests for their behavior in multithreading environment because correct behavior is supported by Java.
- It is not necessary to develop tests for abstract classes, if these classes implement basic functionality for interfaces (`AbstractCollection`, `AbstractList`, `AbstractMap`, `AbstractSet`). Whole their functionality can be tested by tests for interfaces.

Further details may be appeared after methods inspection. For example, it is possible that testing of whole functionality of `ArrayList` is not necessary within project (testing of constructor `ArrayList(int)` and methods `void ensureCapacity(int)` and `void trimToSize()` is not necessary).

The rest of `ArrayList` may be tested by tests for interface `List` because it is methods for list processing.

Thereby it is possible to clearly define set of classes for testing and approximate behavior of tests.

We delay accurate detection of requirements to the testing quality before accurate detection of test aims, i.e. before specification development.

Specification development

Lets it is necessary to develop tests for `ArrayList`. Lets testing methods from interface `List` is enough. According to UniTESK, the first step is definition of interface functionality, i.e. development of formal specifications.

Specifications used in UniTESK describe a structure of target components by describing its fields and *invariants* – additional constraints on the data of correct component. Description should contain only fields for explanation component works. Description should contain also *preconditions* and *postconditions* for all component methods. A precondition defines conditions when method can be called. A postcondition defines constraints to the results of method which must be true if method is correct. The following table contains descriptions methods of `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.

Lets the system under test is selected methods group (i.c. all methods from `java.util.List`).

It's necessary for specification development to define the following:

1. State of the system, which is enough to describe the reaction of the system on any method invocation.
2. Signatures for each method: clear name, parameters types, type of return value, exceptions.
3. Preconditions for each method – it means the situations when method's behavior is defined and method can be called.
4. Postconditions for each method – it means constraints on the results of the correctly invoked method.

Consider these aspects in details.

1. State of the system is defined by information about method's invocation history for possibility of describing results of method invocation with any arguments. In our case elements of list and their order are enough for full description of methods in `java.util.List`.
2. It's necessary to define a signature of the specification method for each method of the system

under test. Don't forget the following:

   a. Types of parameters, results and exceptions may be differ from the correspond types of the target system.

   b. List of exceptions must consist of all exceptions which occur in regular conditions including descendants of `RuntimeException` and `Error`. Term «Regular conditions» means fulfillment of precondition and maybe extra conditions on resources about testing modes, e.g. large amount of memory availability.

3. It's necessary to define a precondition of the specification method for each method of the system under test. The first step is explication of situations (state of the system a,d parameters of methods) which are not intended for method. Method's behavior is not defined in these situations, method invocation may cause to unpredictable consequences up to the target system crash. In our case method invocation shouldn't cause to system crash, so its precondition should be true always. Specification method may report about incorrect values of arguments (it can't process these arguments by common rule) by special return value or throw an exception. Not all preconditions may be the weakest. For example, system of memory managing may require link to the allocated (by special operation) object for memory dispose because its high efficiency. These constraints are emphasized especially in documentation and presented always at operations especially for limited amount of developers. In our case preconditions of all methods `java.util.List` should be true because these are common use interface.

4. It's necessary to define a postcondition of the specification method for each method of the system under test (i.e. define method's behavior for any system state and any arguments). Postcondition describes constraints on the results of correctly executed method. It can be explicated from documentation or another sources with method requirements. We have enough documentation for methods of `java.util.List`. Consider few methods.

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()).

Suppose that our tests will be expected to the common functionality of list without restrictions to the object classes and stored objects itself. In addition suppose that all operations are supported by list. It means that exceptions `UnsupportedOperationException`, `IllegalArgumentException` are impossible in our tests. Similar assumptions narrow resulting tests' domain of applicability a bit.

Begin a specification development.

Declaration of the specification class for list looks in the following way. Specification class must have a type parameter because specification is intended for lists of elements of the same type.

```
package jatva.examples.list;

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

State of list is fully described by enumeration its elements with the same order. This state should be arrange as one or many fields of specification class. Use an array of objects for it.

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

Consider creation a specification method for method `add()` of list.

Use the same name for specification method. Don't change types of parameters, type of method result and exceptions because they don't take part in specified system:

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

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

Preconditions of all methods must be true:

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

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

But this precondition may be omitted because absence of precondition is the same as its identical true.
Describe constraints on the method result in regular situation, i.e. after execution without exceptions. Object (from the second argument) must be inserted to the i-th position into the list, all elements before i-th must be remained at their positions, and all elements after i-th must be shifted to the 1 position right. Use temporary copy source list and check list after addition. Create a special method `arrayCompare` for comparing list ranges. Create a special method for comparing objects, when each object may be **`null`**.

```
  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);
  }
}
```

Operator `pre` is used for getting a copy of `items` before target method invocation. Method `clone()` is invoked because `items` is reference to the object but not a value. Operator `pre` has lesser priority that method invocation, so `clone()` returns a copy of array before method invocation. In case of non-object type of `a`, expression `pre a` returns value of `a` before method invocation.

Expression `thrown == null` may be used to check exceptions absence while method is invoked. Keyword `thrown` means a reference to the created exception.

Add constraints about exceptions to the postcondition. Exception `IndexOutOfBoundsException` must be occurred when `i` is out of correct positions range, list mustn't be changed.

```
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 method for `indexOf()` can be created similarly:

```
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);
      }
    }
  }
```

Specification contains additional constraint on the method result using special variable which name is the same with name of method.

The following consists of specifications for all methods.

```
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);
```

```
        }
      }
    }
  }
```
It's necessary to write a constructor for initialization of the model state (fields of specification class). In our case `items` should be assigned with an empty array in the following way:
```
  @SuppressWarnings("unchecked")
  public ListSpecification()
  {
    items = (T[])new Object[0];
  }
```
Now we have specifications represented formalized requirements to the target system. This representation is useful for testing automation.

Requirements refinement to the quality of testing

This stage is intended to definition a quality of testing for each tested component, subsystem, class and method.

Quality of testing is measured by provided *coverage* of different situations occurred when system works. The first step is explication of full situations set (*coverage targets*) based on some *coverage criterion*. The next is calculating occurred situations when testing worked. So measure of quality of testing is ratio from this amount to the amount of all situations (*coverage percent*).

Coverage criteria may be divided into the groups:

- *Structural criteria*. These criteria are defined coverage targets based on the testing system structure, i.e. system architecture, internal organization of components, structure of code. For example, coverage targets can be methods of the system under test, measure is percent of invoked methods. Another example is coverage of lines of code, coverage targets is execution all lines of code of the system under test.
- *Functional criteria*. These criteria are defined coverage targets based on the structure of requirements to the system. For example, coverage targets may be certain requirements, and quality of testing is measured by ratio from count of tested requirements to the count of all requirements.

UniTESK is intended to the development functional tests aimed to the high quality on functional coverage criteria. These criteria are defined on the developed specification structure. It is possible because specifications are developed proceeding from requirements and they are formalized representation of requirements. Further, development of these tests can be automated easily, because formal specifications can be processed automatically.

Consider specification of method `add()` once more.
```
  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);
        }
      }
    }
```

The postcondition has 2 branches with appreciably different requirements on the method results. So these 2 situations are necessary for testing. These branchings in postconditions with different constraints on the result of the tested method are named as branches of functionality. They are marked by branch operator. Further occurrence conditions for the first or the second branch are depend on state of list and arguments only. All operators in a postcondition before `branch` are executed before target method's invocation, and all operators after `branch` are executed after invocation. So all identifiers before `branch` means values of fields, parameters, etc. before target method invocation. Operator `pre` may be used after `branch` only. Operator `pre` is not necessary after addition of `branch` for our specification. Each operator `branch` would have a description in brackets:

```
  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);
      }
    }
  }
```

These branches of functionality set functional coverage criteria naturally. If type of a target method's result is not `void`, a postcondition should have constraints on method's result after `branch` by variable with the same name as specification method's name. Resulting specification can be the following:

```
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);
      }
    }
  }
}
```

Now we can require coverage percent which tests will provide (for example, 100% of branches). All branches of functionality for one class set *coverage criteria of branches of functionality.*

The next step is analysis of conformance 100% coverage of branches with requirements to testing from customers, developers and other persons. Missing situations can be defined explicitly in specifications by referencing with some marks. Lets it's necessary to test methods `indexOf()` and `remove()` on lists with 1 element. Each additional situation can be defined by operator **`mark`**. Lets insert it to the beginning of the postcondition (the rest of code is preserved):

```
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";
      ...
    }
  }
}
```

Marks introduced by **`mark`** define *coverage criterion of marked paths*. Situations of this criterion correspond to different combinations of **`mark`** and **`branch`** from one execution of postcondition. Situations for our case are collected in the following table:

Method	**`add()`**	**`indexOf()`**	**`remove()`**
list is empty	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
list has 1 element only	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
list has more than 1 element	Exceptional case	List contains the object	Exceptional case
	Normal case	List does not contain the object	Normal case

Table 1: Test situations for coverage criteria of marked paths

Selected cells correspond to impossible situations — empty list doesn't contains elements and there isn't parameter for empty list which more or equal than 0 but less than it length which equals to 0.

The second situation will be ignored automatically because it corresponds to identically false formula on integers: `items.length == 0`, `!(i < 0)`, `!(i >= items.length)`.

The first situation corresponds to only human-understandable constraints: `items.length == 0`, ar-

`rayContains(items, 0, items.length, o)`. So it's necessary to add a *tautology* with this constraints for ignoring the first situation. The final specification for `indexOf()` is the following:

```
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);
    }
  }
}
```

Now specification development is fully completed and testing aim is defined strongly — full coverage achievement on criteria of marked paths.

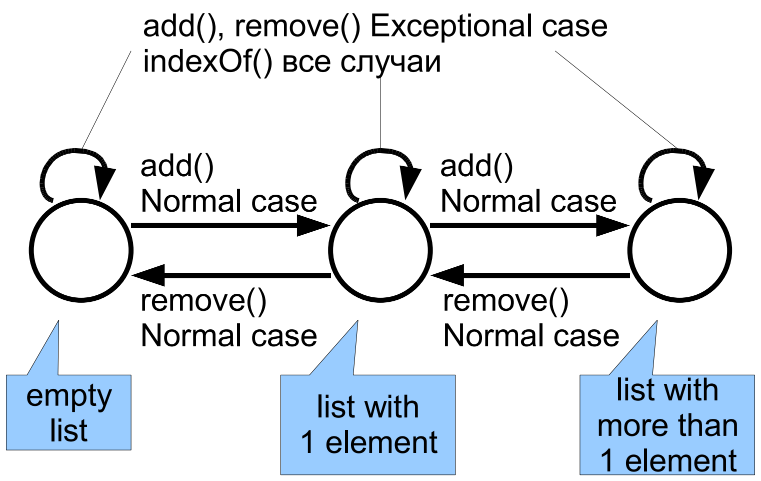
Scenario classes development

Test scenario of UniTESK sets a construction of tests sequence. If method's behavior depends on arguments only, test scenario contains a sequence of this method invocations provided required coverage. If method's behavior depends on history of target system processing, test scenario should observe some states of the target system. To solve this problem a test scenario describes briefly *finite state machine* modeled the target system's behavior. Test scenario executes all acceptable methods in each occurred state. And test scenario should provide high coverage percent on selected coverage criterion, full coverage in ideal case.

Lets consider different situations in `ListSpecification` for criterion of marked paths. These situations are collected in the table 1. It's necessary to define a finite state machine (FSM) for test scenario. Traversal of this FSM must cover all these situations. The FSM can be specified by the following aspects:

1. FSM states (*generalized scenario states*). In our case states can be constructed from situation parts not depended from method parameters. So FSM will have 3 states: empty list, list with 1 element, and list with more than 1 element.
2. FSM transitions. They can be constructed by meaning of acceptable methods and arguments for each state. In our case it's necessary to cover 2 situations in each generalized state: normal and exceptional for `add()` and `remove()`, situations for `indexOf()` differ by containing object-parameter in the list. So it's enough to define 2 transitions in each state for each meth-

od using different values of parameters. 0 is enough as parameter for `add()` in `Normal case` situation, -1 is enough for `Exceptional case` situation. The same parameters are suitable for `remove()`. `items[0]` is enough as parameter for `indexOf()` in `List contains the object` situation with the exception of empty list situation. `new Object()` is enough as parameter for `List does not contain the object` situation. Resulting graph are in the picture 1.



Picture 1: Graph for list operations

Besides FSM description it's necessary to define an algorithm for FSM traversal. This algorithm implemented by some *engine*. An engine is a library component of JavaTESK. Selected engine must be able to deal with constructed FSM.

We select `jatva.engines.DFSMExplorer` because it deals with deterministic strongly connected FSM.

Strongly connected FSM is FSM with ability to transfer from any state to any state by some transitions only. Out FSM is strongly connected.

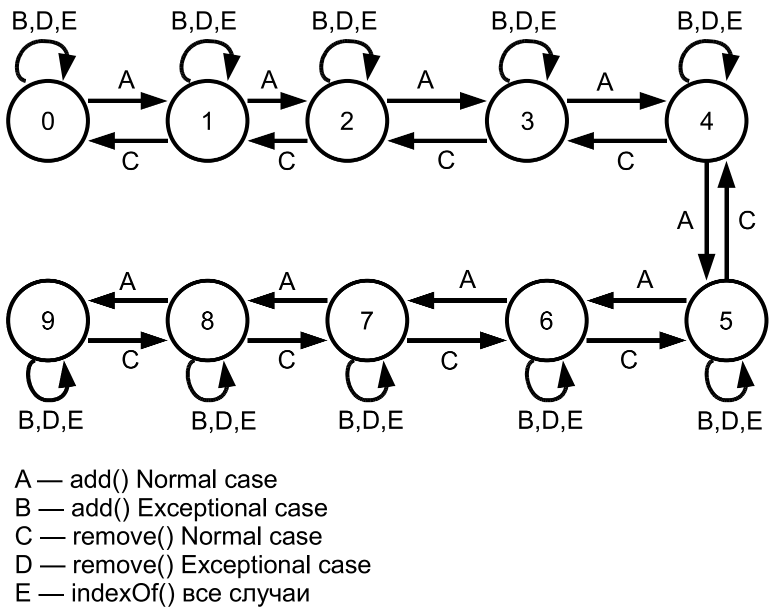
In *deterministic FSM* invocation of any operation from any state defines transition from this state unambiguously. Destination state mustn't depend on way to reach source state.

Our FSM is not deterministic — `remove()` invocation with parameter around the bound of list from «list with more than 1 element» state can stay at the same state (if list has more than 2 elements), and transfer to the «list has 1 element only» state (if list has 2 elements only).

UniTESK requires to improve FSM to make it deterministic. It's can be done by splitting states contained non-determinism until FSM becomes deterministic.

In our case “list with more than 1 element” state should be split into 2 states: «list with 2 elements only» and «list with more than 2 elements». But resulting FSM is non-determinism yet because of `remove()` invocation from state «list with more than 2 elements». Splitting it creates the following states: «list with 3 elements only», «list with 4 elements only», etc. — *infinity FSM*, each state corresponds to length of the list.

So FSM should be limited, for example, by forbidding to add new elements to the list if it has 9 elements already. Corresponding graph is at the picture 2.



Picture 2: Graph of the deterministic FSM modeled list

Now we have enough information for test scenario building:

1. Use specification class `ListSpecification`.
2. Use `jatva.engines.DFSMExplorer` as test engine.
3. Made up FSM ensured full coverage by criteria of marked paths.
    a. States (generalized scenario states) correspond to count of list's elements and can be specified by integer value.
    b. Transitions correspond to invocations of `List`'s methods.

More clearly, FSM transitions correspond to invocations of:

- method `add()` with arguments:
    - 0 and **new** `Object()` (for lists with more than 8 elements)
    - -1 and **new** `Object()`
- method `remove()` with arguments:
    - 0
    - -1
- method `indexOf()` with arguments:
    - `items[0]` (for non-empty list)
    - **new** `Object()`

It's necessary to develop a new component of testing system called *mediator*. The next section is devoted to developing it. Scenario development will continue after the next section.

Mediators development

We developed class `ListSpecification` to describe functionality of the target system presented by objects of interface `List`. Using of specifications is impossible without connection between specification and the system under test. Connection is provided by special classes called *mediators*.

Declaration of mediator class (lets give it name `ListMediator`) contains name of specification class. Lets put mediator class to the same package with specification class. Mediator class has a type parameter because the specification class has a type parameter. These type parameters must be equal:

```
package jatva.examples.list;

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

Mediator class must have a reference to the target system object. Target methods will be invoked by this object from operators `branch` of postconditions:

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

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

Mediator class must override all specification methods from `ListSpecification`. Mediator methods implement specification methods by target method's invocation. So in our case mediator methods will invoke (and return value) target methods with the same parameters as specification method invoked. Don't forget about exceptions — they should be in mediator method's signatures. Target methods' invocations should be placed in `implementation`-block:

```
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);
    }
  }
  ...
}
```

The next step is making up *procedure for model state synchronization*. This procedure aims to build model object state after invocation of target method.

In our case state of model object is presented by `items[]` array. Use methods `get(int)` and `size()` from interface `java.util.List`. We have to rely on correctness of these methods. Their testing can be performed by separate process.

Using these methods, synchronization can be the following:

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

Each execution of this code (i.e. after any target method's invocation) caused to create a new array `items` and fill it by objects returned from `get(int)`. Procedure for model state synchronization should be placed in **`update`**-block:

```
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);
  }
}
```

Now we may continue to develop test scenario.

Test scenario development

Scenario classes are intended to test scenario, i.e. to define *test sequence*. Execution of a test sequence performs required testing process. Scenario classes are marked by **`scenario`** modifier after other modifiers and before **class** keyword. Lets scenario class has name `ListTestScenario`. Put it to the same package with mediator and specification classes:

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

Scenario class can have fields and methods definitions. It's necessary to define a field of specification class type (in our case with `ListSpecification` type). Lets it has name `objectUnderTest`. It's necessary to choose type for specification class actualization. Lets choose `Object`:

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

Later generalized FSM state was built to count of `items` 's elements from object `objectUnderTest`. Procedure of generalized state calculation are written by **state**-block:

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

Scenario methods are intended to define a sequence of accesses to the target system from each reachable generalized state. Scenario methods look like methods of scenario class with modifier **scenario** without parameters. It's impossible to specify a return type — it's predefined and equal to **boolean**: **true** – if checking shows a correctness of the target system, **false** – otherwise. Scenario method provides addition an extra check to postcondition. If this extra check is not necessary, it's enough to return **true**. Lets get names to scenario methods as the same with specification methods (it makes easier to watch an invoked specification method from generalized state). Lets begin from method `add()`. This method must be invoked twice from each generalized state. Parameters for the first invocation are -1 and **new** `Object()` , and for the second invocation are — 0 and **new** `Object()`. So method `add()` must be invoked with parameters i and **new** `Object()` for each i from {-1, 0} set. These invocations can be coded by operator **iterate** :

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

  scenario add()
  {
    iterate(int i = -1; i <= 0; i++)
    {
      objectUnderTest.add(i, new Object());
    }
    return true;
  }
  ...
}
```

Lets remember possible exceptions from specification methods. This exception for method `add()` is `IndexOutOfBoundsException`. Scenario method's not need to check correctness of throwing this exception because this check is performed by specification method:

```
  scenario add()
```

```
{
  iterate(int i = -1; i <= 0; i++)
  {
    try
    {
      objectUnderTest.add(i, new Object());
    }
    catch(IndexOutOfBoundsException e) { }
  }
  return true;
}
```

If scenario method is invoked from 9th generalized state (with 9 elements), addition of element will be performed but FSM prohibits this addition. So add checking of elements' count to the scenario method:

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

Scenario method `remove()` can be created similarly. It's not necessary to check elements' count because it can't be increased.

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

Lets create scenario method `indexOf()`. This method should be invoked from each generalized state with argument `items[0]`, if array `items` is not empty, and with argument **`new`** `Object()`, otherwise. These cases can be coded by operator **`iterate`**. Iteration variable of this operator has value 1 or 2 to set required kind of argument:

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

Lets create constructor of scenario class. Its aim is initialization operations for the scenario: test engine setting (in our case `jatva.engines.DFSMExplorer`) and initial model state creation. Setting of test engine can be performed by `setTestEngine` method with test engine's object as parameter. It's necessary to create mediator object for model state creation. Mediator object is created with creation of this implementation fields. Mediator `ListMediator` has 1 implementation field - `targetObject`:

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

It's necessary to create method `main` for scenario launch. This method may be placed in separate class or in scenario class (we put it to the scenario class). Method can create scenario class's object and invoke its method `run` for launching:

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

Prepared scenario class should have the following code:

```
package jatva.examples.list;

scenario class ListTestScenario
{
  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 <= 0; i++)
    {
      try
      {
        objectUnderTest.remove(i);
      }
      catch(IndexOutOfBoundsException e) { }
    }
    return true;
  }

  scenario indexOf()
  {
    iterate(int i = 1; i <= 2; 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();
  }
}
```
We have prepared test set with specification, mediator and scenario.

Test set processing

Building

Building of test set can be performed in Eclipse by command **Build Project** for project with developed classes. Specifications, mediators and scenario will be translated to the corresponding components of the test set in Java.

We can do it for `ListTestScenario` scenario.

Test execution

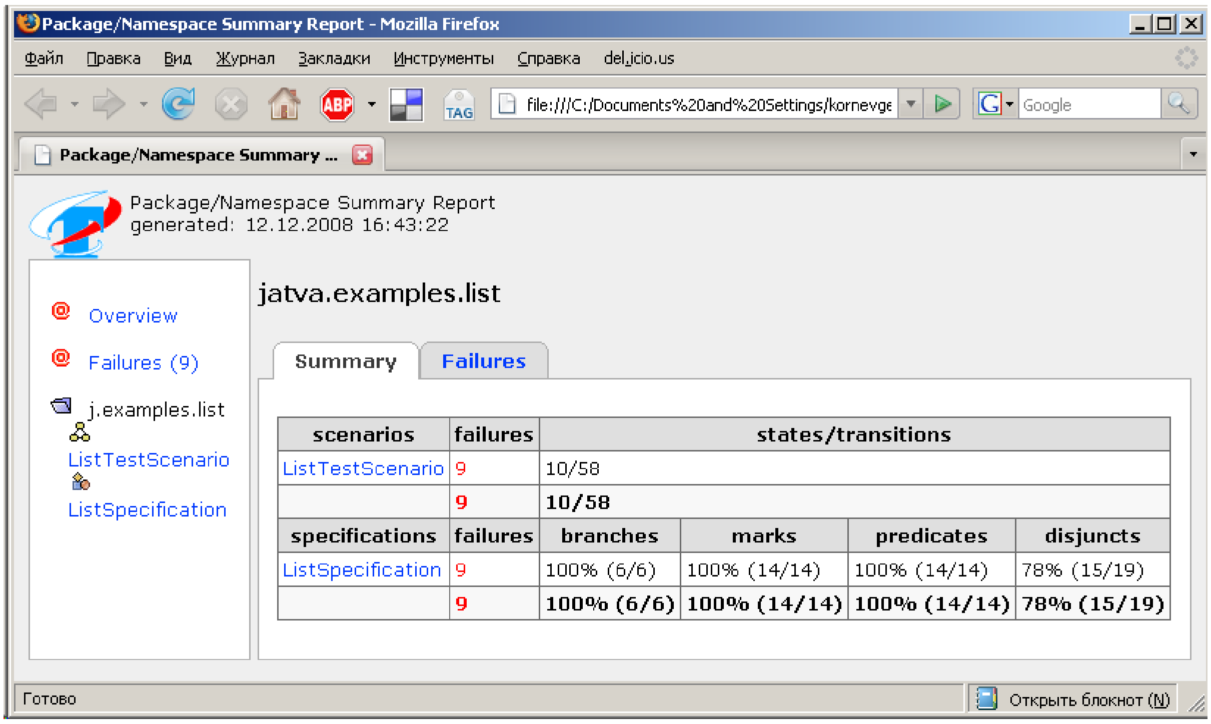
Test execution can be performed in Eclipse by command **RunAs > JavaTESK Test** for corresponding scenario class.

We can execute test for scenario `ListTestScenario`.

Test results analysis

After test execution new file will appear in **Package Explorer** window near the scenario node. This new file will have special icon (running human figure). This file corresponds to the trace of executed test. Creating of report about testing can be performed automatically by command **Generate Report** from the context menu of this file.

Report shows 9 failures.

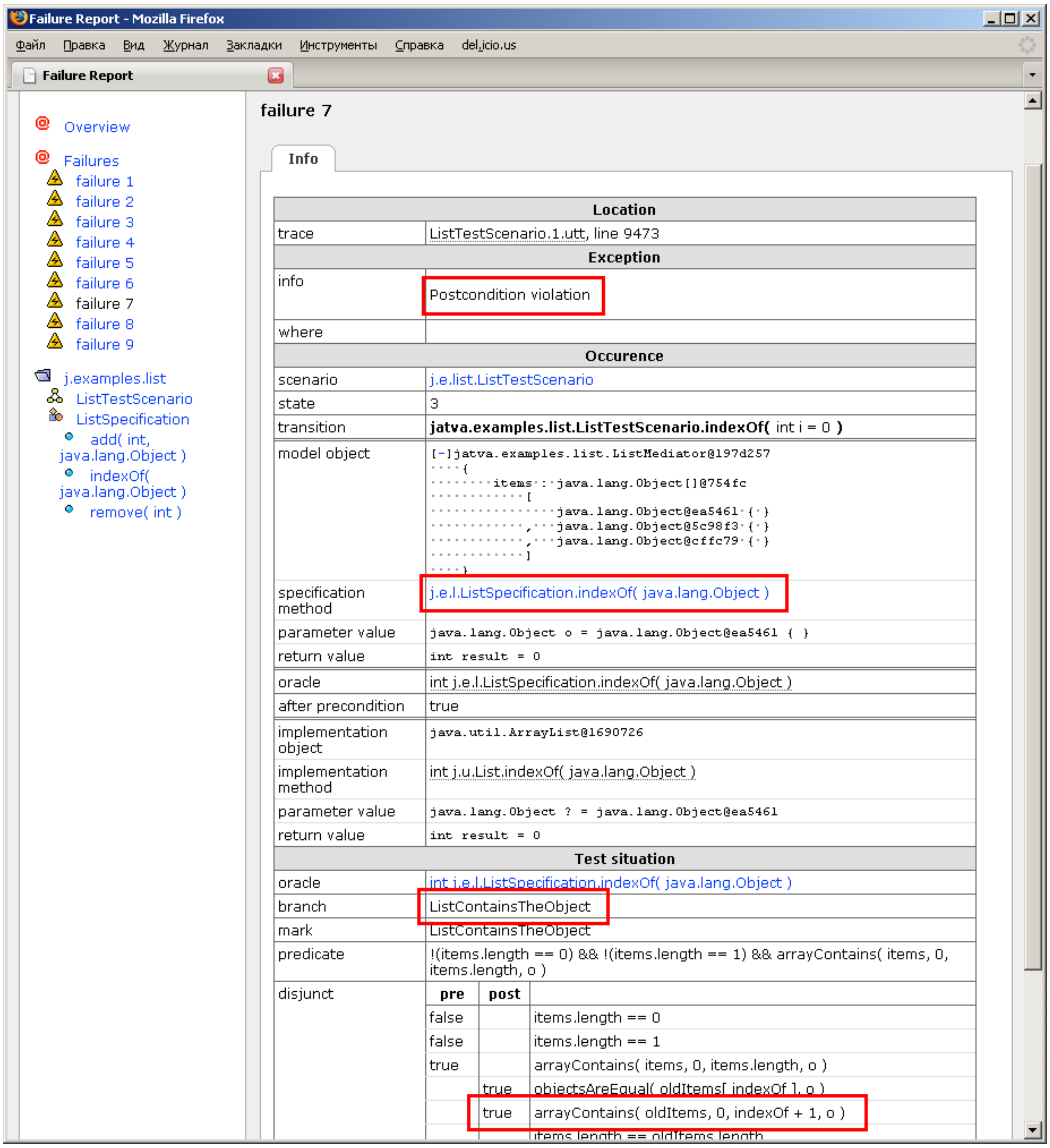

Package/Namespace Summary Report - Mozilla Firefox
Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us
file:///C:/Documents%20and%20Settings/kornevge
Google
Package/Namespace Summary ...
Overview
Failures (9)
j.examples.list
ListTestScenario
ListSpecification
Summary
Failures
Готово
Открыть блокнот (N)

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

jatva.examples.list

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)**

Look at the information about the first failure by **Failures/failure 7** menu item.


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	see below
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	see below

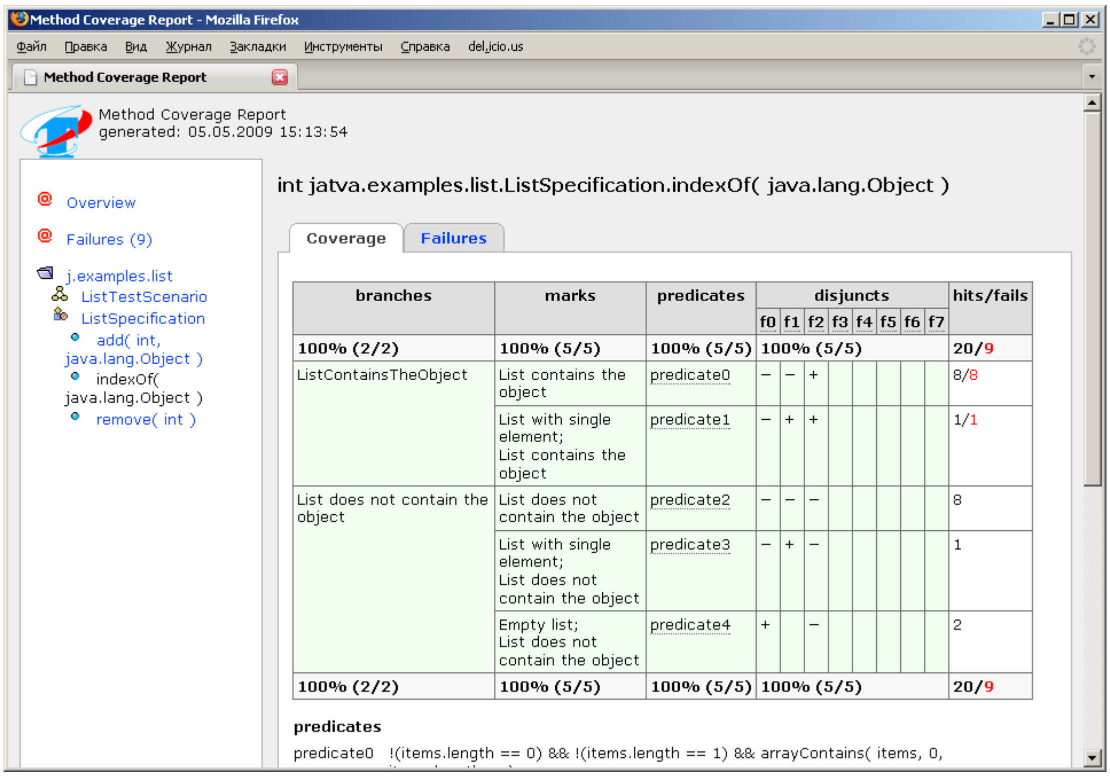
model object:

```
[-]jatva.examples.list.ListMediator@197d257
····{
········items·:·java.lang.Object[]@754fc
············[
················java.lang.Object@ea5461·{·}
············,···java.lang.Object@5c98f3·{·}
············,···java.lang.Object@cffc79·{·}
············]
····}
```

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

Report shows that the failure caused by postcondition violation from `indexOf()` invocation with argument contained in the list. According to specification `arrayContains(oldItems, 0, indexOf+1, o)` must have **`false`** value but it had **`true`** value when test executed. Review of other failures shows that they correspond to the same situation. Besides report shows all invocations of `indexOf()` was failed with argument contained in the list.


Method Coverage Report - Mozilla Firefox
Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us
Method Coverage Report

Method Coverage Report
generated: 05.05.2009 15:13:54

Overview
Failures (9)
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 | Failures

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)								20/9
ListContainsTheObject	List contains the object	predicate0	–	–	+						8/8
	List with single element; List contains the object	predicate1	–	+	+						1/1
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)								20/9

predicates

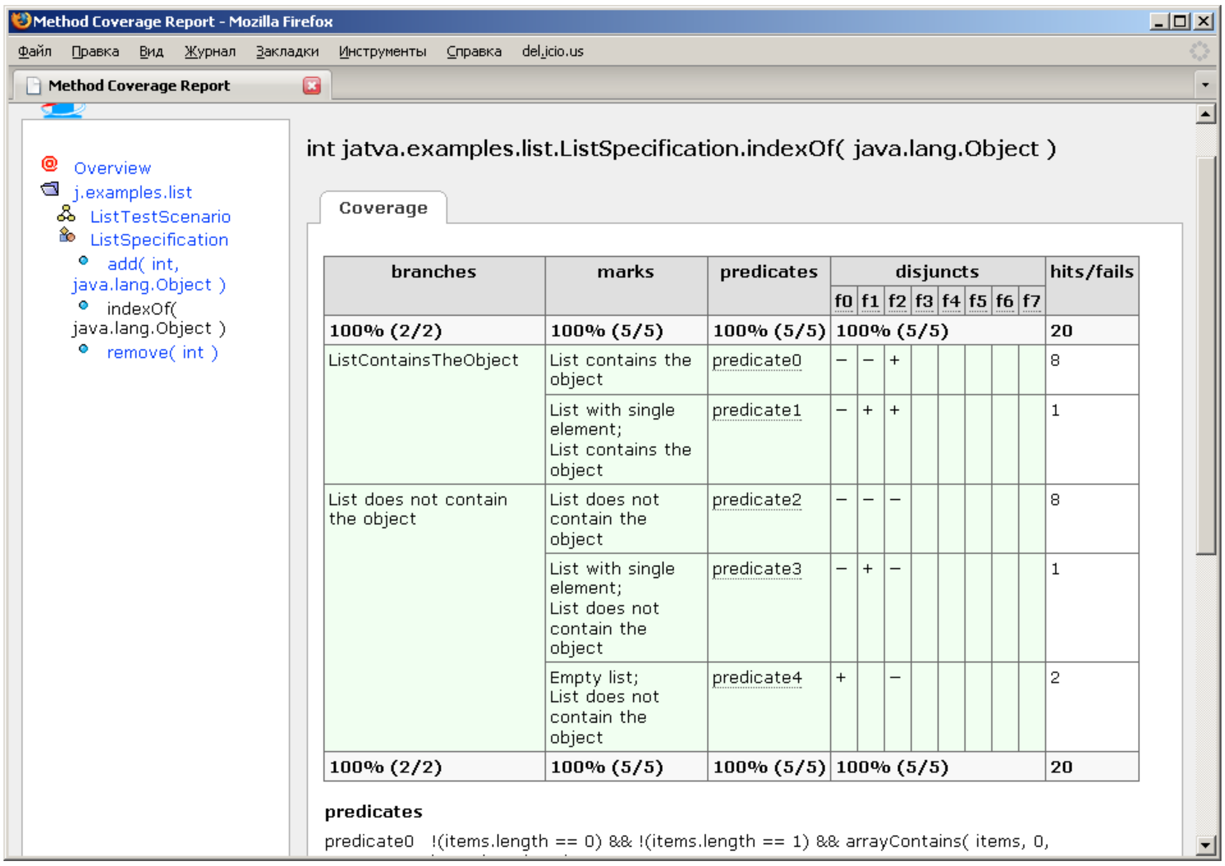
predicate0 !(items.length == 0) && !(items.length == 1) && arrayContains(items, 0,

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

Non-complex analysis of `arrayContains(oldItems, 0, indexOf+1, o)` shows it is always **`true`** in this situation, because it checks occurrence of `o` in the list on positions from 0 to the results of `indexOf()` invocation *inclusive*. So failure means error in specifications — part of `indexOf()` postcondition in case of containing argument in the list should be the following:

```
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);
}
```

After correction, recompilation and test execution new report hasn't contained any failures. Coverage report about `indexOf()` method can be the following:


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)

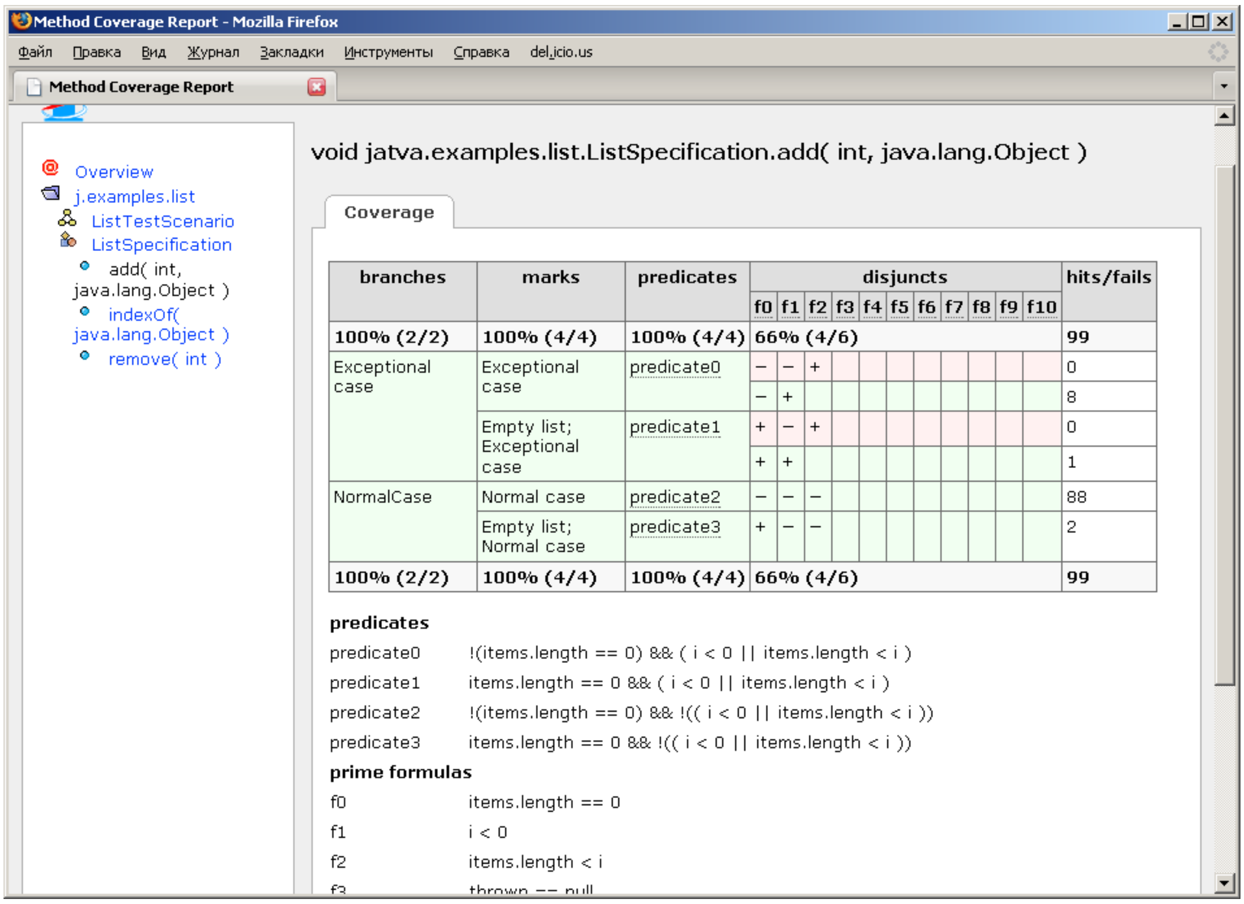
int jatva.examples.list.ListSpecification.indexOf(java.lang.Object)

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

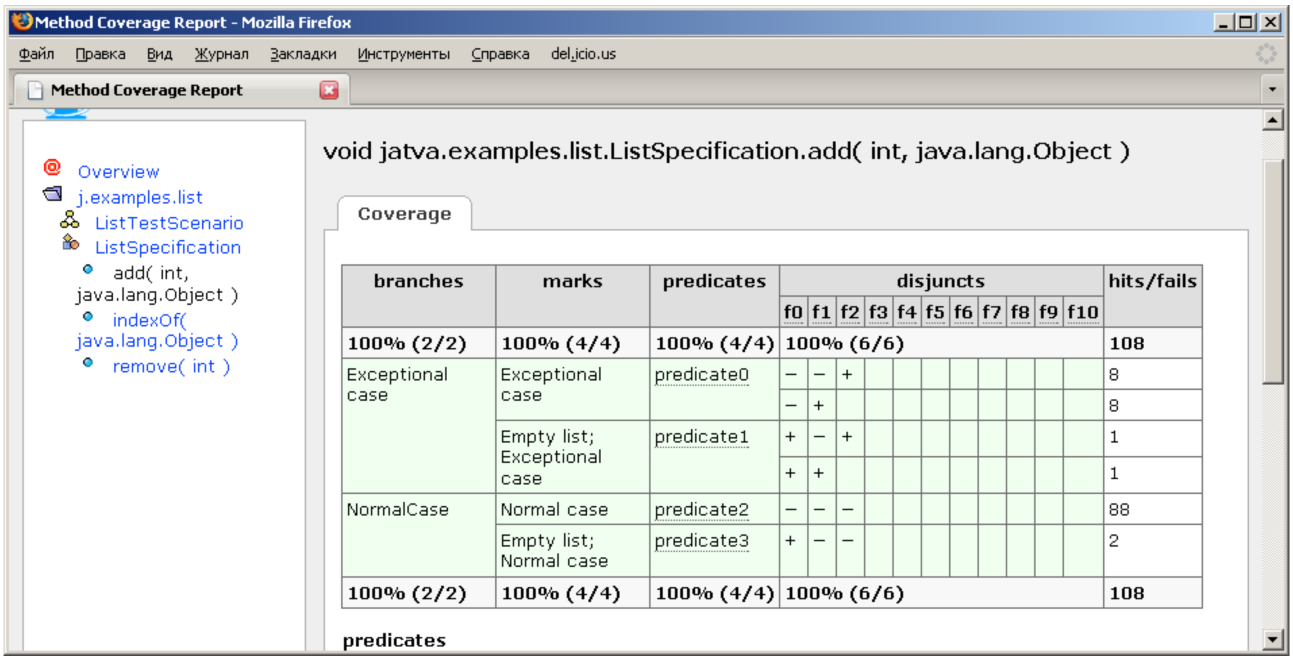
Coverage report for `add()` contains information about achieved 100% coverage by criteria of marked paths ("marks" column contains «100%»). "disjuncts" column means *coverage criteria by disjuncts*. Achieved coverage for this criteria is only 66%. Two rows with pink cells show non-100% coverage. Each pink row is divided into cells corresponding to elemental formula (without conjunction and disjunction). Cells for occurred elemental formula in marked paths has «+» (if it's true) or «-» (if it's false). Different sequences of "+" and "-" can correspond to the same marked path (because of lazy logic in Java). Report has all possible variants of elemental formula calculations with the exception of identically false formula. "Pink" formula didn't realized whilst testing. More clearly the following cases didn't be realized:

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

All cases can be merged into one – !f1 && f2, because value of f0 can be anyone. According to formula definition, it's necessary to realize `!(i < 0) && (items.length < i)` in scenario for `add()`. Other words it's necessary to invoke method `add()` with argument which value is more than length of the list:

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

Execution of the new test for `add()` reports about full coverage for all criteria:

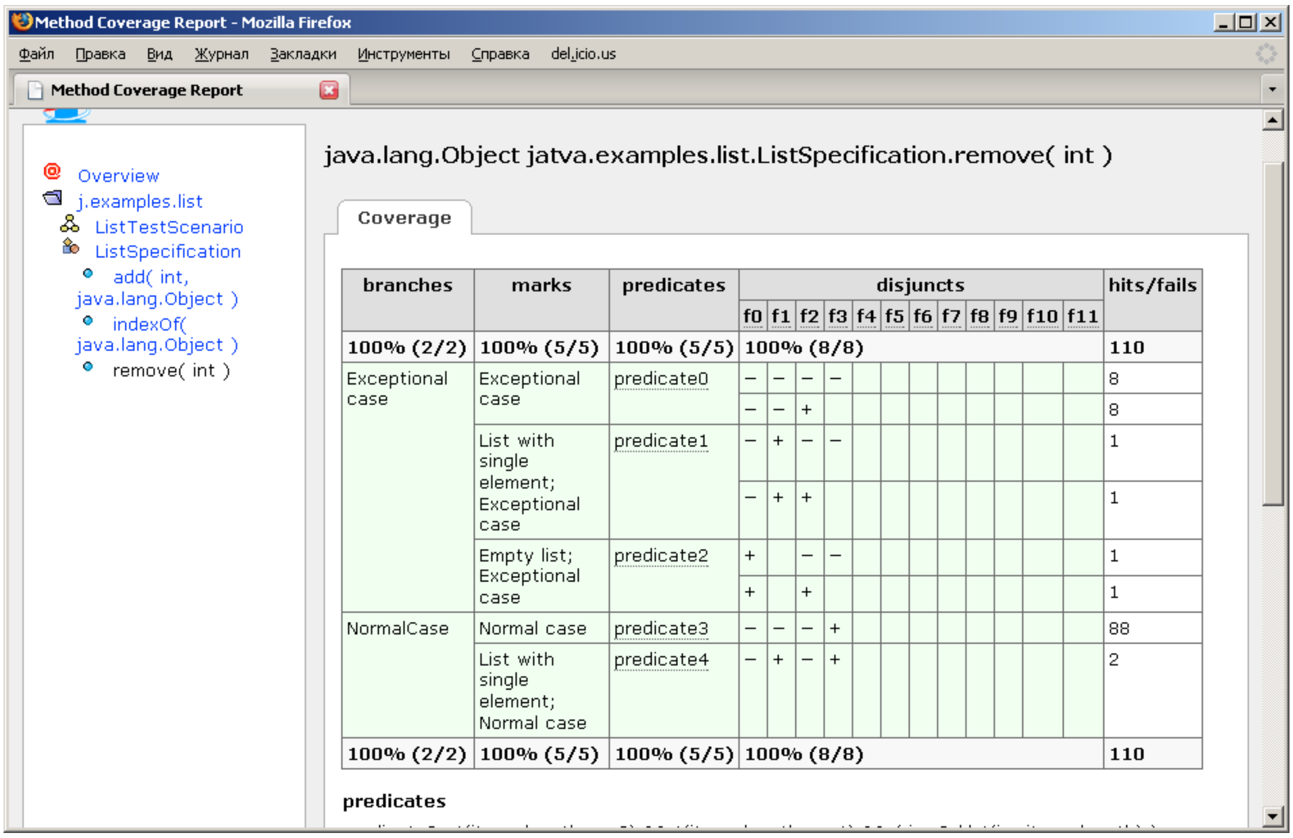

Method Coverage Report - Mozilla Firefox
Файл Правка Вид Журнал Закладки Инструменты Справка del.icio.us
Method Coverage Report
Overview
j.examples.list
ListTestScenario
ListSpecification
add(int, java.lang.Object)
indexOf(java.lang.Object)
remove(int)
Coverage

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

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

Scenario method `remove()` can be changed similarly. New scenario get full coverage by all criteria:



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)

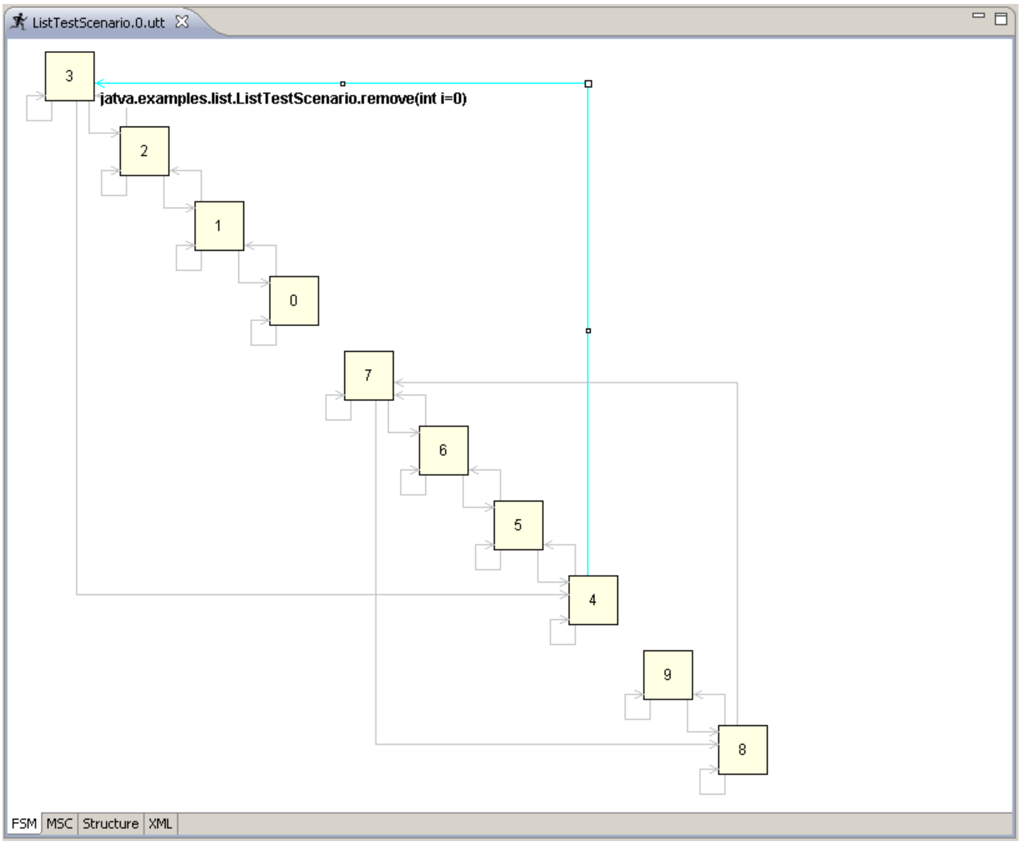
java.lang.Object jatva.examples.list.ListSpecification.remove(int)

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

Besides of failures report and coverage report, there is graphical report about test execution. It can be accessed by command **Open** from context menu for node corresponding to trace of test.


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