

JavaTESK: первое знакомство

version 2.1

Введение

В данном документе рассматривается процесс разработки тестов с использованием инструмента JavaTESK на примере класса, реализующего методы для работы с банковским кредитным счетом.

Перед его прочтением рекомендуется ознакомиться с *JavaTESK Installation Instruction*.

В дальнейшем, класс, для которого разрабатываются тесты, будем называть *целевым классом*, а тестируемые методы данного класса — *целевыми методами*.

Пример состоит из следующих частей:

- [Описание целевого класса](#)
- [Создание проекта в среде разработки](#)
- [Спецификация целевого класса](#)
- [Разработка медиатора](#)
- [Разработка тестового сценария](#)
- [Выполнение тестов и анализ результатов](#)

Описание целевого класса

Для банковского кредитного счета определены два метода: зачисление и снятие денег со счета. Данные класса ограничены балансом счета. Также задан максимальный размер кредита, определяющий, на сколько баланс счета может опускаться ниже нуля.

Вместе с JavaTESK поставляется демонстрационный класс Account вместе с набором тестов. Проект, включающий все необходимые файлы, расположен в каталоге `examples/account`.

Данные класса Account:

- `public int balance` — текущий баланс счета.
- `static public int maximumCredit` — максимальный размер кредита данного класса.

Интерфейс класса Account состоит из следующих методов:

- `Account()` — конструктор, создает банковский кредитный счет с нулевым балансом;
- `void deposit(int sum)` — выполняет зачисление положительной суммы `sum` на счет, увеличивает баланс счета на указанную сумму;
- `int withdraw(int sum)` — выполняет снятие положительной суммы `sum` со счета, если разница текущего баланса и суммы `sum` укладывается в допустимый размер кредита, метод возвращает `sum`, иначе — 0.

Создание проекта в среде разработки

Проект, содержащий целевой класс Account, требования к нему и тесты, содержится в поставке JavaTESK, в проекте `account`. Чтобы ознакомиться с

ними, импортируйте проект в среду разработки Eclipse. Для этого выберите в меню команду **File/Import...** В появившемся диалоге **Import** разверните узел **General** и выберите пункт **Existing Projects into Workspace**.

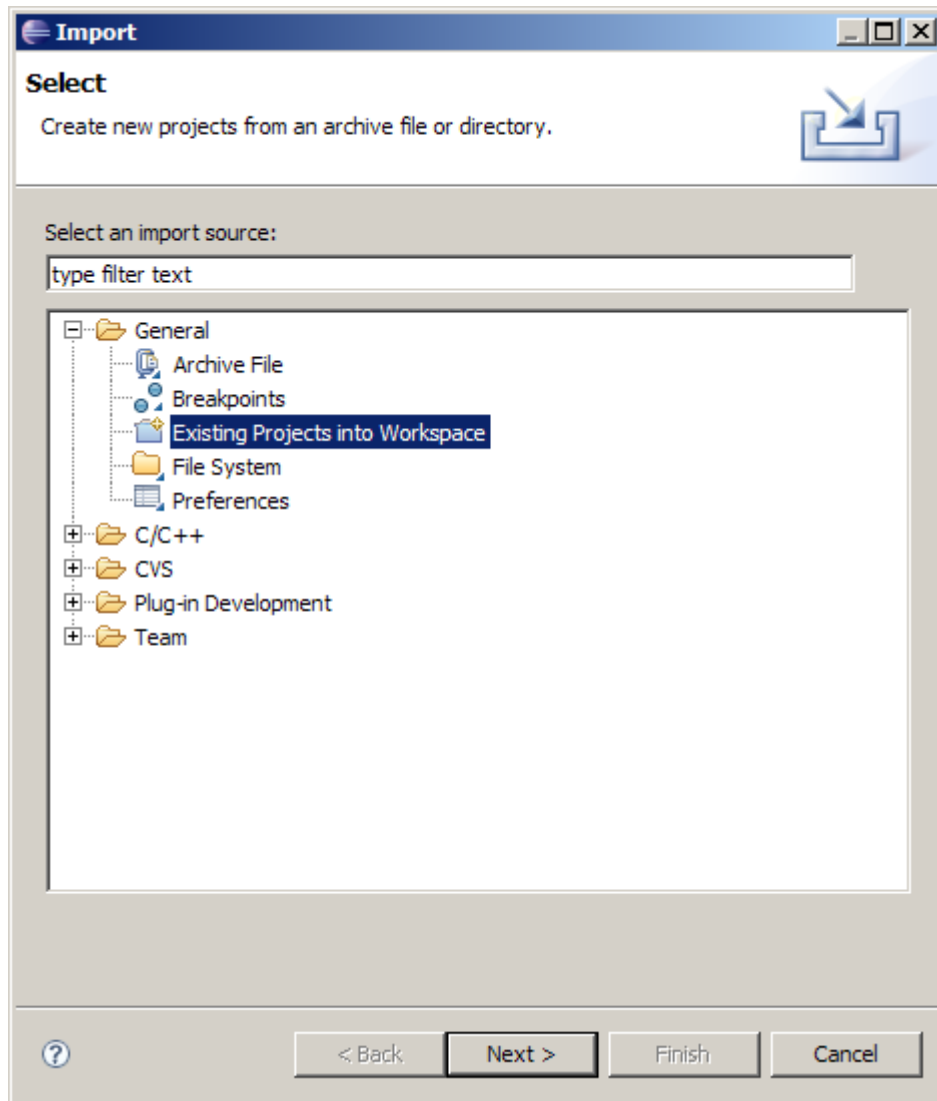


Рисунок 1. Выбор способа добавления файлов примеров в проект.

После нажатия кнопки **Next** диалог **Import** изменится, предлагая выбрать папку с файлами примеров. Нажмите на кнопку **Browse...**, соответствующую полю **Select root directory**.

Папка с файлами примеров находится в каталоге установки JavaTESK: найдя этот каталог, следует развернуть подкаталог `examples` и выбрать в качестве целевого каталога папку `account`.

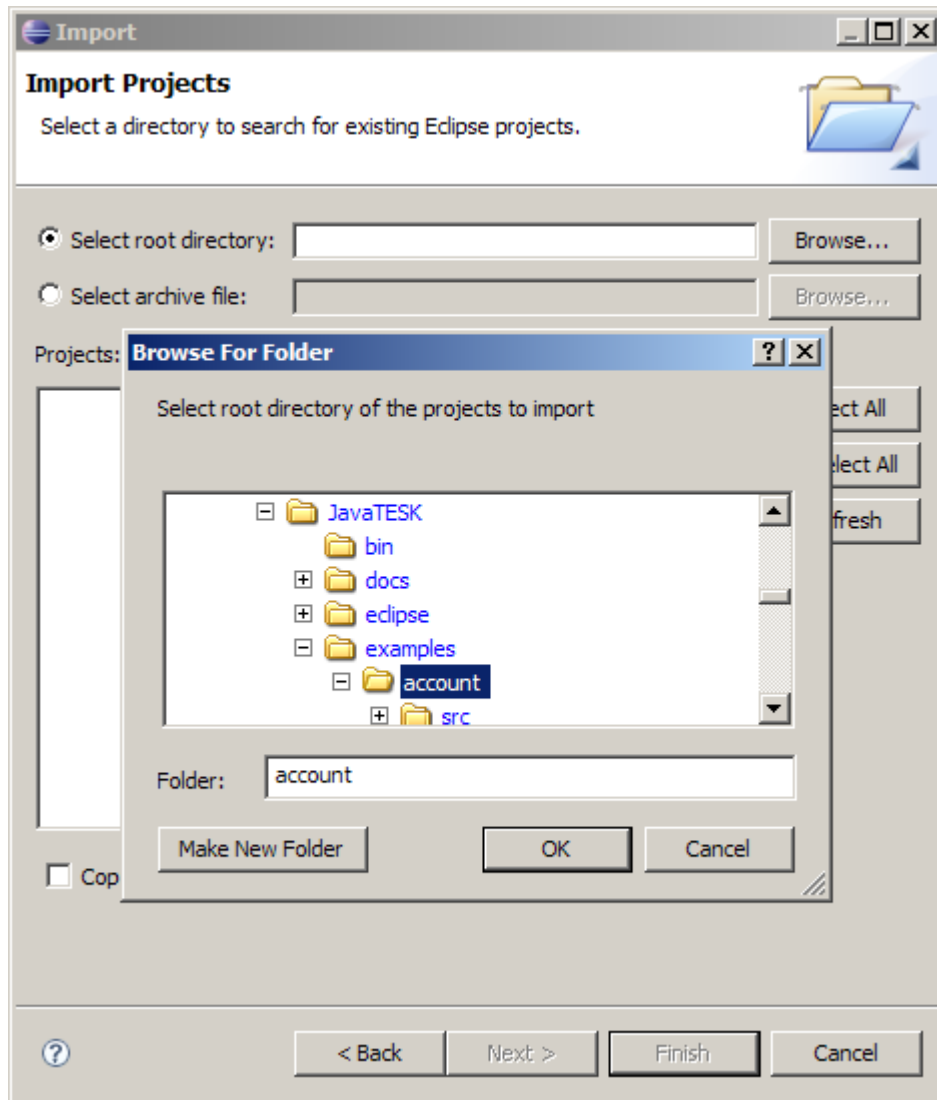


Рисунок 2. Выбор каталога с файлами примеров для импорта.

По нажатию **OK** среда найдет в этой папке проект `account`. Рекомендуется также отметить опцию **Copy projects into workspace**, чтобы работать с копией проекта, гарантируя неприкосновенность файлов исходных примеров.

По нажатию кнопки **Finish** проект будет импортирован.

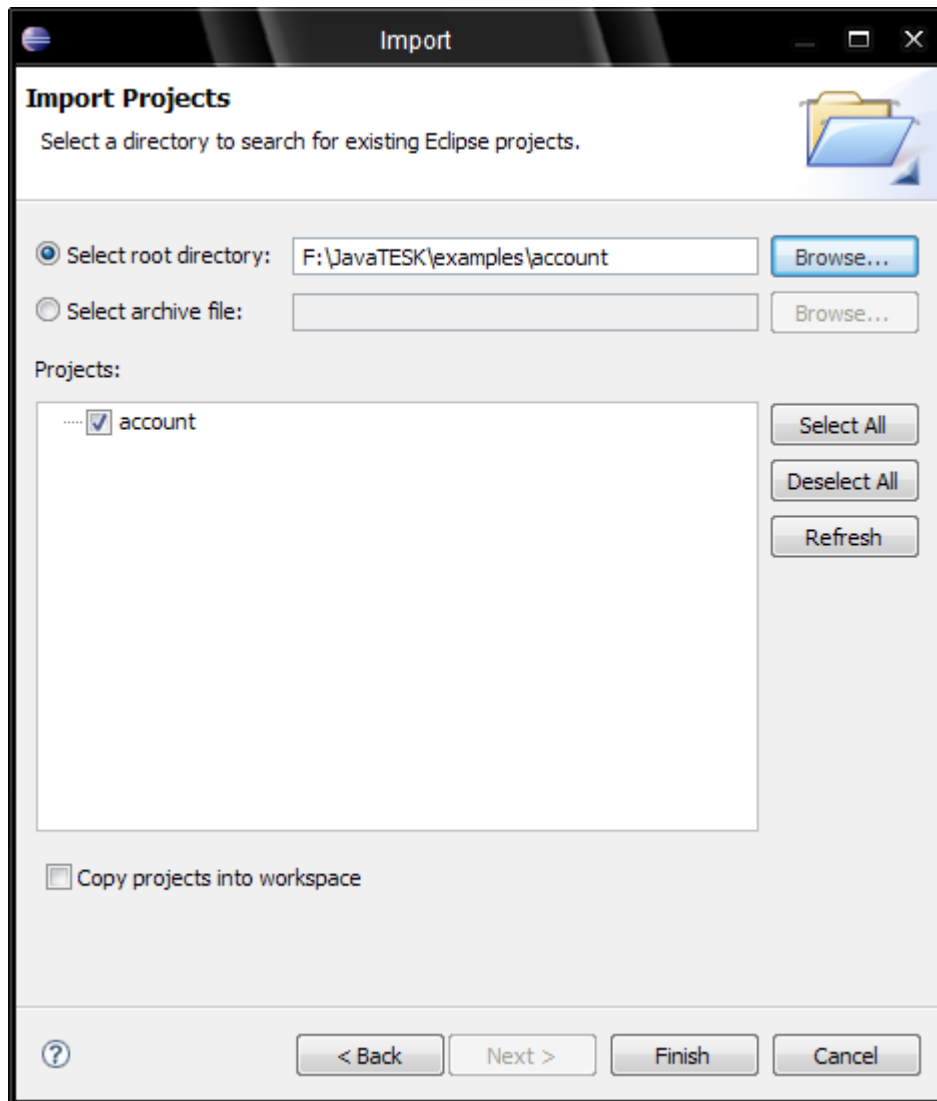


Рисунок 3. Выбор каталога с файлами примеров для импорта.

Проект готов к работе. Убедитесь, что структура каталогов проекта соответствует образцу:

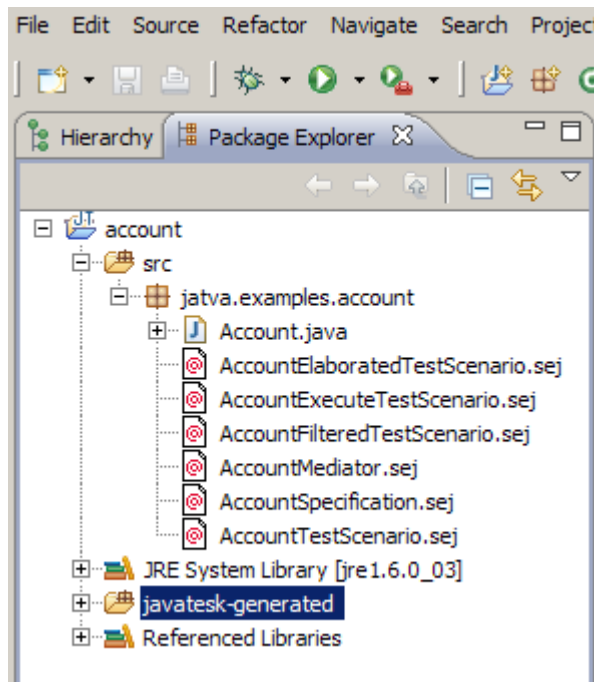


Рисунок 4. Структура каталогов готового проекта.

Спецификация целевого класса

Технология тестирования UniTESK, поддерживаемая JavaTESK, предполагает, что требования к целевому классу записаны в четкой недвусмысленной форме. Такая форма представления требований называется *формальной спецификацией*. Она может использоваться для автоматической генерации *оракулов* — компонентов тестовой системы, проверяющих соответствие между поведением целевых методов и требованиям к ним.

В JavaTESK формальные спецификации пишутся на специальном языке, являющемся расширением языка программирования Java. Данный язык позволяет описывать *функциональные требования*, которые определяют *функциональность* целевых методов, то есть то, что они должны делать.

Спецификации на языке JavaTESK очень похожи на Java код. Они оформляются в виде специальных классов, называемых *спецификационными*, которые располагаются в файлах с расширением `.sej`.

У нас в проекте уже есть готовый спецификационный класс `AccountSpecification`, рассмотрим его код, содержащийся в файле `AccountSpecification.sej`. Чтобы открыть этот файл, дважды кликните по его иконке в окне **Package Explorer**.

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

```
}

```

Спецификационный класс отличается от обычного Java-класса ключевым словом `specification`.

На основе полей целевого класса строится *модель данных*, для хранения которой создаются поля спецификационного класса. В случае класса `Account` соответствие простое, поэтому класс `AccountSpecification` обладает точно такими же полями — `public int balance` для хранения баланса счета и `static public int maximumCredit` для максимального кредита.

Ограничения на поведение целевых методов оформляются как методы спецификационных классов, помеченные модификатором `specification`. Такие методы называются *спецификационными методами*. Обычно спецификационный метод описывает поведение одного целевого метода и имеет такое же имя.

Рассмотрим спецификационный метод `deposit`, описывающий вклад денег на счет:

```
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;
    }
}
```

Тело спецификационного метода, которое описывает поведение целевого метода в форме пред- и постусловий.

Предусловие — это блок, помеченный ключевым словом `pre` и возвращающий результат типа `boolean`, зависящий от входных параметров и состояния объекта спецификационного класса. Предусловие задает область применения спецификационного метода — если предусловие возвращает `true`, при вызове метода можно ожидать от него корректного поведения, в противном случае результат работы метода может быть не определен.

В нашем примере предусловие показывает, что сумма вклада `sum` должна быть положительной, а результат сложения текущего баланса `balance` и суммы вклада `sum` не должен превосходить максимального допустимого значения типа `int`.

Предусловие спецификационного метода может быть опущено, что эквивалентно наличию предусловия, всегда возвращающего `true`:

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

Постусловие – это блок, помеченный ключевым словом `post` и возвращающий результат типа `boolean`. Постусловие анализирует значение параметров спецификационного метода, его результат, состояние объекта спецификационного класса и показывает, соответствует ли поведение метода ожиданиям.

В нашем примере постусловие показывает, что поведение метода `deposit` корректно, если баланс счета после вызова метода (`balance`) равен балансу счета до вызова метода (`pre balance`), увеличенному на сумму вклада `sum`.

Операторы `branch` и `mark` используются для определения *тестового покрытия*, о котором будет рассказано позже.

Теперь рассмотрим спецификационный метод `withdraw`, описывающий снятие денег со счета:

```

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
              ;
    }
  }
}

```

Предусловие показывает, что снимаемая сумма `sum` должна быть положительной.

С помощью второго условия `if` постусловие обрабатывает два варианта: когда снятие указанной суммы невозможно (условие выполняется) и когда снятие указанной суммы возможно (условие не выполняется). Постусловие показывает, что в первом случае баланс не должен измениться и метод должен вернуть 0. Во втором — баланс должен уменьшиться на снятую со счета сумму `sum`; эта сумма должна быть также и возвращаемым значением метода. Для обозначения

возвращаемого методом `withdraw` результата используется одноименный идентификатор.

Также язык JavaTESK предоставляет средства для описания ограничений на возможные значения полей спецификационных классов. Такие ограничения называются *инвариантами данных* и оформляются как специальные методы, помеченные модификатором `invariant`. Тип возвращаемого результата инварианта — `boolean`.

Для спецификационного класса `AccountSpecification` определен один инвариант данных `I`:

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

Данный инвариант показывает, что баланс не должен быть ниже максимального допустимого кредита.

Приведем полный текст спецификационного класса `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)
    {
        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)
    {
        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
                    ;
    }
}
}
}
}
}

```

Для того чтобы убедиться в корректности кода спецификационного класса, запустите сборку проекта командой **Project/Build Project** или запустите автоматическую сборку:

1. Вызвав командой **Project/Clean** диалог **Clean**, проставьте маркер **Clean all projects** или, проставив маркер **Clean projects selected below**, отметьте галочкой проект `account`.
2. Нажмите **ОК**.
3. Если код корректен, окно **Problems** (вызывается командой **Window/Show View/Problems**) останется пустым. Таким образом проверяется весь проект, корректность медиаторов и сценариев можно проверить точно так же.

Разработка медиатора

Спецификационный класс `AccountSpecification` описывает формальные требования к классу `Account`. Для того чтобы дать возможность тесту проверить соответствие между целевым и спецификационным классами, нужна некоторая связка между ними.

Для этой цели используются специальные компоненты тестовой системы, называемые *медиаторами*. Медиаторы оформляются в виде специальных классов, называемых *медиаторными*.

Файл примера `AccountMediator.sej` содержит медиаторный класс `AccountMediator`, который связывает спецификационный класс `AccountSpecification`

tion и целевой класс Account. Таким образом, определение медиатора должно каким-то образом включать явные указания на эти классы.

- Спецификационный класс AccountSpecification, для которого создается медиатор, обозначается в объявлении медиаторного класса:

```
mediator class AccountMediator implements AccountSpecification
```

Ключевое слово implements в Java обозначает реализацию объявляемым классом некоторого интерфейса. Аналогично, в определении медиаторного класса должен присутствовать медиаторный метод для каждого метода спецификационного класса. Также медиаторный класс наследует поля спецификационного класса, которые предназначены для хранения тестовой модели данных.

- Целевой класс Account обозначается в объявлении специального поля медиаторного класса:

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

Ключевое слово implementation говорит о том, что это поле предназначено для хранения тестового объекта целевого класса. К этому объекту и применяются тестовые воздействия.

Итак, спецификационный и целевой классы обозначены. Чтобы результаты тестирования заслуживали доверия, тестовая модель данных (заключенная в унаследованных от спецификационного класса полях) и данные тестируемого объекта (заключенные в полях целевого класса) должны строго соответствовать друг другу. Для этого в медиаторе предусмотрен блок синхронизации update, синхронизирующий поля медиатора с текущим состоянием объекта целевого класса. Эта синхронизация и должна гарантировать то, что поведение целевого класса и его спецификационной модели вызвано одними и теми же исходными данными. Рассмотрим блок updates медиаторного класса AccountMediator:

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

Так как поля целевого класса — стековые переменные, синхронизация достаточно проста, полям медиатора присваиваются значения соответствующих полей объекта целевого класса. В более запутанных случаях, например, при синхронизации полей, содержащих объекты других классов или структуры данных, реализация блока update может значительно усложниться.

Синхронизация данных необходима для проверки результатов работы методов целевого класса. Вызов этих методов также выполняется медиатором, с помощью медиаторных методов. Рассмотрим медиаторный метод `deposit`:

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

Объявление метода предварено ключевым словом `mediator`. Ключевое слово `implementation` в данном случае обозначает блок вызова целевого метода.

Метод `deposit` ничего не возвращает, но принимает один параметр. При необходимости (если параметры обладают сложным внутренним устройством, которое отлично для спецификационного и целевого класса), медиаторный метод может содержать код, преобразующий передаваемые параметры в вид, приемлемый для вызываемого метода.

Значение, возвращаемое целевым методом, аналогично обработке параметров, может преобразоваться в вид, приемлемый для спецификационного метода. Так как вызов происходит в блоке `implementation`, код преобразования должен находиться внутри этого блока. Результат преобразования возвращается в спецификационный метод оператором `return` также в блоке `implementation`.

Код медиаторного метода `withdraw`:

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

Приведем полный текст медиаторного класса `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;
    }
}
}

```

Разработка тестового сценария

Тестовые сценарии разрабатываются для достижения некоторой цели тестирования. Обычно такая цель формулируется в терминах тестового покрытия, например, требуется достичь покрытия 70% строк исходного кода целевого класса. В JavaTESK критерий покрытия описывается в терминах покрытия спецификации.

Предположим, наша цель – достичь 100% покрытия ветвей функциональности спецификационного класса `AccountSpecification`. Это означает, что мы должны выполнить набор тестов, покрывающих все ветви функциональности, определенные в постусловиях спецификационных методов данного класса с помощью операторов `branch`. Выделены следующие ветви:

- `Single`
- `TooLargeSum`
- `Normal`

Тестовые сценарии оформляются в виде специальных классов, называемых *сценарными*. В проекте примера тестовый сценарий содержится в файле `AccountTestScenario.sej`.

Определение сценарного класса должно содержать ключевое слово `scenario`:

```
scenario class AccountTestScenario
```

Сценарный класс содержит объект спецификационного класса, чьи методы будут вызываться в процессе тестирования:

```
protected AccountSpecification objectUnderTest;
```

Указание целевого класса и медиатора производится в конструкторе класса `AccountTestScenario`: в нашем примере для этого предназначен метод `configureMediators`, возвращающий инициализационное значение для поля `objectUnderTest`:

```

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

public static AccountSpecification configureMediators ()
{

```

```

AccountSpecification result = mediator AccountMediator
                               ( targetObject = new Account() );
result.attachOracle();
return result;
}

```

Ключевое слово `mediator` служит в данном случае для создания объекта медиаторного класса. Отличие от ключевого слова `new` состоит в том, что перед созданием происходит проверка: если медиатор типа `AccountMediator`, инициализированный объектом класса `Account`, уже присутствует, то новый медиатор не создается, а используется старый.

Вызовы методов объекта `objectUnderTest` называются *тестовыми воздействиями* и производятся в *сценарных методах*. Рассмотрим метод `deposit`:

```

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

```

Чтобы ограничить число тестовых воздействий, для баланса счета введен некоторый максимум (поле `maxBalance`). Предполагается, что метод `deposit` при каждом вызове увеличивает баланс на 1, таким образом, конечное число вызовов должно привести к достижению установленного максимума.

В сценарном методе `withdraw` происходит перебор параметров:

```

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

```

Обычные циклы работают до тех пор, пока не нарушится некоторое внутреннее условие. Оператор `iterate` аналогичен циклу, с тем отличием, что при каждом вызове сценарного метода, в котором заключен оператор, выполняется только одна итерация цикла.

Таким образом, оператор `iterate` позволяет ограничивать каждый вызов сценарного метода единственным тестовым воздействием, одновременно задавая параметры множества таких воздействий в коде одного и того же сценарного метода.

После того, как определен объект тестирования и сценарные методы, в метод `main` нужно включить инструкции, запускающие тест:

```

AccountTestScenario myScenario = new AccountTestScenario();
myScenario.run();

```

Приведем полный текст сценарного класса 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;
}
}

```

Выполнение тестов и анализ результатов

Для запуска тестового сценария нужно выбрать соответствующий файл в окне **Package Explorer** и нажать сочетание клавиш **Ctrl+F11**, кликнуть на нем правой кнопкой, и выбрать элемент контекстного меню **Run As/JavaTESK Test**.

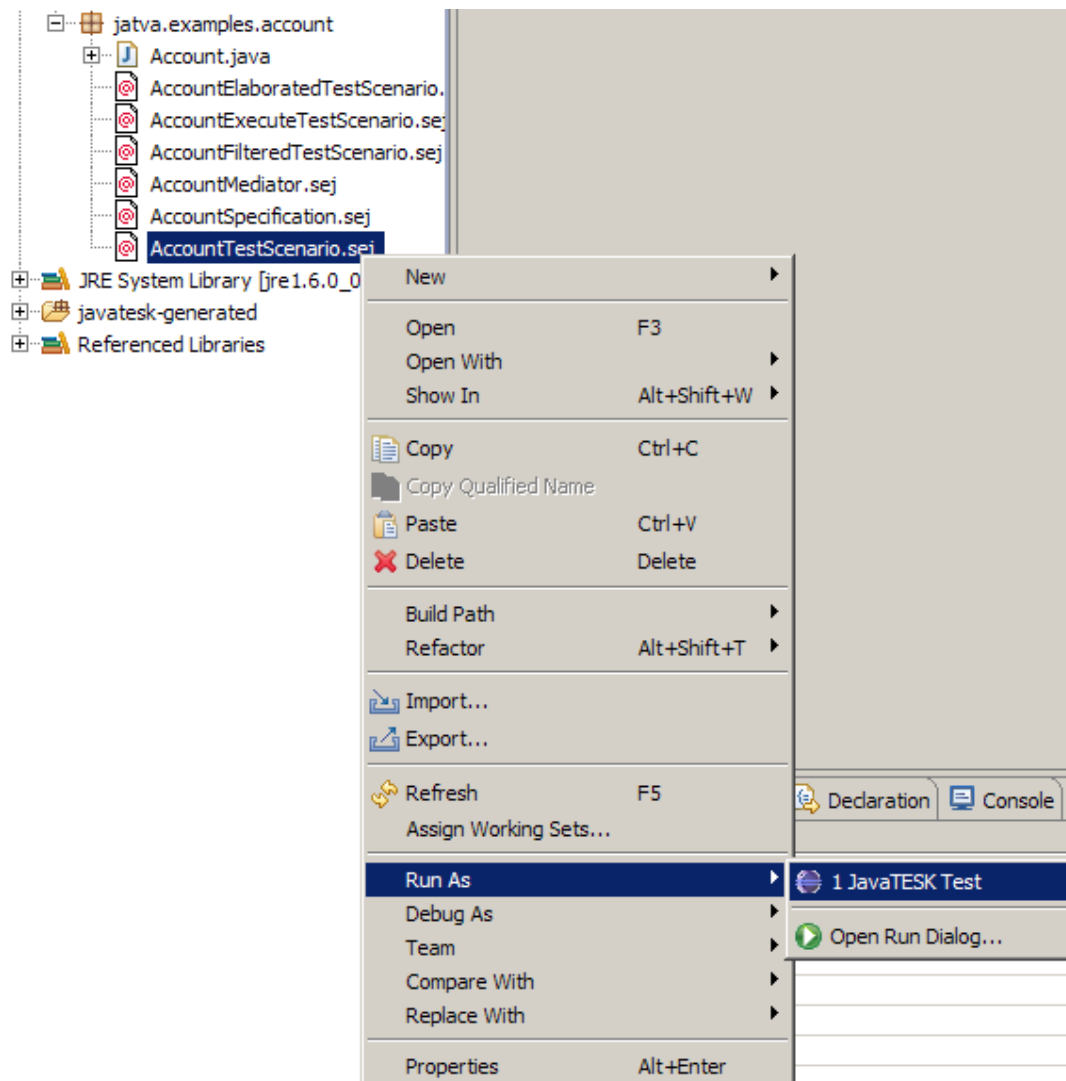


Рисунок 5. Запуск сценарного класса AccountTestScenario.

В случае успешного выполнения теста создается *файл трассы*, имеющий расширение `.utt`. Его имя состоит из имени сценарного класса и номера трассы, разделенных точкой. Файл трассы появляется в проекте на том же уровне иерархии, что и выполненный сценарий.

Убедитесь, что файл трассы создан и вкладка окна **Package Explorer** выглядит как показано на рисунке.

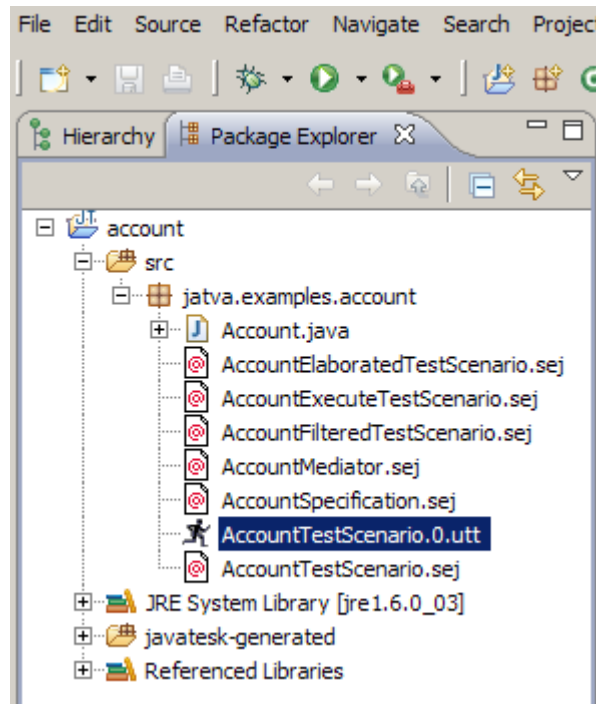


Рисунок 6. Файл трассы после запуска сценария AccountTestScenario.

Файл трассы содержит исходную информацию для генерации отчетов о проведенном тестировании.

Отчеты предоставляют пользователю информацию об обнаруженных в процессе тестирования ошибках и уровне достигнутого тестового покрытия.

JavaTESK поддерживает два вида отчетов:

- [HTML отчеты](#)
- [Представления трассы](#)

HTML отчет

HTML отчет представляет собой набор HTML документов, содержащих информацию о проведенном тестировании.

Для генерации HTML отчета нужно кликнуть в окне **Package Explorer** правой кнопкой мыши на соответствующем файле трассы, затем выбрать команду **Generate Report**.

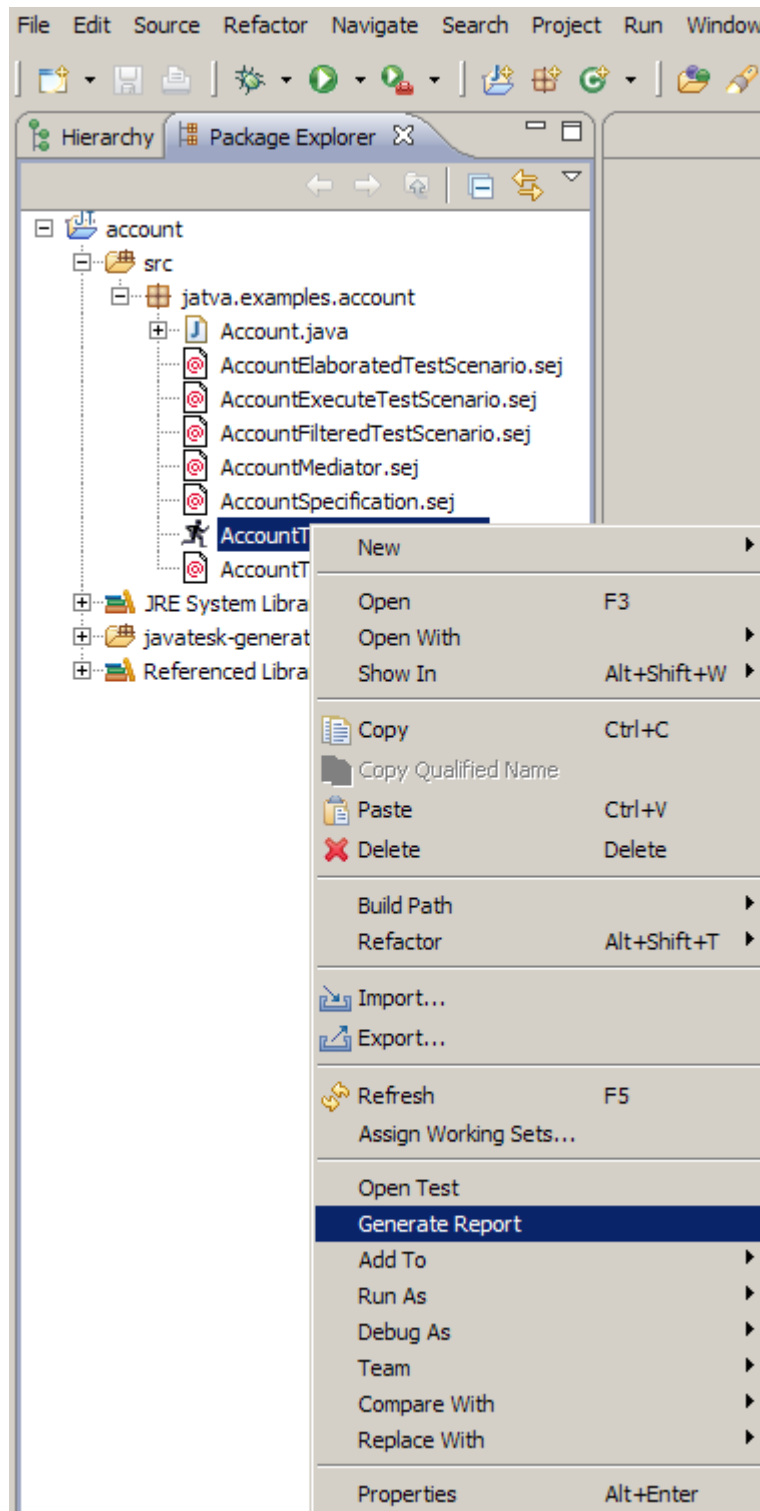


Рисунок 7. Генерация HTML отчета.

Будет открыт диалог **Generate UniTESK Report**, предназначенный для настройки HTML отчетов. Вкладка **Generate** содержит параметры генерации файлов, тогда как вкладка **Report** позволяет настроить содержание отчета. Для того, чтобы после генерации отчет был открыт в системном браузере, а не в Eclipse, нужно отметить галочку **Use external browser to view report**.

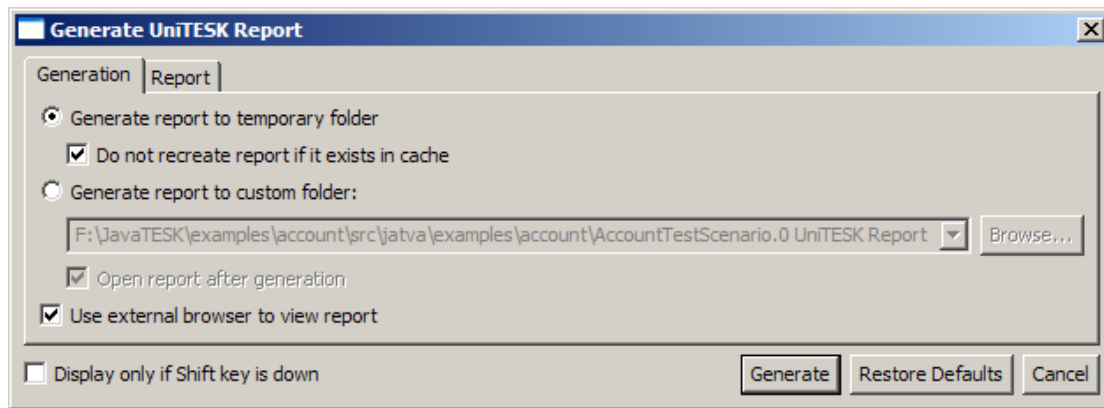


Рисунок 8. Настройка генерации HTML отчета.

Согласно выбору, сгенерированный HTML отчет будет открыт внутри Eclipse или в браузере по умолчанию. Таблица на первой странице (**Overview**) представляет информацию о полноте покрытия участвовавших в тесте пакетов и пространств имен. Предусмотрены следующие столбцы:

- **branches** — уровень покрытия ветвей функциональности
- **marks** — уровень покрытия помеченных путей
- **predicates** — уровень покрытия предикатов
- **disjuncts** — уровень покрытия дизъюнктов
- **states/transitions** — количество состояний/количество переходов между состояниями

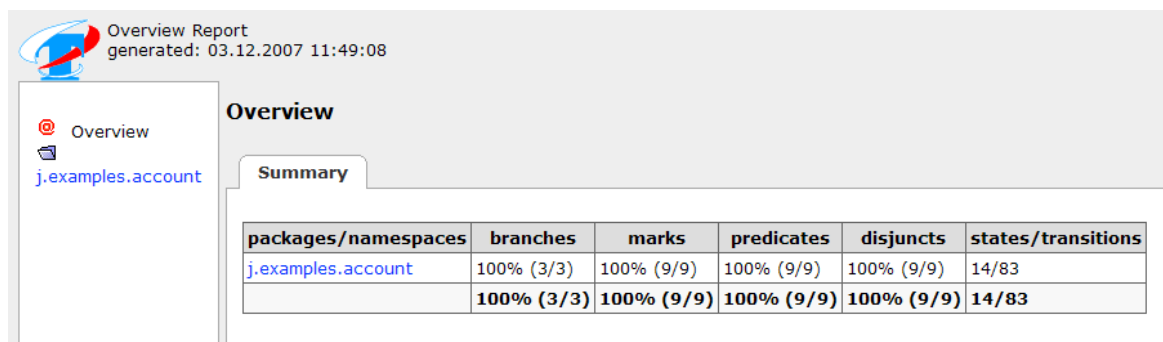


Рисунок 9. Страница Overview HTML отчета.

Для получения информации о результатах тестирования класса Account, выберите элемент меню **j.examples.account**. Информация из таблицы **Overview** будет разделена между таблицами **scenarios** и **specifications**: количества состояний — характеристики сценария, а показатели покрытий — характеристики спецификаций.

Package/Namespace Summary Report
generated: 03.12.2007 11:49:08

jatva.examples.account

Summary

scenarios	states/transitions			
AccountTestScenario	14/83			
	14/83			
specifications	branches	marks	predicates	disjuncts
AccountSpecification	100% (3/3)	100% (9/9)	100% (9/9)	100% (9/9)
	100% (3/3)	100% (9/9)	100% (9/9)	100% (9/9)

Рисунок 10. Страница сводной информации о результатах тестирования.

Подэлемент меню **AccountTestScenario** дает доступ к таблице **Scenario Transitions Report**, в которой перечислены все переходы между состояниями тестовой модели, со следующими характеристиками:

- **states** — начальные состояния (до перехода). Переходы группируются по этому параметру — между группами границы ячеек таблиц двойные.
- **transitions** — методы, вызвавшие переход и значения итерационных переменных операторов `iterate`, использованные при вызове
- **end states** — конечные состояния (после перехода)
- **hits/fails** — количество переходов, совершенных описанным образом/количество ошибок, обнаруженных при переходах

Scenario Certain Transitions Report
generated: 03.12.2007 11:49:08

jatva.examples.account.AccountTestScenario

Certain trans All trans

			states/trans/failures
			14/83
states	transitions	end states	hits/fails
-1	jatva.examples.account.AccountTestScenario.withdraw (int i = 3)	-1	1
	jatva.examples.account.AccountTestScenario.withdraw (int i = 4)		1
	jatva.examples.account.AccountTestScenario.withdraw (int i = 5)		1
-1	jatva.examples.account.AccountTestScenario.withdraw (int i = 1)	-2	2
-1	jatva.examples.account.AccountTestScenario.withdraw (int i = 2)	-3	1
-1	jatva.examples.account.AccountTestScenario.deposit ()	0	13
-2	jatva.examples.account.AccountTestScenario.deposit (-1	10

Рисунок 11. Страница с данными о сценарных переходах HTML отчета.

Подэлемент меню **AccountSpecification** открывает таблицу **Specification Summary Report**, в которой параметры покрытия детализированы для каждого спецификационного метода:

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)

Рисунок 12. Сводная таблица покрытия спецификационных методов.

По клику на любом из методов будет открыта расширенная таблица покрытия:

- **branches** — покрытые ветви (в коде спецификационного метода обозначены ключевым словом `branch`)
- **marks** — метки вариантов прохода ветви (обозначены ключевым словом `mark`)
- **predicates** — условия, которым соответствуют метки `mark` той же строки таблицы
- **disjuncts** — логические значения элементов пред- и постусловий, соответствующие варианту прохода ветви
- **hits/fails** — количество проходов, произведенных по данной ветви и количество ошибок, обнаруженных при этих проходах

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 $!(0 < \text{balance}) \ \&\& \ !(\text{balance} == 0)$

predicate1 $!(0 < \text{balance}) \ \&\& \ \text{balance} == 0$

predicate2 $0 < \text{balance}$

prime formulas

f0 $0 < \text{sum}$

f1 $\text{Integer.MAX_VALUE} - \text{sum} < \text{balance}$

f2 $0 < \text{balance}$

f3 $\text{balance} == 0$

f4 $\text{balance} == \text{pre balance} + \text{sum}$

Рисунок 13. Таблица с результатами тестирования спецификационного метода `deposit`.

Аналогичная таблица для метода `withdraw`; варианты, принадлежащие разным ветвям `branch` группируются соответственно.

Method Coverage Report
generated: 03.12.2007 11:49:08

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	

Рисунок 14. Таблица с результатами тестирования спецификационного метода `withdraw`.

Заметим, что данные отчета свидетельствуют о полном покрытии целевой функциональности: все намеченные варианты поведения были корректно протестированы.

Представления трассы

JavaTESK поддерживает четыре вида представления трассы:

- **XML** (*XML представление*) — представление трассы в формате XML, текстовый файл, генерируемый тестовой средой
- **Structure** (*структурное представление*) — представление трассы в виде дерева из структурных элементов трассы
- **MSC** (*MSC представление*) — представление трассы в виде MSC (*Message Sequence Charts*) диаграммы
- **FSM Model** (*представление в виде автомата*) — представление одного из выполнявшихся в тесте сценария в виде графа состояний и переходов конечного автомата

В IDE Eclipse встроена возможность просмотра всех четырех видов представления: по двойному клику открывается представление по умолчанию (FSM), а переключение между представлениями осуществляется с помощью закладок в нижней части окна.

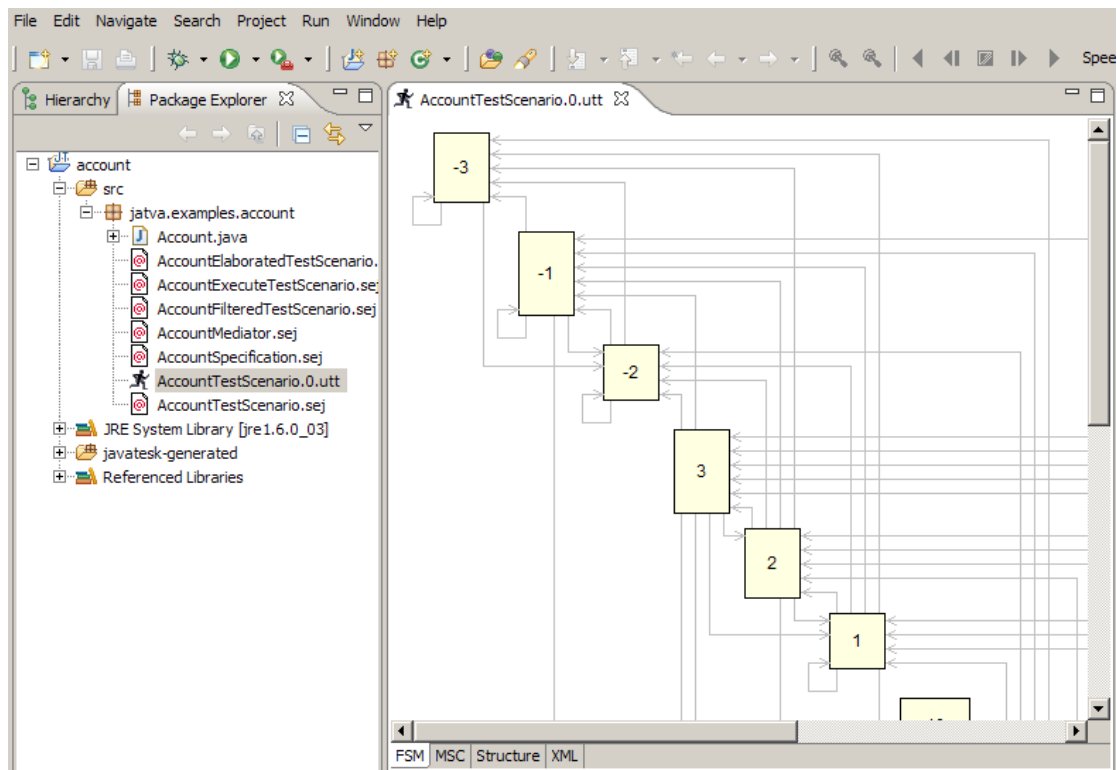


Рисунок 15. FSM-представление трассы.

В данном представлении трасса показана в виде графа состояний конечного автомата, заданного в сценарном классе `AccountTestScenario`. Подробности о переходе можно узнать, кликнув на стрелке, которая его обозначает. Линия будет выделена цветом, а под стрелкой будет выведено полное имя сценарного метода и соответствующие переходу значения итерационных переменных.

JavaTESK предоставляет возможность проигрывания представления трассы как вперед, так и назад. По умолчанию при нажатии кнопки **Play Forward** воспроизведение работы автомата начнется с начального состояния; если же перед запуском выделить кликом мыши некоторое состояние или стрелку, направленную на него, то воспроизведение начнется с этого места. Текущие состояния и переходы выделяются цветом.

Задание скорости воспроизведения производится с помощью счетчика **Speed**. Остальные кнопки управления действуют соответственно иконкам: позволяют проигрывать запись в обратном направлении, останавливать воспроизведение до окончания работы автомата и просматривать запись пошагово в обоих направлениях.

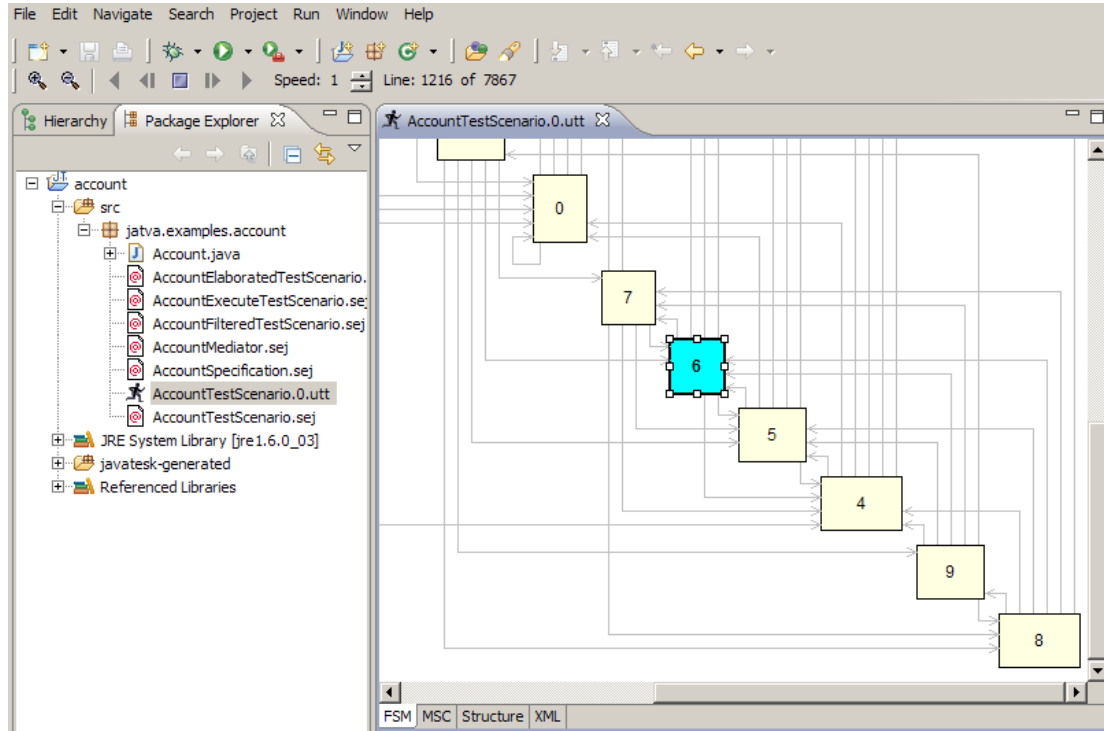


Рисунок 16. FSM-представление трассы в режиме воспроизведения.

Активировав вкладку **XML**, можно увидеть XML представление трассы. В данном представлении трасса показана такой, как она хранится в файле с расширением `.utt`.

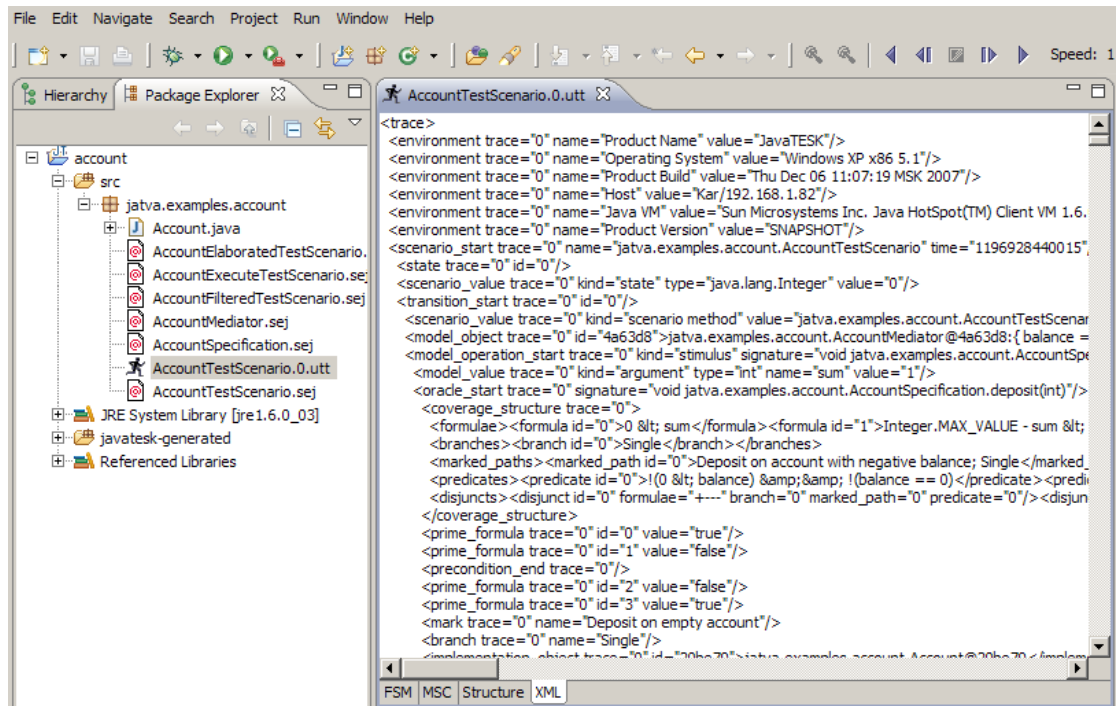


Рисунок 17. XML представление трассы.

Для просмотра структурного представления трассы перейдите на вкладку **Structure**. Трасса в нем представлена в виде дерева из структурных элементов.

В корне дерева находится узел **Trace**, соответствующий выбранному файлу трассы. Его потомками являются узлы **Thread**, описывающие потоки, создаваемые тестом. В нашем примере создается один поток с идентификатором **0**. Его потомками являются узлы **Scenario**, описывающие сценарные классы, запущенные в данном потоке. В нашем примере это единственный сценарный класс `AccountTestScenario`. Его потомками являются узлы **State**, описывающие состояния конечного автомата, перечисленные в порядке их обхода в процессе тестирования. Для каждого состояния указан его идентификатор и переход **Transition**. Переход характеризуется сценарным методом (`deposit` из класса `AccountTestScenario`) и моделью целевого класса **Model**, в которой указан спецификационный метод (`AccountSpecification.deposit (int)`) и узел **Oracle**, содержащий выполняемые оракулом проверки.

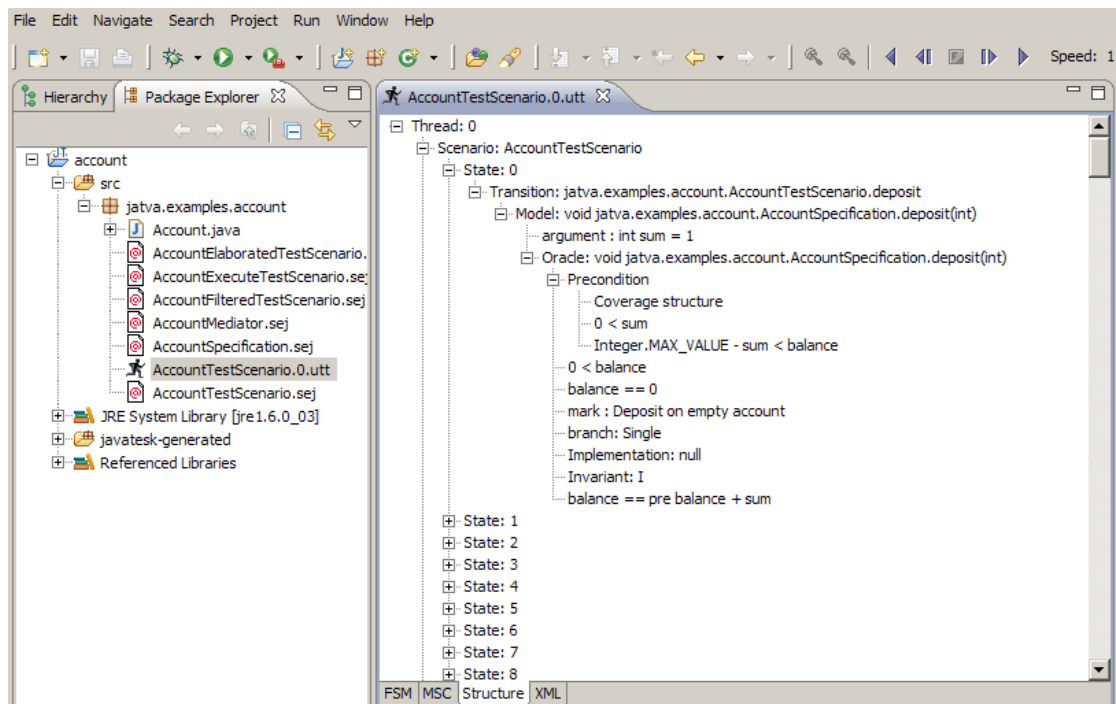


Рисунок 18. Структурное представление трассы.

Для просмотра трассы в виде MSC диаграммы, перейдите на вкладку **MSC**. Сообщениями на диаграмме являются вызовы сценарных и спецификационных методов, а также вердикт о правильности поведения объекта целевого класса.

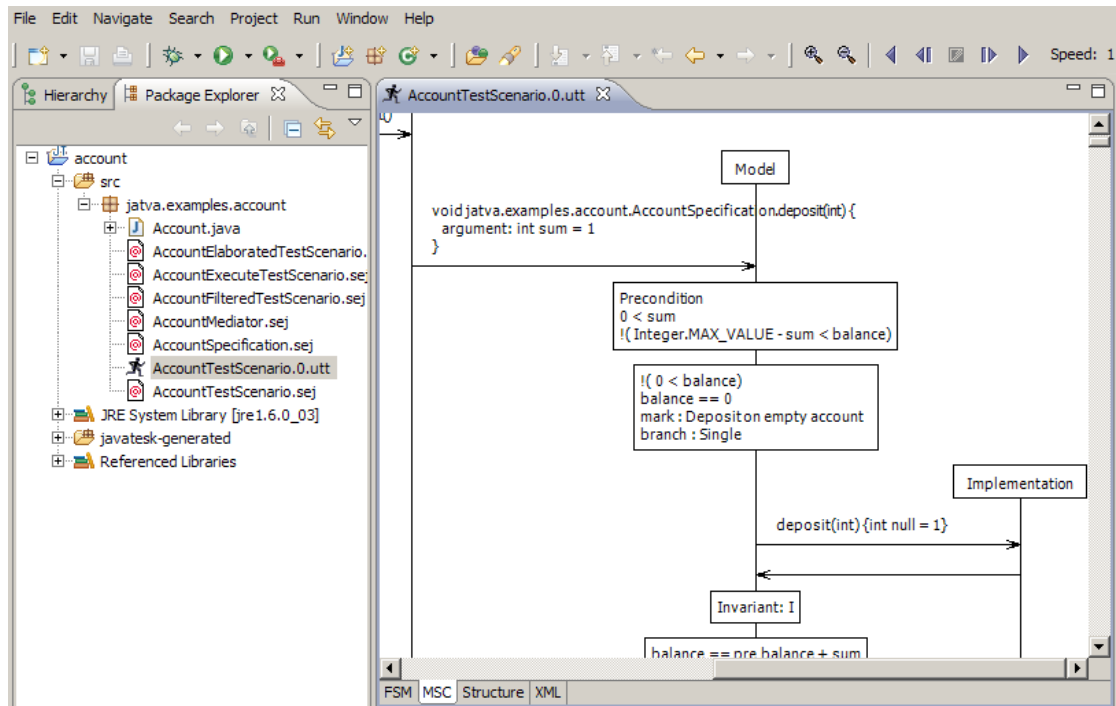


Рисунок 19. MSC представление трассы.