JavaTESK: Introduction

version 2.0

Introduction

In this document the process of tests development using JavaTESK tool is presented on the example of the class that implements methods of bank account processing.

Before reading, it is recommended to familiarize with *JavaTESK Installation Instruction*.

From this point on, class that tests are developed for will be called *target class* and testing methods of this class – *target methods*.

The example consists of the following parts:

Target Class Description

For any bank account, two methods are defined: add money or withdraw it from an account.

Data is constrained with the account balance. In addition, the maximum credit size is given that determines the amount that balance can descend below zero.

Class **Account** that implements the above methods is included in the examples that ship with JavaTESK, and it is located in the **jatva.examples.account** package.

Data of the class **Account**:

- **public int balance —** current balance of the account.
- **static public int maximumCredit** — the maximum size of the credit for the given class.

Interface of the class **Account** consists of the following methods:

- **Account()** — constructor, creates bank account with zero balance;
- **void deposit(int sum)** — performs adding positive amount **sum** to the account, increases a balance by the given amount;
- **int withdraw(int sum)** — performs withdrawing of the positive amount **sum** from the account, if the difference of the current balance and the amount **sum** is lower than allowed credit size then the method returns **sum**, otherwise it returns 0.

Project Creation in Development Environment

Project that contains target class **Account**, its requirements and tests is included in JavaTESK delivery in the project **account**. To familiarize with it import the project to **Eclipse** development environment. Select the command **`File/Import…`** in the

menu for this. Dialog **Import** will appear. Select **`General/Existing Projects into Workspace`**.

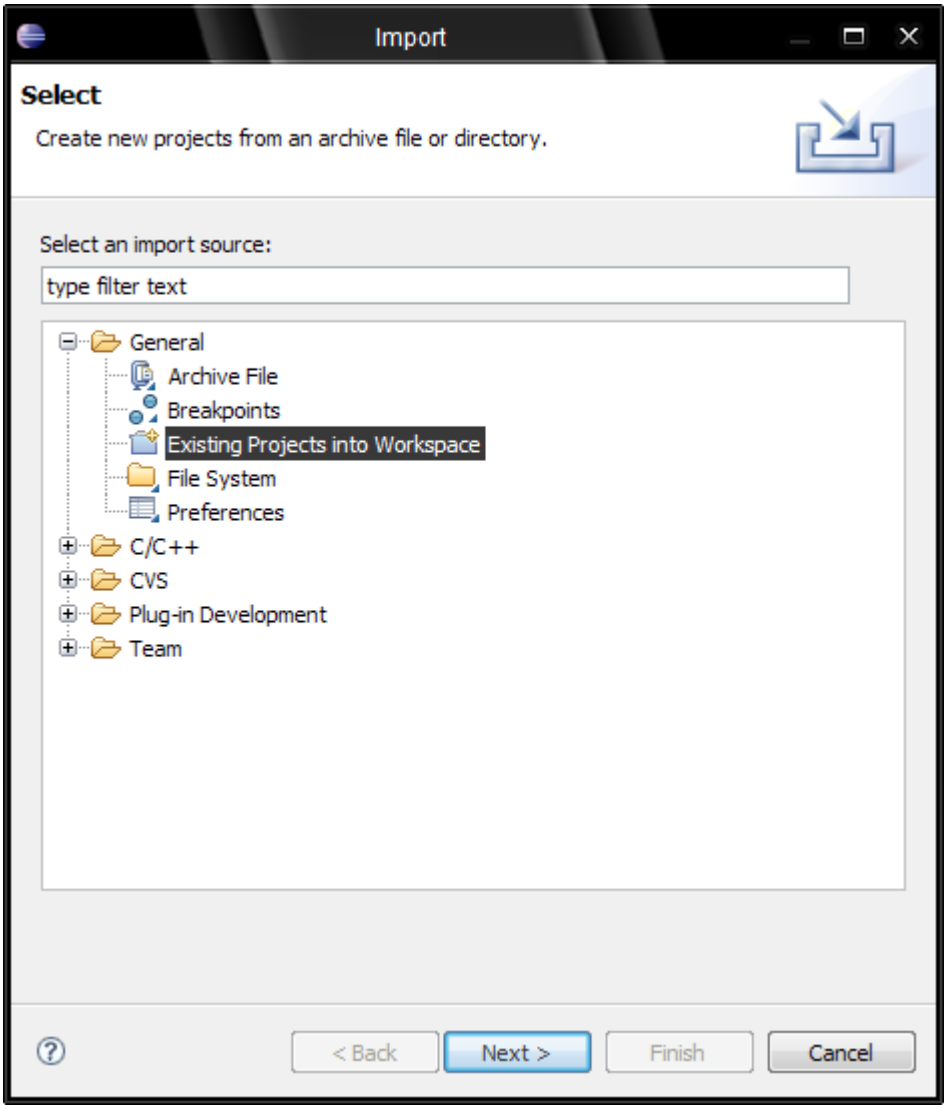


Figure 1. Select method of adding files to project.

Press **Next** button and dialog **Import** will change to allow selection of the folder with the examples. Press **Browse…** button that corresponds to the field **Select root directory**.

The folder with the examples is located at JavaTESK installation directory: after finding it, one should expand **examples** subfolder and select folder **account** as the target**.** Then finish the import process by pressing **OK** button.

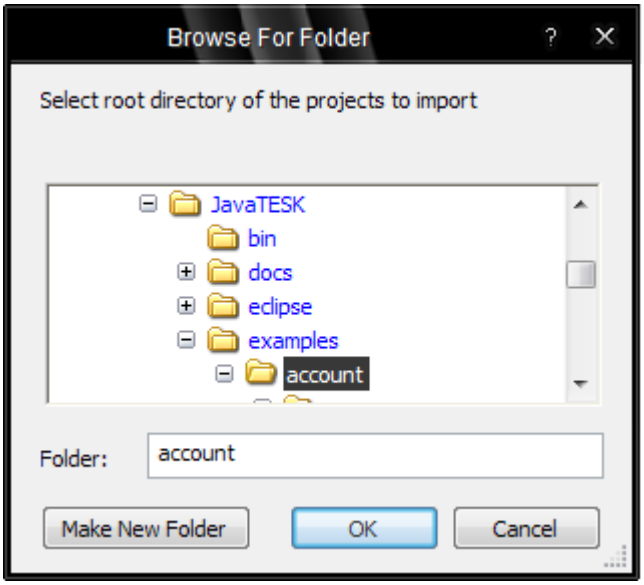


Figure 2. Select folder with example files for import.

After pressing **OK,** the environment will find project **account** at this folder. It is recommended to check **Copy projects into workspace** option to work with just copy of the project and to guarantee inviolability of the example files.
After pressing **Finish** button the project will be imported.

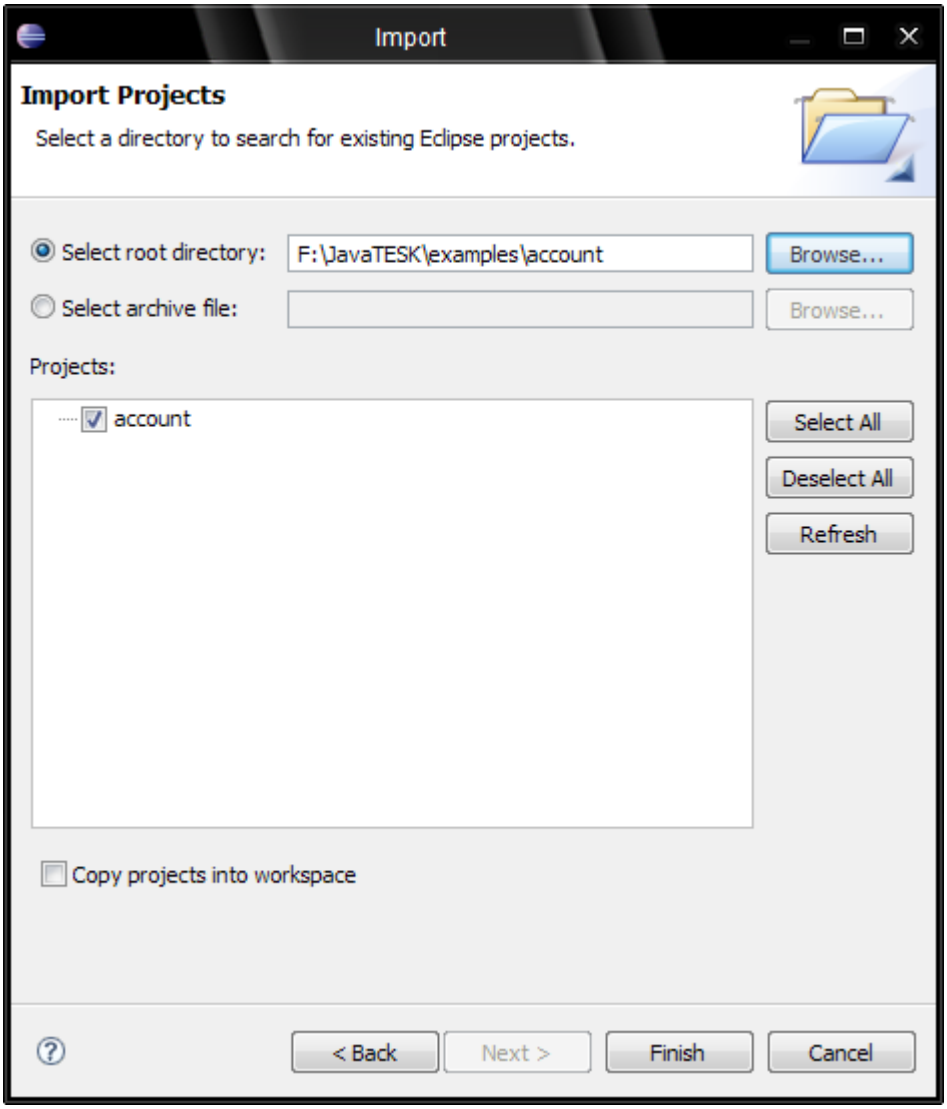


Figure 3. Select folder with the example files to import.

Project is ready for work. Make sure that directory structure is like this:

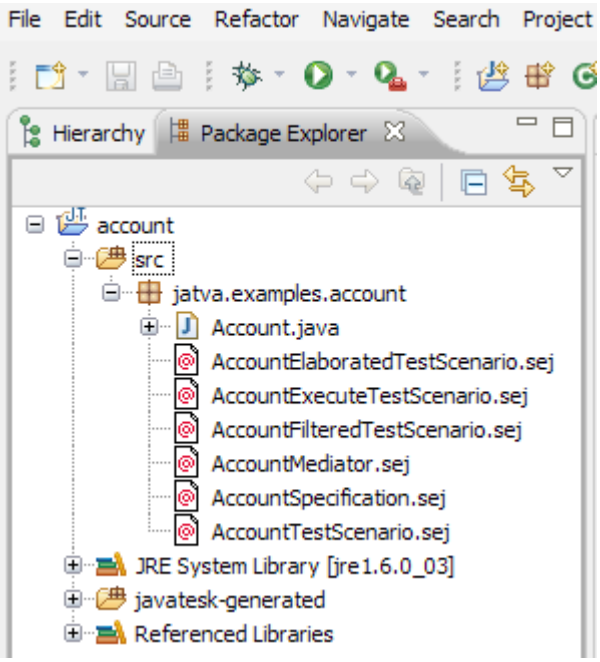


Figure 4. Directory structure of the ready project.

Target Class Specification

UniTESK testing technology that is supported by JavaTESK assumes that target class requirements are formulated in unambiguous form. This form of requirements representation is called *formal specification*. It may be used for automated *oracles* generation. Oracle is a component of test system that checks correspondence between target methods behavior and requirements for these methods. In JavaTESK formal specifications are developed using special language that is Java programming language extension. This language allows to describe *functional requirements* that determine the *functionality* of the target methods, that is what these methods should do.

Specifications using JavaTESK language are very similar to the usual Java code. Ones are described using special classes that are called *specification classes* and are located in the files with extension **.sej**.

In our project there is ready specification class **AccountSpecification.** Let's examine its code located in the file **AccountSpecification.sej**. To open this file, double click its icon in the **Package Explorer** window.

```
specification class AccountSpecification
{
  ...
}
```

Specification class differs with keyword **specification** from usual Java-class.

At the basis of the target class, *data model* is built and to store this model fields of specification class are added. In this case, class correspondence is simple that is why

class **AccountSpecification** has the same fields — **public int balance** to store the account balance and **static public int maximumCredit** for the maximum credit.

Constraints on the methods behavior are described as the methods of specification classes marked with the identifier **specification**. Such methods are called *specification methods*. Usually, specification method describes the behavior of single target method and has the same name.

Let's look at specification method **deposit**, that describes adding money to a bank account:

```
specification void deposit(int sum)
{
  pre {return (0 < sum ) && (balance <= Integer.MAX_VALUE - sum);}
  post
  {
    if(balance > 0)
      mark "Deposit on account with positive balance";
    else if(balance == 0)
      mark "Deposit on empty account";
    else
      mark "Deposit on account with negative balance";

    branch Single;

    return balance == pre balance + sum;
  }
}
```

Body of the specification method describes the behavior of the target method using the form of preconditions and postconditions.

Precondition is a block that is marked with the keyword **pre** and it returns the result of type **Boolean** depending on input and output parameters and the state of the object of the specification class. Precondition gives field of application for a specification method – if precondition returns **true** then one may expect correct method behavior; otherwise, the method result may be undefined.

In the above example, the precondition shows that the amount of the deposit **sum** should be positive and the result of addition of the current balance **balance** and the amount of deposit **sum** should not exceed maximum allowed value of type **int**.

Precondition of any specification method may be omitted and it is equivalent to presence of precondition that always returns **true**:

```
pre { return true; }
```

Postcondition is a block that is marked with keyword **post** and it returns result of type **boolean**. Postcondition analyses parameter values of the specification method and its result, state of the object of the specification class and shows if method behavior is as expected.

In the above example, the postcondition shows that the behavior of the method **deposit** is correct if the account balance after the method call (**balance**) equals to the sum of the account balance before method call (**pre balance**) and the amount of the deposit **sum**.

Operators **branch** and **mark** are used for definition of *test coverage* that will be described later.

Let’t look at the specification method **withdraw**, that describes withdraw from an account:

```
specification int withdraw(int sum)
{
  pre { return sum > 0; }
  post
  {
    if(balance > 0)
      mark "Withdrawal from account with positive balance";
    else if(balance == 0)
      mark "Withdrawal from empty account";

    else
      mark "Withdrawal from account with negative balance";

    if(balance < sum - maximumCredit)
    {
      branch TooLargeSum;

      return     balance == pre balance
              && withdraw == 0
             ;
    }
    else
    {
      branch Normal;

      return     balance == pre balance - sum
              && withdraw == sum
             ;
    }
  }
}
```

Precondition shows that the amount to withdraw **sum** should be positive.

Using the second condition **if,** postcondition processes two cases: when the withdraw of the given amount is impossible (the condition is true) and when the withdraw of the given amount is possible (the condition is false). Postcondition shows that in the first case the balance should be unchanged and the method should return **0**. In the second case, the balance should be decreased by the withdrawn amount **sum** and this sum also should be the method return value. To reference the result of the method **withdraw** the identifier with the same name is used.

In addition, JavaTESK language provides means to describe constraints for the possible values of the fields of specification classes. These constraints are called *data invariants* and are described as specification methods marked with the modifier **invariant**. The return value is of type **boolean**.

For the specification class **AccountSpecification** one data invariant **I** is defined:

```
invariant I()
{
  return balance >= -maximumCredit;
```

```
}
```

This invariant shows that the balance should not be lower than maximum allowed credit.

This is a complete source code listing of the specification class **AccountSpecification**.

```
package jatva.examples.account;

specification class AccountSpecification
{
  static public int maximumCredit;
  public int balance;

  public AccountSpecification() { }

  invariant I()
  {
    return balance >= -maximumCredit;
  }

  specification void deposit(int sum)
//    reads sum, Integer.MAX_VALUE
//    updates balance
  {
    pre { return (0 < sum ) && (balance <= Integer.MAX_VALUE - sum);
}
    post
    {
      if(balance > 0)
        mark "Deposit on account with positive balance";
      else if(balance == 0)
        mark "Deposit on empty account";
      else
        mark "Deposit on account with negative balance";

      branch Single;

      return balance == pre balance + sum;
    }
  }

  specification int withdraw(int sum)
//    reads sum, maximumCredit
//    updates balance
  {
    pre { return sum > 0; }
    post
    {
      if(balance > 0)
        mark "Withdrawal from account with positive balance";
      else if(balance == 0)
        mark "Withdrawal from empty account";
      else
        mark "Withdrawal from account with negative balance";

      if(balance < sum - maximumCredit)
      {
        branch TooLargeSum;

        return     balance == pre balance
                && withdraw == 0
               ;
      }
      else
```

```
      {
        branch Normal;

        return     balance == pre balance - sum
                && withdraw == sum
               ;
      }
    }
  }
}
```

To make sure the code of the specification class is correct, run the project build with the command **`Project/Build Project`** or run automatic build:

1. Call with the command **`Project/Clean`** dialog **Clean** and check **Clean all projects** checkbox or check **Clean projects selected below** checkbox, and mark off the project **AccountExample**.
2. Press **OK**.

If the code is correct, the window **Problems** (to show it select command **`Window/Show View/Problems`**) will be empty. This way the entire project is checked, correctness of mediators and scenarios may be checked the same manner.

Mediator Development

Now we have target class **Account** and specification class **AccountSpecification**. To give test the ability to check the correspondence between target and specification classes we need a link between these classes.

For this purpose special components of the test system are used that are called *mediators*. Mediators are described within special classes that are called *mediator classes*.

The file of the example **AccountMediator.sej** consists of the mediator class **AccountMediator**, that links specification class **AccountSpecification** and target class **Account**. So a mediator definition should implicitly include these classes indications.

- Specification class **AccountSpecification**, that the mediator is created for, is denoted in the declaration of the mediator class:

  ```
  mediator class AccountMediator implements AccountSpecification
  ```

  Keyword **implements** in Java means that declared class implements some interface. Similarly, mediator class definition should include mediator method for each method of specification class.
  In addition, mediator class inherits fields of the specification class to store test model data.

- The target class **Account** is used in the declaration of the special field of the mediator class:

  ```
  implementation Account targetObject = null;
  ```

Keyword **implementation** means that the purpose of this field is to store test object of the target class. Test actions are applied to this object.

So the specification and target class are identified. To test results be trustworthy, test data model (that consists of inherited from data fields of specification class) and data of the object under test (that contains target class fields) should strictly conform to one another. For this purpose, there is a block of synchronization **update** that synchronizes mediator fields with current object state of the target class. This synchronization should guarantee that target class behavior and its specification model behavior are both caused by the same input data. Let's consider block **updates** of the mediator class **AccountMediator**:

```
update
  {
    maximumCredit = Account.maximumCredit;
    if( targetObject != null )
    {
      balance = targetObject.balance;
    }
  }
```

Synchronization is rather simple here because the fields of the target class are stack variables – target class fields are simply assigned to mediator fields. In more complex cases, for example while synchronizing fields that contain objects of different classes and data structures, implementation of block **update** may become much more complex.

Synchronization is needed to check target class methods results. Mediator also performs these methods calls using mediator methods. Let's consider mediator method **deposit**:

```
mediator void deposit( int sum )
{
  implementation
  {
    targetObject.deposit( sum );
  }
}
```

The definition of this method is anticipated with keyword **mediator**. Keyword **implementation** in this case defines the call block of the target method.

The method **deposit** returns nothing, but it takes one parameter. If needed (if parameters have complex internal structure that is different from specification and target class), mediator method may consist some code that processes supplied parameters to the form suitable for the called method.

Value returned by the target method may be transformed to the form suitable for specification method. The transformation code should be inside this block because the call occurs in the block **implementation**. The result of transformation is returned back to specification method by operator **return** in the block **implementation**.

The code of the mediator method **withdraw**:

```
mediator int withdraw( int sum )
{
```

```
  implementation
  {
    return targetObject.withdraw( sum );
  }
}
```

Below is the listing of full source code for the mediator class **AccountMediator**.

```
package jatva.examples.account;

mediator class AccountMediator implements AccountSpecification
{
  mediator void deposit( int sum )
  {
    implementation
    {
      targetObject.deposit( sum );
    }
  }

  mediator int withdraw( int sum )
  {
    implementation
    {
      return targetObject.withdraw( sum );
    }
  }

  implementation Account targetObject = null;

  update
  {
    maximumCredit = Account.maximumCredit;
    if( targetObject != null )
    {
      balance = targetObject.balance;
    }
  }
}
```

Test Scenario Development

Test scenarios are developed to reach some testing goal. Usually, this goal is described in terms of test coverage, for example, one should reach 70% coverage of lines of code of a target class. In JavaTESK coverage criteria is described in terms of specification coverage.

Let’s assume that our goal is to reach 100% coverage of functional brunches of the specification class **AccountSpecification**. It means that we should run a test suite, that covers all functional brunches, defined in postconditions of the specification methods of the given class using operators **branch**.
There are following branches:

- **Single**
- **TooLargeSum**
- **Normal**

Test scenarios are described using special classes that are called *scenario classes*. In the example project, test scenario is located in the file **AccountTestScenario.sej**.

Definition of a scenario class should contain keyword **scenario**:

```
scenario class AccountTestScenario
```

Scenario class contains object of the specification class, methods of which will be called in the testing process:

```
protected AccountSpecification objectUnderTest;
```

In the constructor of the class **AccountTestScenario** there are indications for the target class and mediator: in our example for this purpose the method **configureMediators** is developed, that returns initialization value for the field **objectUnderTest**:

```
public AccountTestScenario()
{
  objectUnderTest = configureMediators();
  setTestEngine( new DFSMExplorer() );
}

public static AccountSpecification configureMediators()
{
  AccountSpecification result = mediator AccountMediator
                               ( targetObject = new Account() );
  result.attachOracle();
  return result;
}
```

Keyword **mediator** in this case is used for mediator class object creation. It differs from keyword **new** with the check: if mediator of the type **AccountMediator**, which is initialized with object of class **Account,** already exists then new mediator is not created and old one is used.

Method calls of the object **objectUnderTest** are called *test actions* and are performed in *test methods*.

Let's look at the method **deposit**:

```
scenario deposit()
{
  if(objectUnderTest.balance < maxBalance)
  {
    objectUnderTest.deposit( 1 );
  }
  return true;
}
```

To limit the number of test actions, maximum value (field **maxBalance**) is introduced. It is assumed that method **deposit** increases balance by one on each call, so after a finite number of calls maximum should be reached.

In the scenario method **withdraw** one may see incremental parameters processing:

```
scenario withdraw()
{
  iterate( int i = 1; i < maxCredit+3; i++; )
  {
    objectUnderTest.withdraw( i );
```

```
  }
  return true;
}
```

Usual cycles work until some internal condition is false. Operator **iterate** is similar to the usual cycle with the only difference that only one iteration of cycle is performed on each scenario method call.

Therefore, operator iterate gives the ability to limit each scenario method call with single test action and at the same time it defines parameters for multiple actions in the code of the same scenario method.

After objects of testing and scenario methods are defined, one should include in the method **main** instructions that run the test:

```
AccountTestScenario myScenario = new AccountTestScenario();     my-
Scenario.run();
```

Below is the full source code of the scenario class **AccountTestScenario**.

```
package jatva.examples.account;

import jatva.engines.DFSMExplorer;

scenario class AccountTestScenario
{
  public static AccountSpecification configureMediators()
  {
    AccountSpecification result = mediator AccountMediator( targetOb-
ject = new Account() );
    result.attachOracle();
    return result;
  }

  int maxCredit = 3;
  int maxBalance = 10;

  public static void main( String[] args )
  {
      jatva.tracer.Tracer.getPrototype().setXmlFormat();
    AccountTestScenario myScenario = new AccountTestScenario();

    if(args.length > 0)
    {
      int n = Integer.parseInt(args[0]);
      if(n < 1) n = 1;
      myScenario.maxBalance = n;

      if(args.length > 1)
      {
        n = Integer.parseInt(args[1]);
        if(n < 0) n = 0;
        myScenario.maxCredit = n;
        Account.maximumCredit = n;
      }
    }

    myScenario.run();
  }

  protected AccountSpecification objectUnderTest;

  public AccountTestScenario()
  {
    objectUnderTest = configureMediators();
    setTestEngine( new DFSMExplorer() );
  }

  state
  {
    return new Integer( objectUnderTest.balance );
  }

  scenario deposit()
  {
    if(objectUnderTest.balance < maxBalance)
    {
      objectUnderTest.deposit( 1 );
    }
```

```
    return true;
  }

  scenario withdraw()
  {
    iterate( int i = 1; i < maxCredit+3; i++; )
    {
      objectUnderTest.withdraw( i );
    }
    return true;
  }
}
```

Tests Run and Results Analyzing

To run test scenario one should select correspondent file in the window **Package Explorer** and press keys combination **`Ctrl+F11`**, right click the window and select the element of the context menu **`Run As/JavaTESK Test`**.

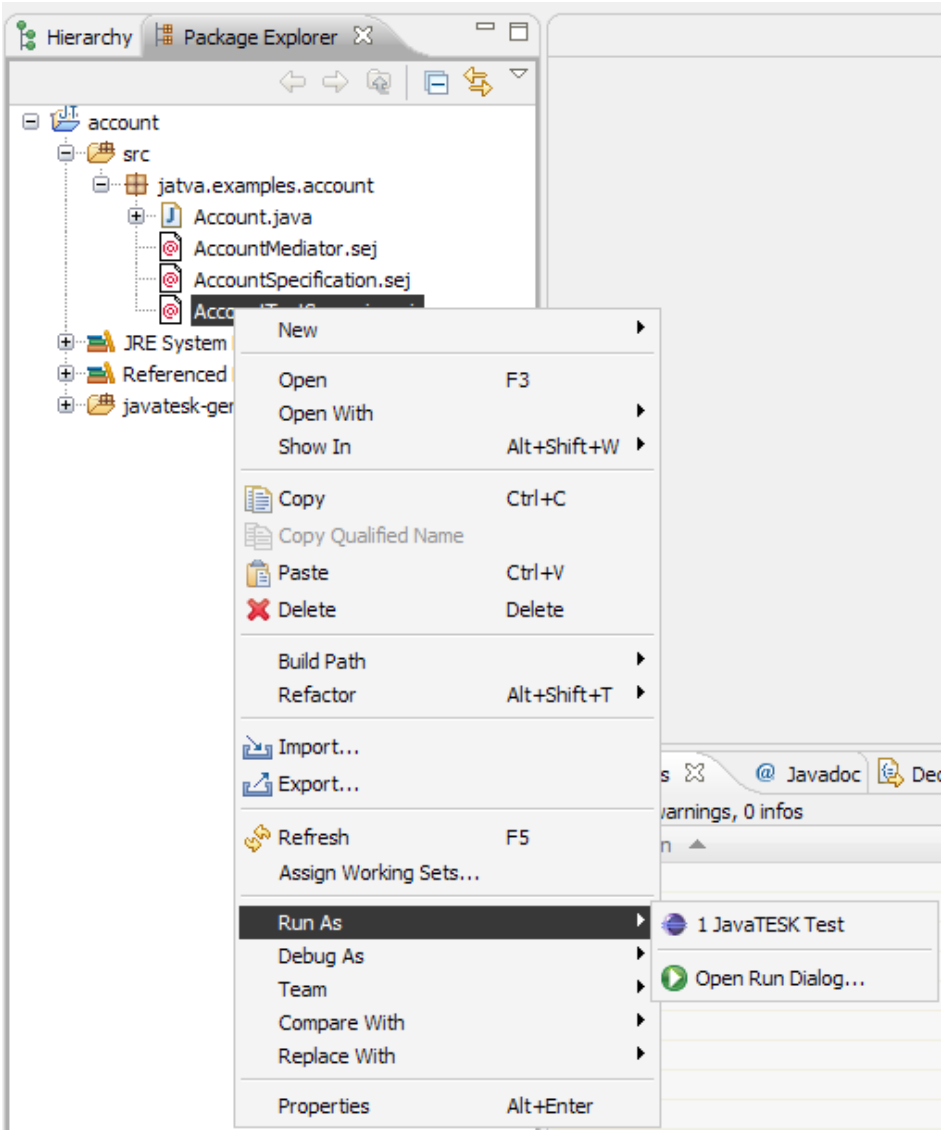


Figure 5. Run of the scenario class AccountTestScenario.

In the case of successful test run, trace file is created with extension .**utt**. Its name consists of the name of scenario class and trace number separated with dot. The trace file appears in the project on the same hierarchy level as the finished scenario.

Make sure that the trace file is created and window tab of **Package Explorer** looks like at the figure.

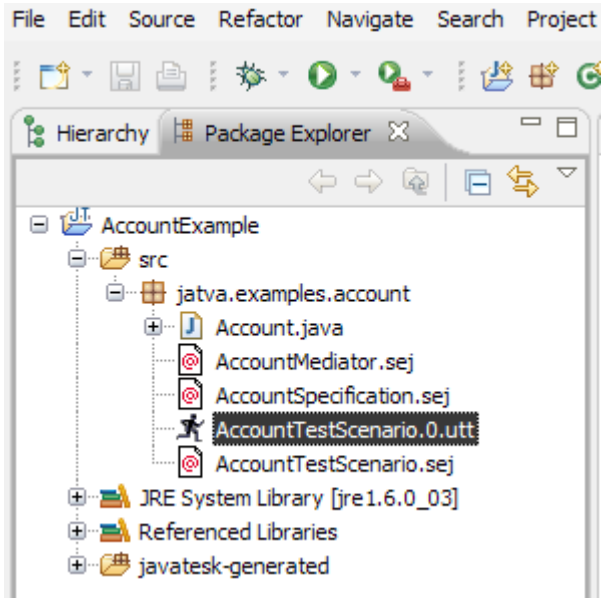


Figure 6. Trace file after run of the scenario AccountTestScenario.

Trace file contains input information for reports generation about testing performed.

In reports there is information about errors, found during testing and level of test coverage reached.

JavaTESK supports to types of reports:

- *HTML reports*
- *Trace representation*

HTML report

HTML report is a collection of HTML documents with information about testing performed.

To generate HTML one should right click in **Package Explorer** window on correspondent trace file and select command **`Generate Report`**.

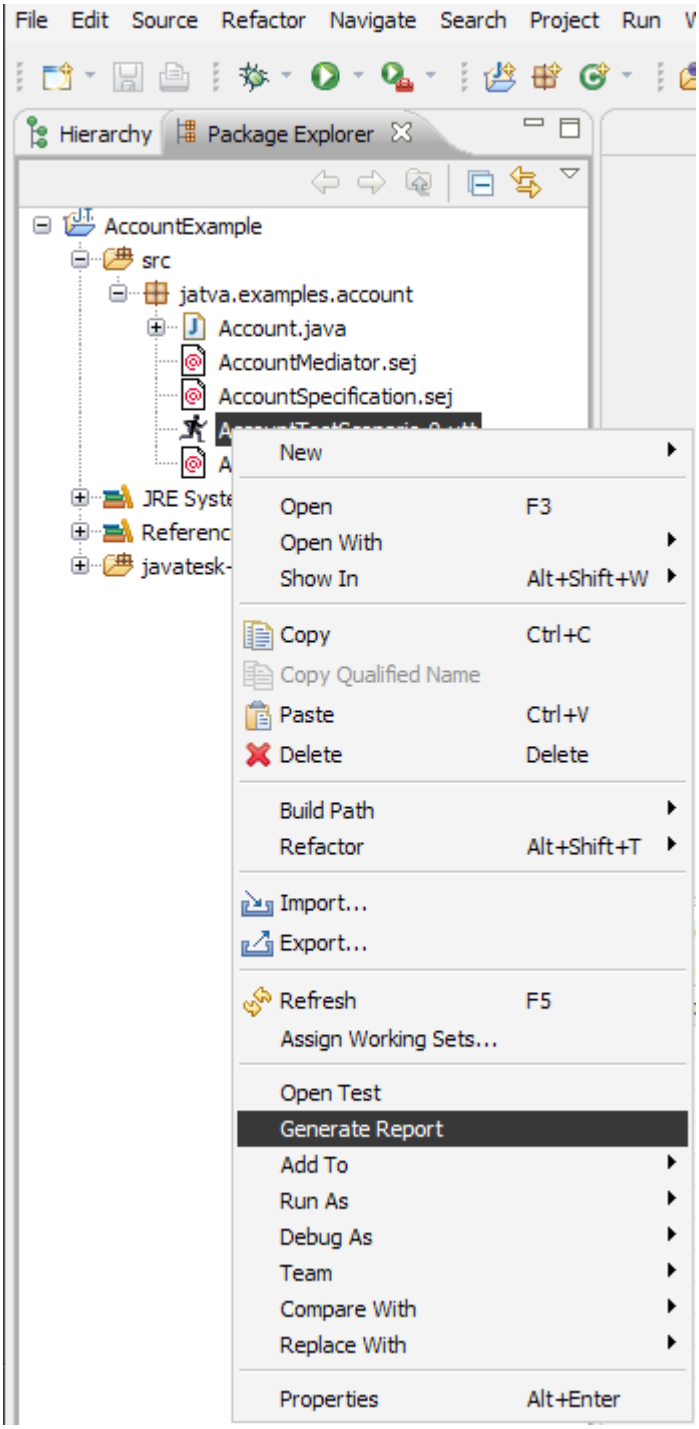


Figure 7. HTML report generation.

Dialog **Generate UniTESK Report** will be opened with options for the HTML report generation. Tab **Generate** contains parameters of the files generation and tab **Report** allows to chose the content of the report.
To view the reports in the external browser one should check **Use external browser to view report** checkbox.

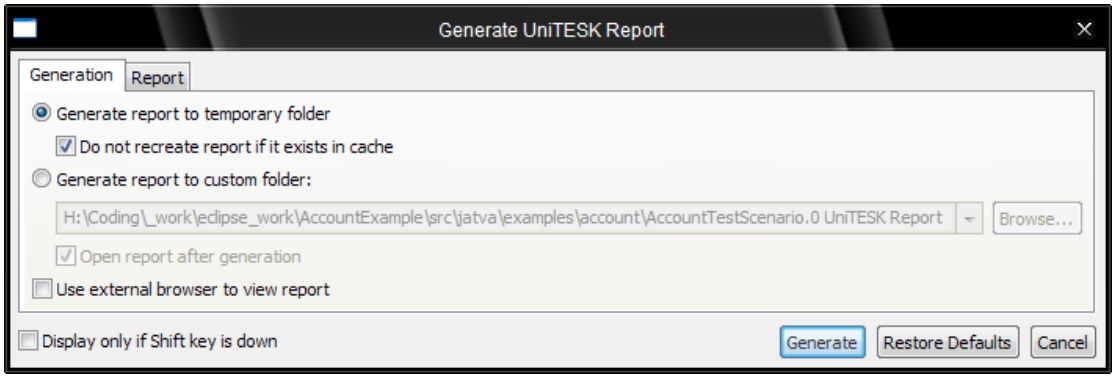


Figure 8. Options to generate HTML report.

The table on the page **Overview** provides information about coverage of test packages and namespaces that were used in the tests run. There are following columns:

- **`branches`** — level of functional brunches coverage
- **`marks`** — level of marked ways coverage
- **`predicates`** — level of predicate coverage
- **`disjuncts`** — level of disjunctive coverage
- **`states/transitions`** — number of states/number of transitions between states

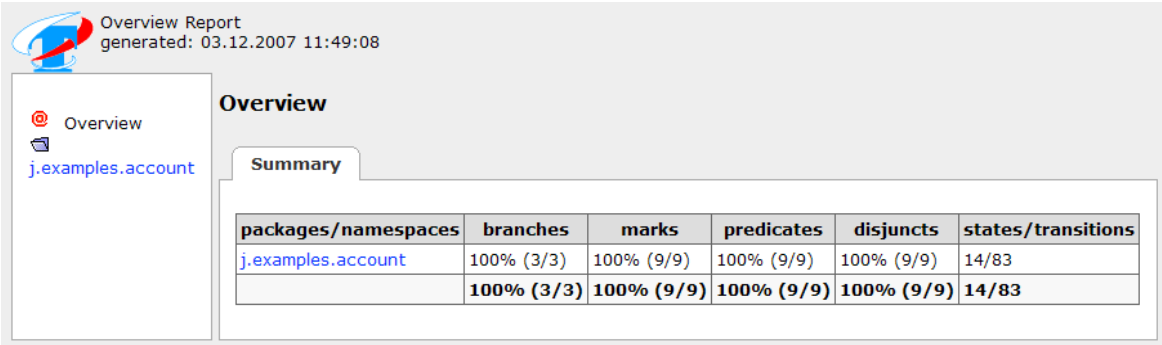


Figure 9. Page Overview HTML of the report.

To get information about testing of the class Account, select menu item **j.examples.account**. Information from the **Overview** table will be divided into the **scenarios** and **specifications** tables: the states number is a scenario property and coverage values are specification property.

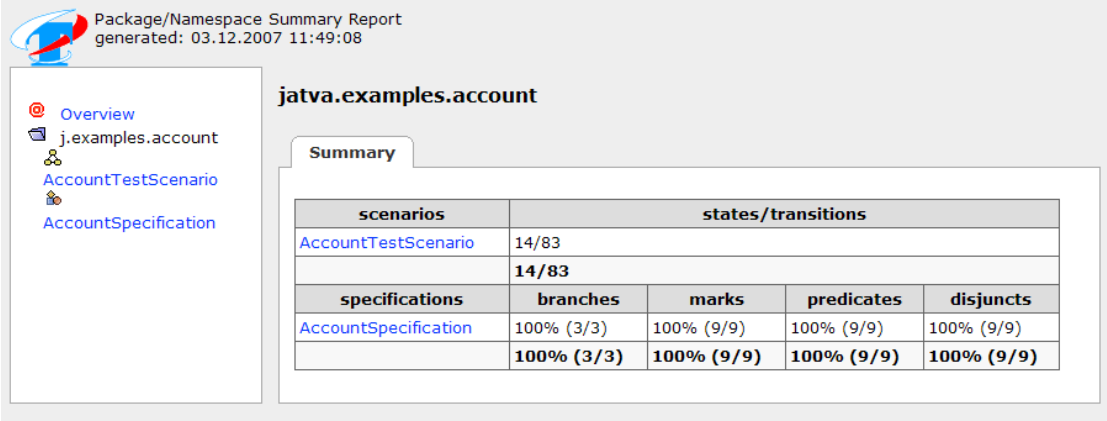


Figure 10. Page with test results overview.

Sub item of menu **AccountTestScenario** opens table **Scenario Transitions Report**, where one may find all transitions between states of the test model with the following characteristics:

- **`states`** — initial states (before transitions). Transitions are grouped by this parameter – there are double borders between groups.
- **`transitions`** — methods that cause transition and values of the iteration variables of the **iterate** operators that were used by call
- **`end states`** — finite states (after transitions)
- **`hits/fails`** — number of transitions performed/number of errors found in the transitions

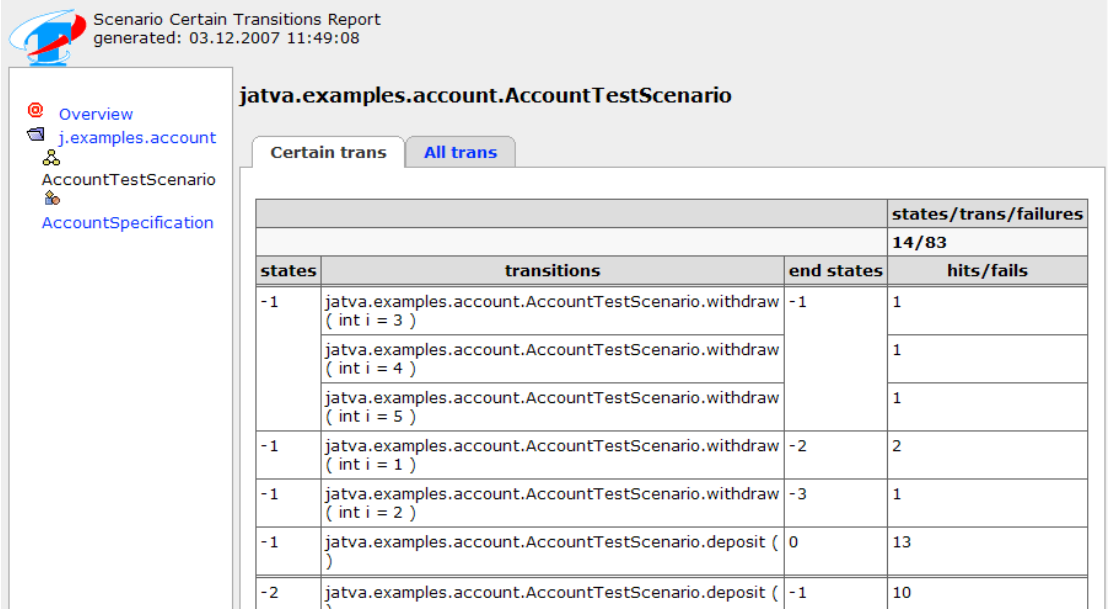
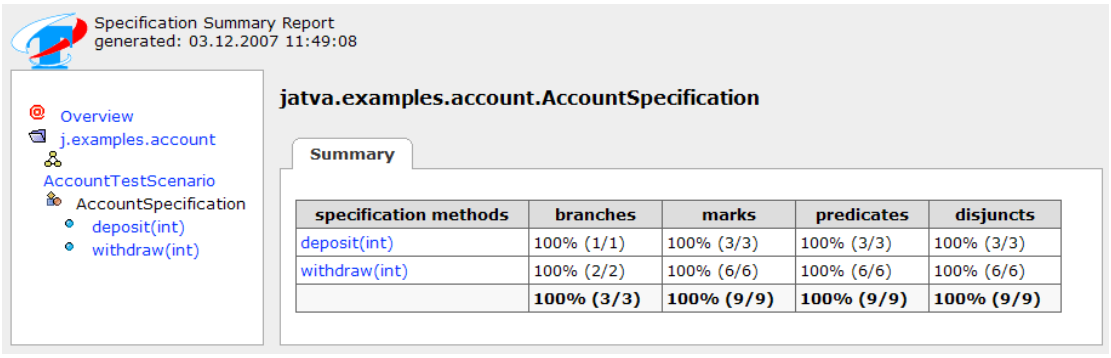


Figure 11. Page with scenario transitions data of the HTML report.

Sub item of the menu **AccountSpecification** opens table **Specification Summary Report**, where coverage parameters are described in detail for each specification method:



Specification Summary Report
generated: 03.12.2007 11:49:08

Overview
j.examples.account
AccountTestScenario
AccountSpecification
deposit(int)
withdraw(int)

jatva.examples.account.AccountSpecification

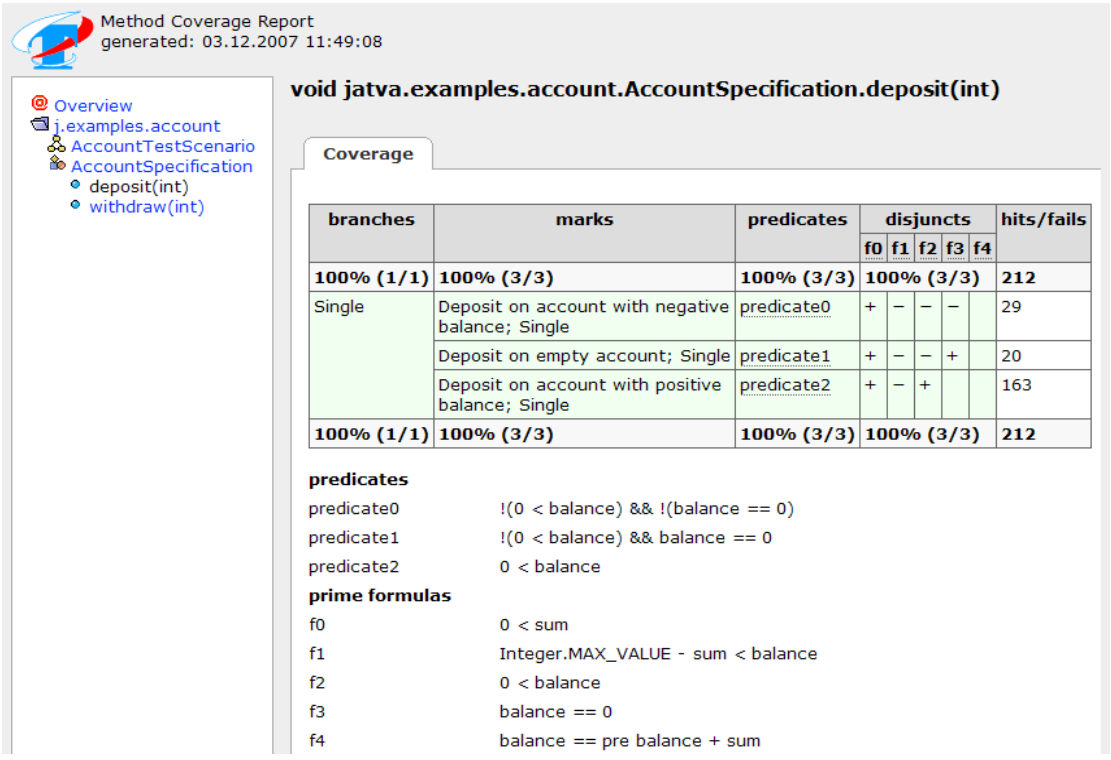
Summary

specification methods	branches	marks	predicates	disjuncts
deposit(int)	100% (1/1)	100% (3/3)	100% (3/3)	100% (3/3)
withdraw(int)	100% (2/2)	100% (6/6)	100% (6/6)	100% (6/6)
	100% (3/3)	**100% (9/9)**	**100% (9/9)**	**100% (9/9)**

Figure 12. Summary page of specification methods coverage.

Click on either method to open the extended coverage table:

- **`branches`** — covered branches (in the code marked with keyword **branch**)
- **`marks`** — marks of variants branch pass (marked with keyword **mark**)
- **`predicates`** — conditions, that correspond to label **mark** of the same table line
- **`disjuncts`** — logic values of items preconditions and postconditions of variant branch pass
- **`hits/fails`** — number of passes for the given branch and number of errors found in these transitions



Method Coverage Report
generated: 03.12.2007 11:49:08

Overview
j.examples.account
AccountTestScenario
AccountSpecification
deposit(int)
withdraw(int)

void jatva.examples.account.AccountSpecification.deposit(int)

Coverage

branches	marks	predicates	disjuncts					hits/fails
			f0	f1	f2	f3	f4	
100% (1/1)	**100% (3/3)**	**100% (3/3)**	**100% (3/3)**					**212**
Single	Deposit on account with negative balance; Single	predicate0	+	−	−	−		29
	Deposit on empty account; Single	predicate1	+	−	−	+		20
	Deposit on account with positive balance; Single	predicate2	+	−	+			163
100% (1/1)	**100% (3/3)**	**100% (3/3)**	**100% (3/3)**					**212**

predicates

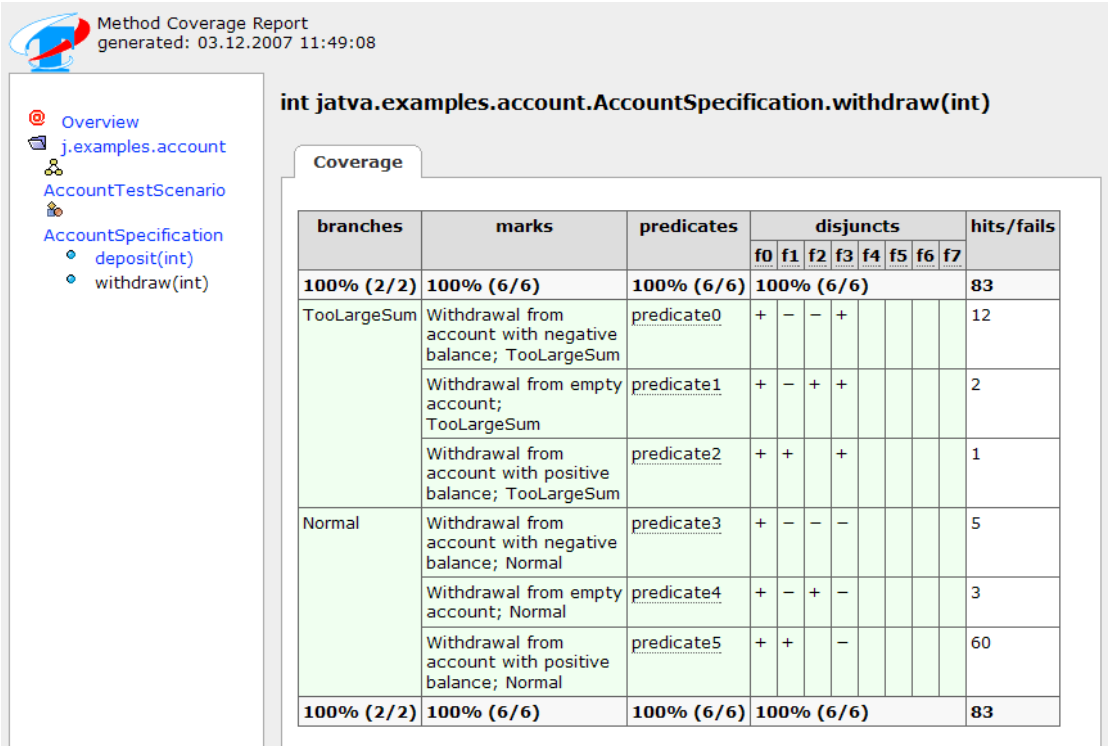
predicate0	!(0 < balance) && !(balance == 0)
predicate1	!(0 < balance) && balance == 0
predicate2	0 < balance

prime formulas

f0	0 < sum
f1	Integer.MAX_VALUE - sum < balance
f2	0 < balance
f3	balance == 0
f4	balance == pre balance + sum

Figure 13. Table with test results for the specification method deposit.

Similar table is for the method **withdraw**; variants for different branches **branch** are grouped accordingly.



Method Coverage Report
generated: 03.12.2007 11:49:08

Overview
j.examples.account
AccountTestScenario
AccountSpecification
deposit(int)
withdraw(int)

int jatva.examples.account.AccountSpecification.withdraw(int)

Coverage

branches	marks	predicates	disjuncts								hits/fails
			f0	f1	f2	f3	f4	f5	f6	f7	
100% (2/2)	**100% (6/6)**	**100% (6/6)**	**100% (6/6)**								**83**
TooLargeSum	Withdrawal from account with negative balance; TooLargeSum	predicate0	+	–	–	+					12
	Withdrawal from empty account; TooLargeSum	predicate1	+	–	+	+					2
	Withdrawal from account with positive balance; TooLargeSum	predicate2	+	+		+					1
Normal	Withdrawal from account with negative balance; Normal	predicate3	+	–	–	–					5
	Withdrawal from empty account; Normal	predicate4	+	–	+	–					3
	Withdrawal from account with positive balance; Normal	predicate5	+	+		–					60
100% (2/2)	**100% (6/6)**	**100% (6/6)**	**100% (6/6)**								**83**

Figure 14. Table with test results of specification method withdraw.

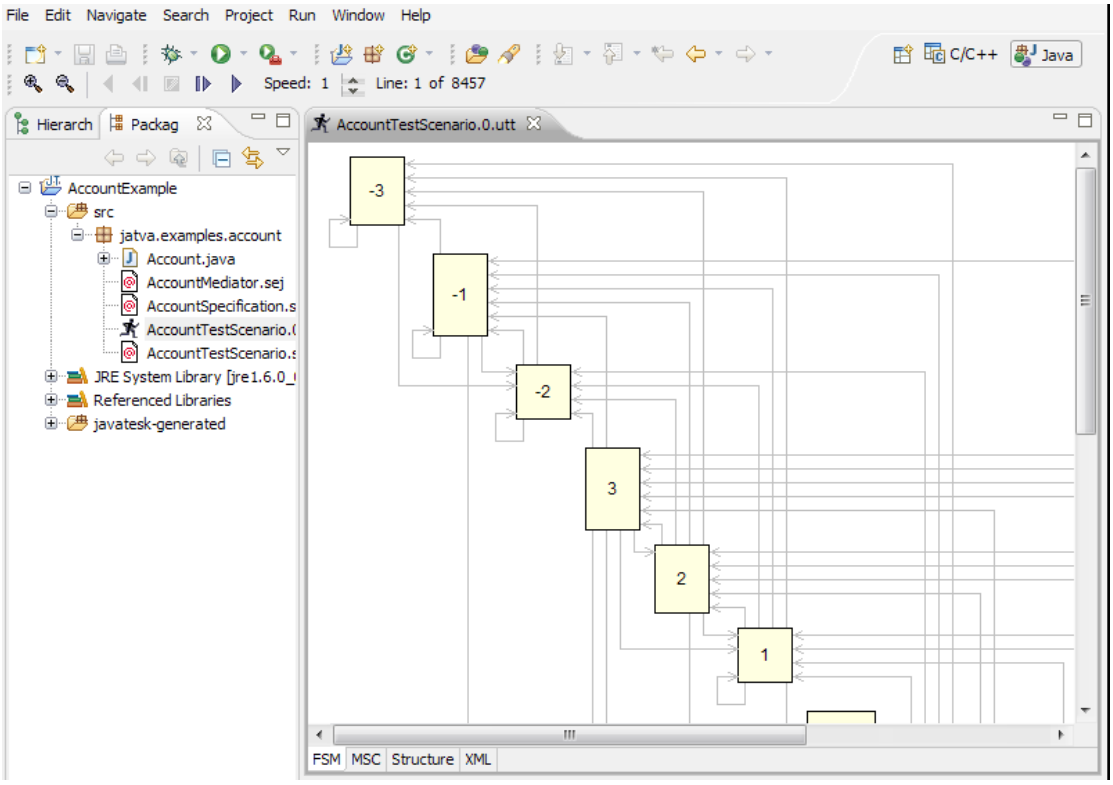
Note that report data here shows full target functionality coverage: all planned behavior variants are correctly tested.

Trace Representations

JavaTESK supports four trace representation views:

- **XML** (*XML representation*) – representation of the trace in XML format, text file generated by test environment
- **Structure** (*structure representation*) – representation of the trace as a tree from structure trace elements
- **MSC** (*MSC representation*) – representation of the trace as MSC (*Message Sequence Charts*) diagram
- **FSM Model** (*automata representation*) – representation of one of the test scenarios as a state and transitions graph of finite automata

In IDE **Eclipse** it is possible to view all four representations: double click to open default representation (**FSM**) and switch between representations using tabs in the lower part of the window.



Fugure 15. FSM-representation of the trace.

In the given representation, the trace is depicted as a state graph of finite automata described in the scenario class **AccountTestScenario**. To find out details about transition click arrow that shows it. Line will be outlined in color and under the arrow full name of the scenario method will be shown and correspondent iteration variables values will be depicted also.

JavaTESK gives the ability to play trace representation back and forward. By default, **Play Forward** button starts automata playback from initial state; if one selects by click some state or arrow pointing to it then playback will start from this place. Current states and transitions are outlined in color.

Use counter **Speed** to set up play speed. Other buttons work according to the icons: allow to play in the reverse direction, stop playback until automata work finishes and see step-by-step run in both directions.

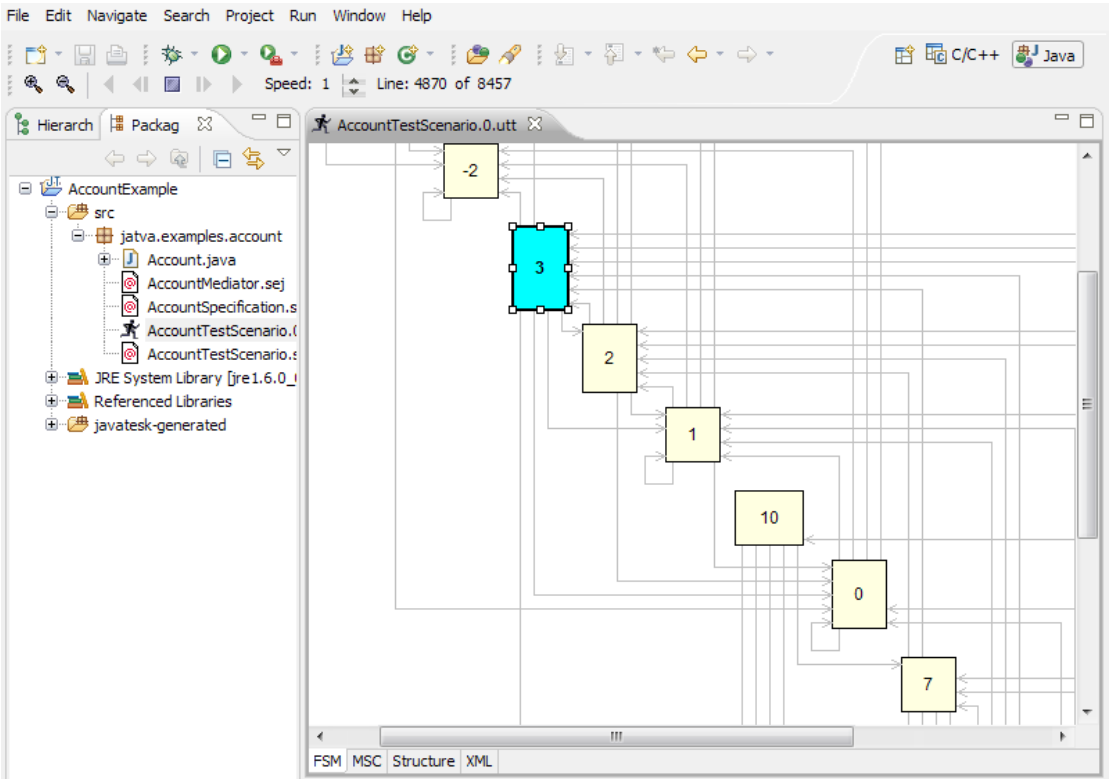


Figure 16. FSM-representation of the trace in the playback mode.

Activate the **XML** tab, to see XML representation of the trace. In this representation, the trace is shown in its original way, as it is stored in **.utt** file.

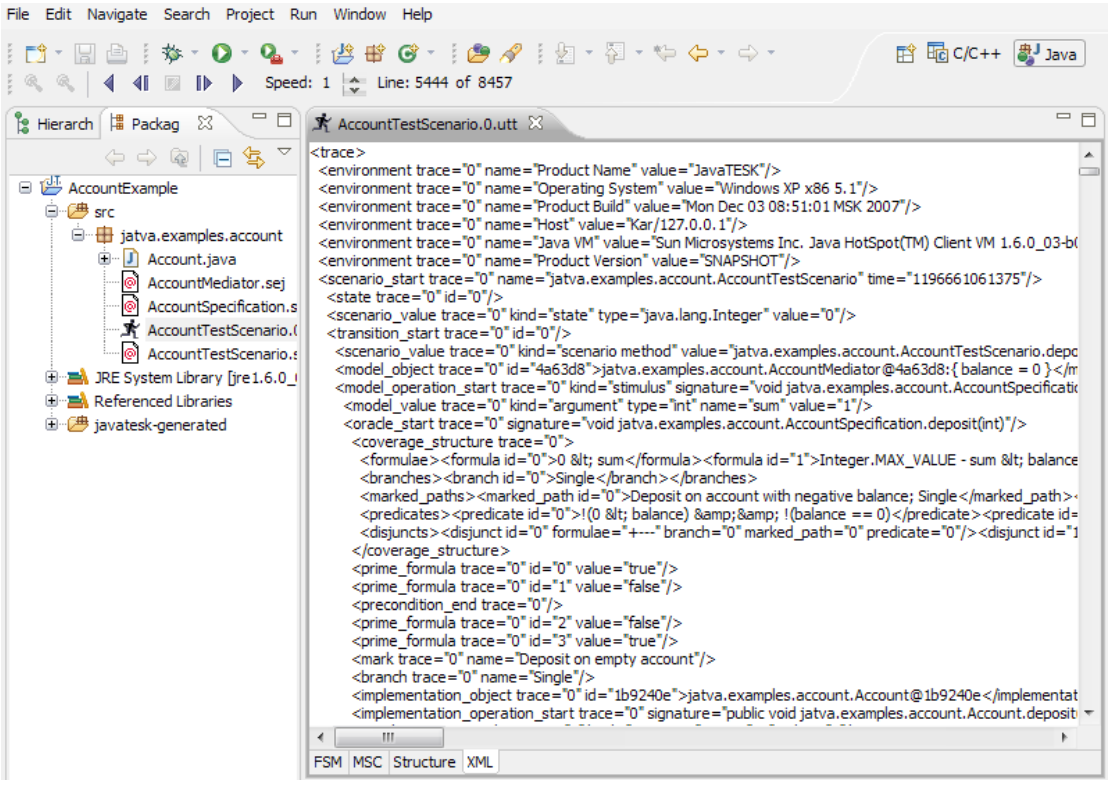


Figure 17. XML representation of the trace.

To see the structure representation of the trace open tab **`Structure`**. Trace is represented as a tree of structure elements there.

The root of the tree is node **`Trace`** that corresponds to the selected trace file. Its descendants are nodes **`Thread`** that describe threads created by the test. In our example we create one thread with the identifier **0**. Its descendants are nodes **`Scenario`**, that describe scenario classes run in the given thread. In our example this is a single scenario class **AccountTestScenario**. Its descendants are nodes **`State`**, that describe states of the finite automata listed in the order passed during testing. For each state, there is its identifier and transition **`Transition`**. Transition is characterized with the scenario method (**jatva.examples.account.AccountTestScenario.deposit**) and model of the target class **`Model`**, where the specification method (**jatva.examples.account.AccountSpecification.deposit(int)**) is denoted and the node **`Oracle`**, that contains checks performed by oracle.

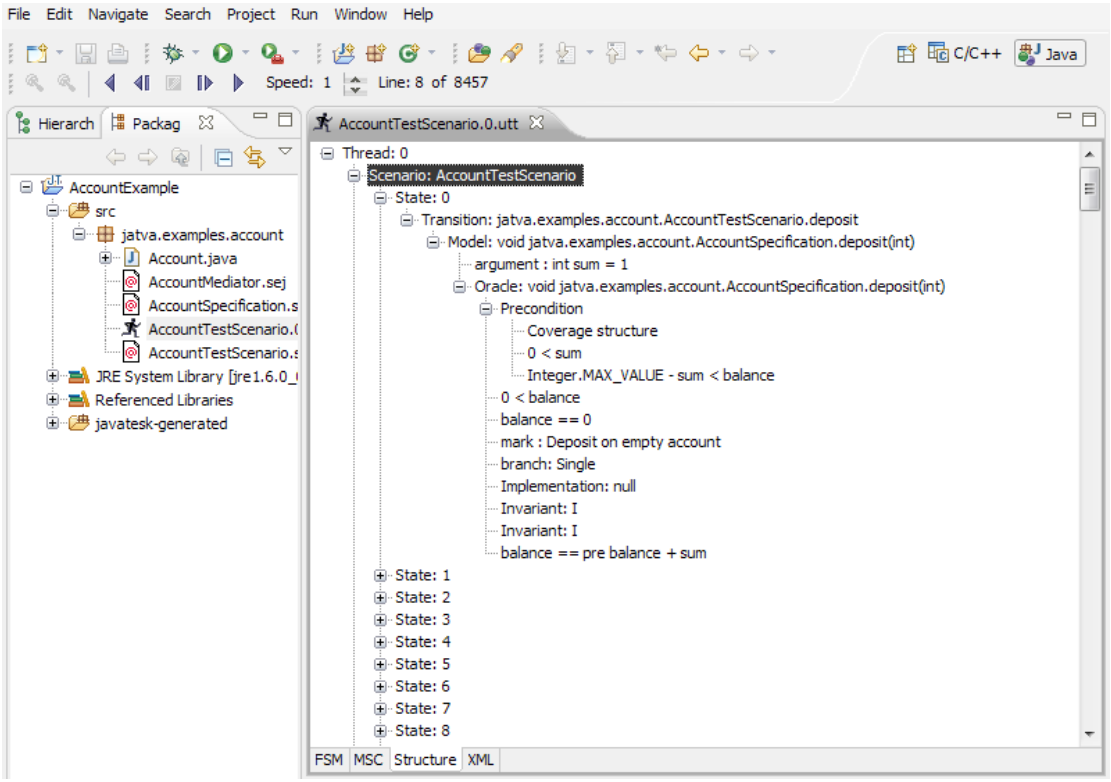


Figure 18. Structural representation of the trace.

To see the MSC diagram of the trace, open tab **MSC**. Scenario and specification method calls and behavior correctness verdict of the target class object are diagram messages.

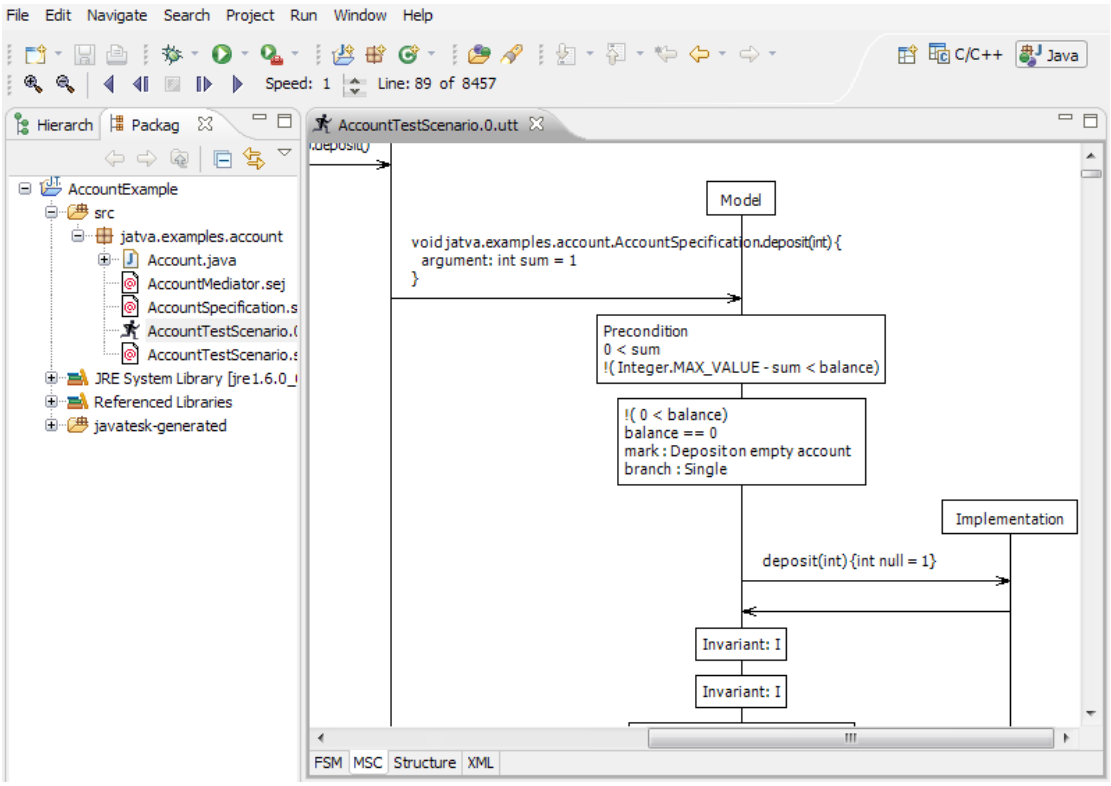


Figure 19. MSC representation of the trace.