JavaTESK:
Быстрое знакомство

Содержание

Введение

Данный документ знакомит с основными понятиями JavaTESK и языка JavaTESK, предоставляя возможность быстрого старта разработки тестов в среде JavaTESK.

JavaTESK реализует технологию разработки тестов UniTesK программного обеспечения, написанного на языке программирования Java. UniTesK — промышленная технология автоматизированной разработки тестов, основанная на формальных методах.

Данная технология поддерживает разработку тестов для *функционального тестирования*. Функциональное тестирование обеспечивает проверку поведения тестируемой программы на соответствие *функциональным требованиям*.

Любая программная система предоставляет интерфейс, через который окружение взаимодействует с этой системой. Поведение системы соответствует функциональным требованиям, если любые наблюдаемые снаружи результаты ее работы согласуются с этими требованиями. То есть функциональные требования *не определяют* как должна быть реализована программная система, они определяют какие видимые снаружи результаты должны производиться при взаимодействии окружения с системой через ее интерфейс.

Автоматизация разработки функциональных тестов, проверяющих выполнение функциональных требований, возможна только при строгом формальном определении требований. Здесь под “формальным” подразумевается способ определения, при котором требования имеют однозначную интерпретацию и могут обрабатываться компьютером. То есть, в данном случае, разница между неформальными и формальными спецификациями требований скорее подобна разнице между естественными языками и языками программирования, чем разнице между языками программирования и математическими формальными языками.

JavaTESK реализует технологию UniTesK для программного обеспечения, реализованного на языке программирования Java. В JavaTESK используется язык *JavaTESK* – специально разработанное спецификационное расширение языка программирования Java. JavaTESK расширяет язык Java нотацией для определения пред- и постусловий, критериев покрытия, медиаторов и тестовых сценариев. JavaTESK позволяет разработчикам тестов определять и генерировать компоненты тестовой системы, из которых могут собираться качественные тесты. Так же JavaTESK позволяет определять полностью независимые от реализации спецификации и сценарии, что дает возможность их повторного использования.

Набор инструментов JavaTESK включает *транслятор* JavaTESK в Java, *библиотеку поддержки тестовой системы*, *библиотеку спецификационных типов* и *генератор тестовых отчетов*.

Транслятор JavaTESK в Java позволяет генерировать компоненты тестов из спецификаций, медиаторов и тестовых сценариев. *Библиотека поддержки тестовой системы* предоставляет *обходчик* — реализацию на языке Java алгоритмов построения тестовой последовательности, и поддержку трассировки выполнения тестов. *Библиотека спецификационных типов* поддерживает типы интегрированные со стандартными функциями создания, инициализации, копирования, сравнения и уничтожения данных этих типов. Так же библиотека содержит набор уже определенных спецификационных

типов. *Генератор тестовых отчетов* предоставляет возможность автоматического анализа трассы выполнения теста и генерацию различных содержательных тестовых отчетов.

Соглашения о форматировании

Курсивом выделяются термины основных понятий и части текста с важной информацией.

“*Курсивом в двойных кавычках”* выделяются ссылки на другие документы по JavaTESK.

```
Примеры на JavaTESK представлены в отформатированных абзацах.
```

`Шрифтом с фиксированной шириной` выделяются фрагменты кода, появляющиеся в основном тексте. **`Полужирным шрифтом с фиксированной шириной`** — ключевые слова JavaTESK.

Полужирный шрифт используется для выделения элементов меню, команд и имен файлов и каталогов.

Другие документы

Дополнительную информацию по JavaTESK и поддерживаемой технологии разработки тестов можно найти в других документах, включенных в набор документации по JavaTESK: *“JavaTESK: Руководство пользователя ”* и *“JavaTESK: Описание языка JavaTESK”*. Сайт по UniTesK http://www.unitesk.com/ содержит информацию по UniTesK, JavaTESK и другим инструментам, поддерживающим UniTesK.

Так же с любыми вопросами по технологии UniTesK и использованию JavaTESK можно обращаться по электронному адресу support@unitesk.com.

Пример тестируемой системы: Банковский кредитный счет

Предполагается, что на Вашем компьютере установлен JavaTESK. Если это не так, то установите инструмент, следуя указаниям в документе "*JavaTESK: Инструкция по установке и использованию*".

В документе рассматривается процесс разработки теста с использованием JavaTESK на примере разработки теста для системы, реализующей функциональность банковского кредитного счета: вклад и снятие денег со счета при заданном максимально допустимом размере кредита.

Кредитный счет реализован как класс `Account`, определенная в файле **account.java** из пакета `jatva.examples.account`.

Интерфейс класса `Account` состоит из следующих методов:

`Account()` – конструктор, создает банковский кредитный счет с нулевым балансом

`void deposit(int sum)` – выполняет внесение 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 укладывается в допустимый размер кредита, уменьшает баланс счета на указанную сумму и возвращает `sum`, иначе возвращает **0**, не изменяя баланса

Текущий баланс счета хранится в поле `int balance` данного класса, а максимальный допустимый размер кредита задан статическим полем `int maximumCredit`.

Допустимый размер кредита должен быть не меньше нуля.

Далее в документе демонстрируется как, используя JavaTESK, разработать тест для системы банковского кредитного счета, выполнить тестирование и проанализировать полученные результаты.

Ниже описывается разработка теста, которая состоит из следующих шагов:

Разработка спецификации тестируемой системы

Разработка медиаторов

Разработка тестового сценария

Выполнение теста

Анализ результатов тестирования.

Создание проекта в среде разработки

Для создания проекта, который будет содержать целевой класс `Account`, его спецификацию и тесты, запустите среду разработки, в которую интегрирован JavaTESK (здесь это будет Eclipse 3.1), и создайте в ней новый проект, называемый `AccountExample`.

Для этого выберите пункт меню **File/New/Project**. В появившемся окне **New Project** нажмите пункт **Java Project** (он нажат по умолчанию). Затем нажмите кнопку **`Next`**.

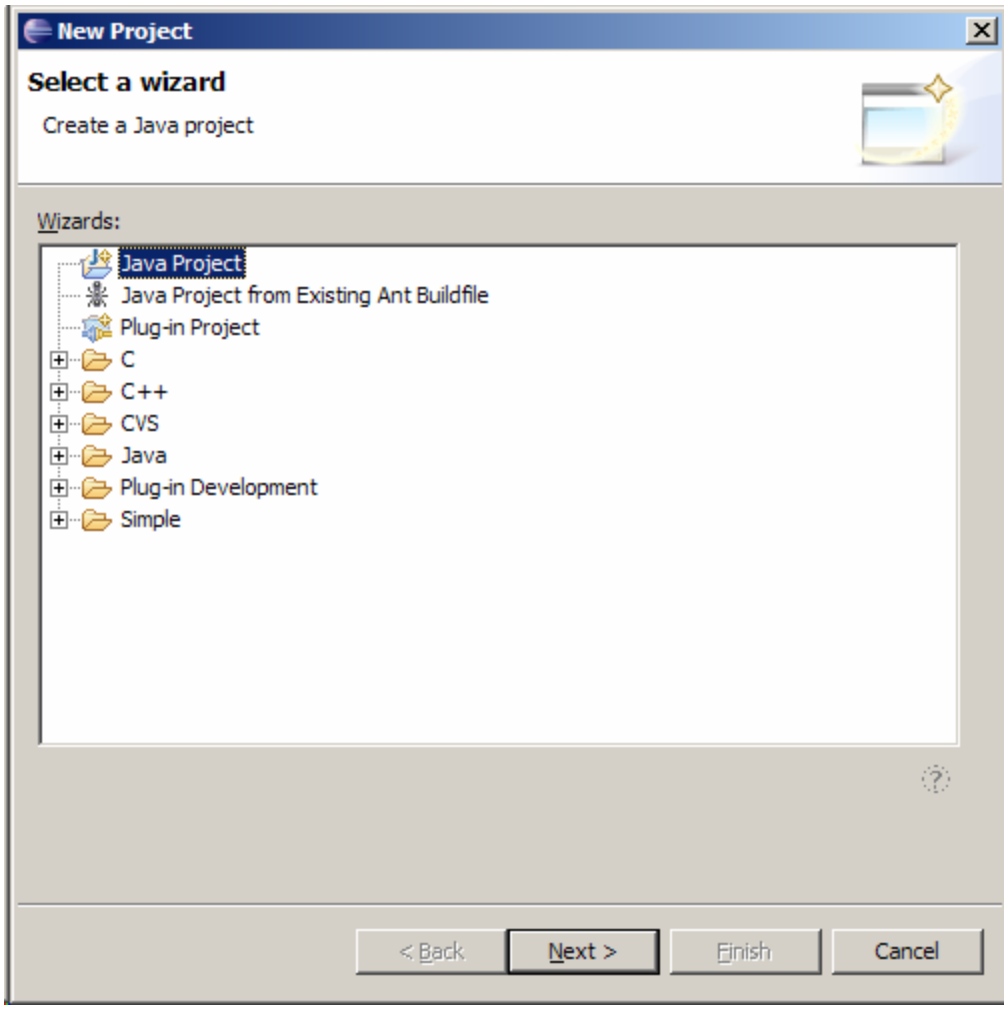

Рисунок 1. Создание проекта AccountExample.

Теперь необходимо указать имя создаваемого проекта и некоторые дополнительные параметры. Поэтому в поле **Project name:** вводим имя проекта `AccountExample`.

Сразу же за этим полем расположена группа из двух переключателей, помеченная **Contents**. В ней можно выбрать **Create new project in workspace**. Оно означает то, что файлы создаваемого проекта будет расположены в папке рабочего пространства (workspace).

Следующая группа переключателей – **JDK Compliance** – отвечает за версию компилятора Java. Убедитесь в том, что выбрана версия Java 5.0 : если выбран первый переключатель,

то в его строке должно быть написано **(Currently 5.0)**. Если это не так, выберите второй переключатель. В его строке будет выпадающий список. Выберите в нем **5.0**. Если такого пункта у Вас нет, то Вам необходимо установить JDK 1.5.

Третью группу переключателей можно оставить, как есть.

Нажмите на кнопку **Finish**.

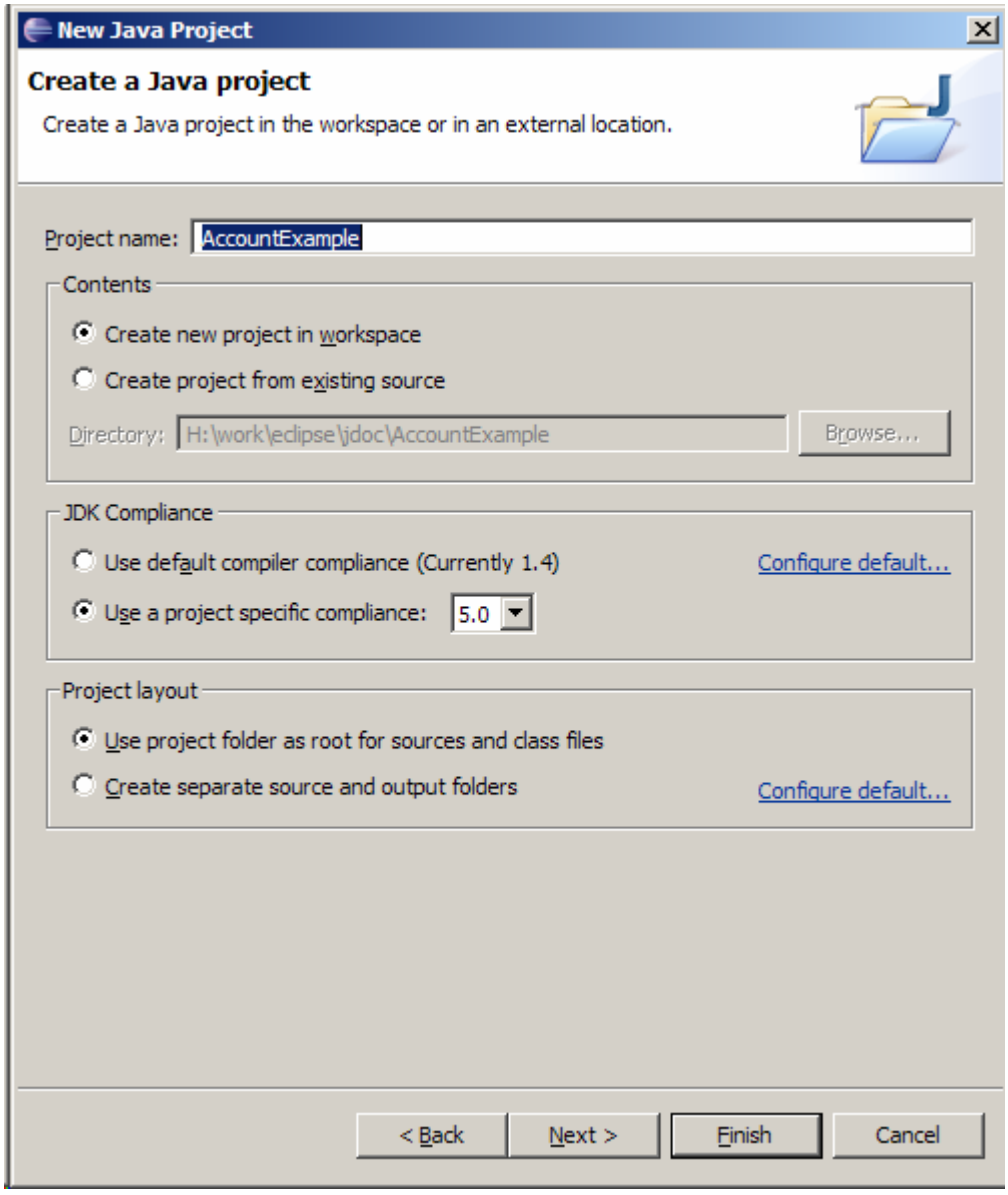

Рисунок 2. Задание свойств проекта AccountExample.

Если у Вас не открылось окно проекта, нажмите на большую стрелку в верхнем правом угла (при наведении указателя мыши на него появляется надпись **Workbench. Go to the workbench**).

…..

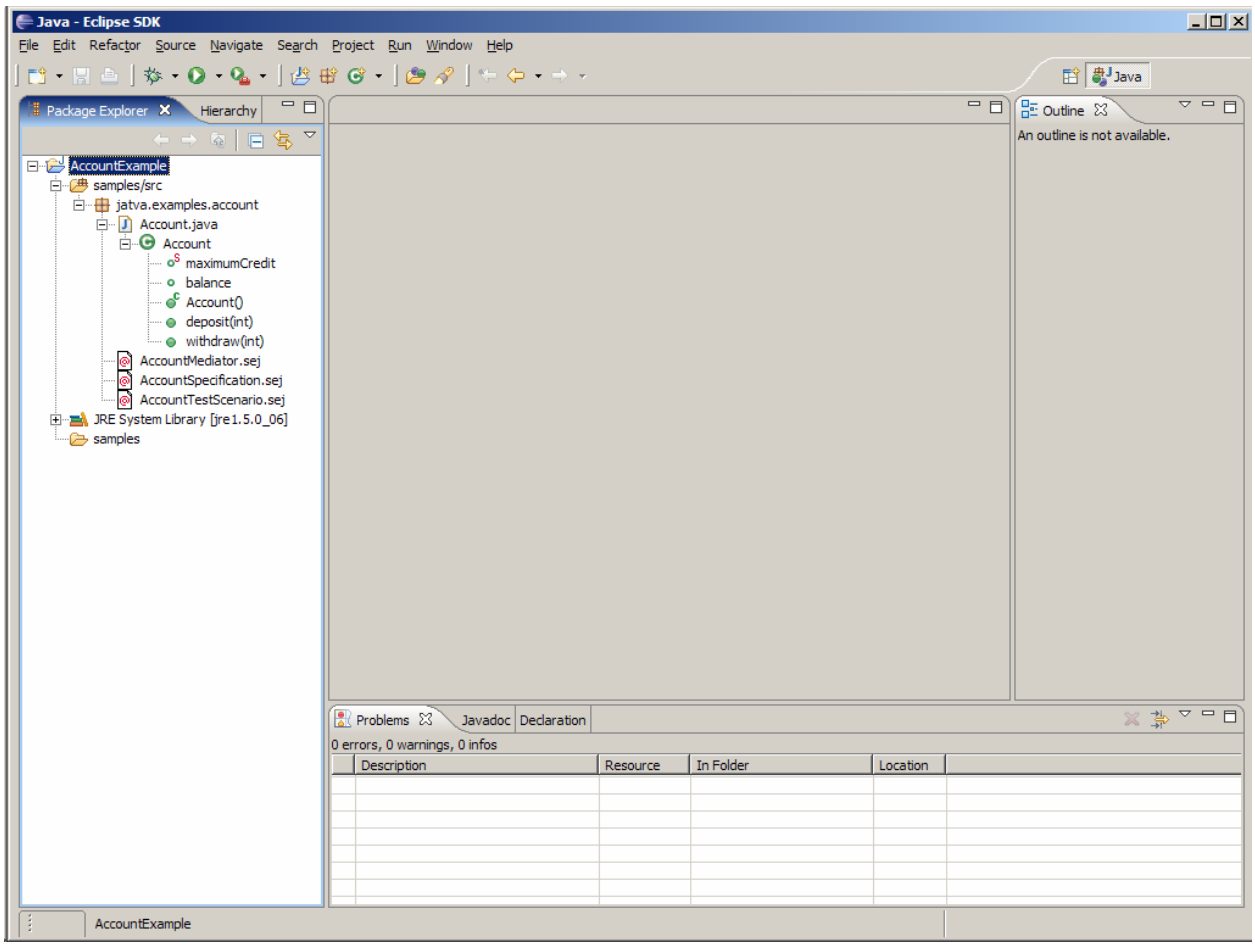

Рисунок 3. Структура каталогов проекта AccountExample

??????Убедитесь также, что структура каталогов C:\JAT\AccountExample\target и C:\JAT\AccountExampe\tests выглядит как показано на следующем рисунке.

Спецификация функциональности системы Банковского кредитного счета

Поддерживаемая JavaTESK технология разработки тестов UniTesK предполагает, что требования к тестируемой системе записаны в четкой однозначно интерпретируемой форме. Такая форма представления требований называется *формальной спецификацией*. Формальные спецификации могут использоваться для автоматической генерации программных компонентов тестовой системы, проверяющих соответствие между требованиям и реальным поведением интерфейсных функций тестируемой системы.

В инструменте JavaTESK формальные спецификации разрабатываются на специальном языке JavaTESK, являющимся расширением языка программирования Java. Язык JavaTESK позволяет описывать *функциональные требования*, которые определяют *функциональность* интерфейсных функций, то есть то, что тестируемая система должна делать при вызовах интерфейсных функций.

Спецификации на языке JavaTESK имеют синтаксис похожий на синтаксис Java. Файлы, содержащие код на JavaTESK имеют расширение **.sej**.

Спецификация системы кредитного счета находится в классе `AccountSpecification`, который расположен в пакете `jatva.examples.account`.

У вас в проекте уже есть этот спецификационный класс, поэтому можно пропустить описание того, как создать новый спецификационный класс, и приступить к разработке медиатора (см. раздел *Разработка медиатора*).

Для того чтобы создать новый спецификационный класс, выделите мышкой в окне **Package Explorer** пакет, в который желаете поместить этот класс. После этого выберите пункт меню **File/New/File**. В появившемся окне **New File** в строке для ввода с надписью **File Name:** введите имя файла, в котором будет находиться код спецификационного класса. Имя файла должно иметь расширение **.sej**. Нажмите на кнопку **Finish**.

После выполнения указанных действий должно появится окно редактора, содержащее код пустого спецификационного класса.

В дальнейшем для разработки теста будет использоваться существующий спецификационный класс `AccountSpecification`.

Можно посмотреть код спецификационного класса `AccountSpecification`. Для этого дважды нажмите левой кнопкой мыши на соответствующем узле.

После выполнения указанных действий должно появиться окно редактора, подобное тому, которое показано на рисунке 4.

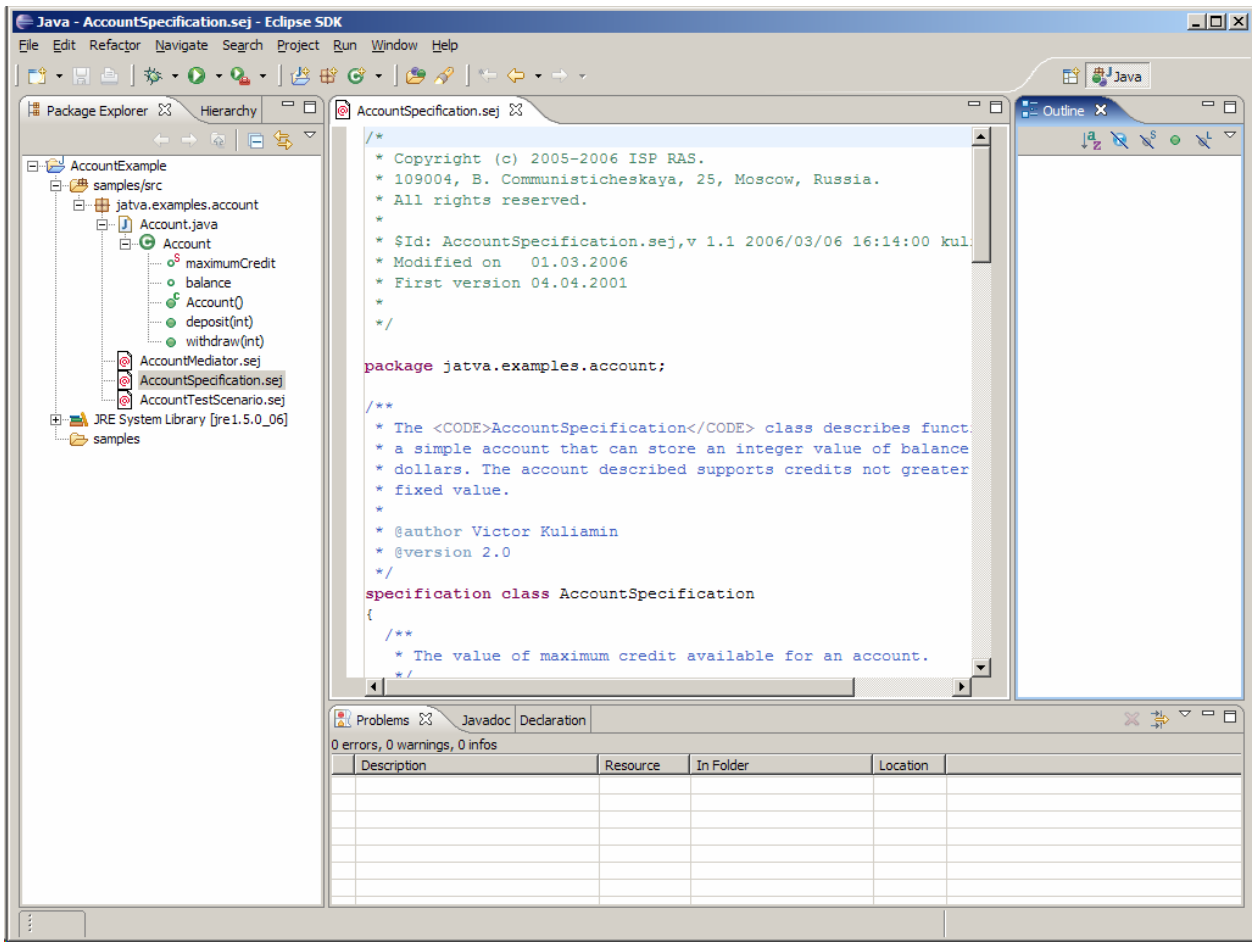

Рисунок 4. Код спецификационного класса AccountSpecification.

Спецификация кредитного счета начинается с указания пакета, которому принадлежит спецификационный класс. В данном случае это `jatva.examples.account`:

```
package jatva.examples.account;
specification class AccountSpecification
{
  static public int maximumCredit;
  public int balance;
  public AccountSpecification() { }
  …
}
```

Также, как и класс `Account`, класс `AccountSpecification` содержит поля `balance` и `maximumCredit`.

На языке JavaTESK 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. Обратите внимание, что в заголовке не надо указывать некоторые модификаторы (**`public`**, **`private`**, **`protected`**, **`abstract`**).

Рассмотрим спецификационный метод `deposit`, описывающий внесение 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 branch";

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

Спецификационный метод состоит из описания поведения целевого метода. Поведение записано в форме *пред-* и *постусловий*.

Предусловие задает область определения целевого метода. Оно оформляется как блок, помеченный ключевым словом **`pre`** и возвращающий результат типа `boolean`, зависящий от значений параметров спецификационного метода и состояния объекта спецификационного класса. Если результат вычисления предусловия равен `true`, можно вызывать целевой метод и ожидать от него корректного поведения, в противном случае поведение целевого метода не определено.

Предусловие должно быть без побочных эффектов. В спецификационной функции может быть только одно предусловие. Предусловие спецификационного метода может быть опущено, что эквивалентно наличию предусловия, всегда возвращающего `true`.

Предусловие спецификационного метода `deposit` показывает, что вносимая сумма `sum` должна быть положительной, а полученный в результате вызова метода баланс `balance` не должен превосходить максимального допустимого значения типа **`int`**.

Целью постусловия является проверка корректности результата вычисления целевого метода (входные и выходные значения параметров и результат вызова целевого метода соответствуют функциональным требованиям). Для этого можно использовать значения параметров спецификационного метода, состояние объекта спецификационного класса до и после вызова целевого метода. Постусловие оформляется как блок, помеченный ключевым словом **`post`** и возвращающий результат типа `boolean`. Если его результат равен `true`, поведение целевого метода корректно, в противном случае – нет.

Постусловие спецификационного метода `deposit` показывает, что поведение целевого метода `deposit` корректно, если баланс счета после вызова метода (`balance`) равен балансу счета до вызова метода (**`pre`** `balance`), увеличенному на сумму `sum`. Обратите внимание, если перед именем поля написать ключевое слово **`pre`**, то можно получить значение этого поля до вызова целевого метода.

Операторы **branch** и **mark** используются для определения *тестового покрытия*, они выделяют интересные для тестирования ситуации. Смысл этих конструкций будет дан при рассмотрении тестовых сценариев для системы банковского кредитного счета.

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 "Withdrawn sum is too large";
      return balance == pre balance && withdraw == 0;
    }
    else
    {
      branch "Successful withdrawal";
      return balance == pre balance - sum && withdraw == sum;
    }
  }
}
```

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

Второй **if** описывает два случая: когда снятие указанной суммы невозможно (ветвь `then`) и когда снятие указанной суммы возможно (ветвь `else`). В первом случае постусловие показывает, что баланс не должен измениться (`balance == pre balance`) и метод должен возвратить **0** (`withdraw == 0`), во втором – баланс должен уменьшиться на сумму `sum` и метод должен возвратить эту же сумму. Заметьте, что для обозначения возвращаемого методом `withdraw` результата используется одноименный идентификатор.

Язык JavaTESK предоставляет также 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 не указывается, поскольку он всегда boolean. Список аргументов у инварианта пустой. Также для инвариантов разрешены не все модификаторы `Java`.

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

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

Инвариант этого типа возвращает **`true`**, если значение поля `balance` проверяемой структуры соответствует требованию, и **`false`** в противном случае.

Инвариант вызывается до и после вызова любой спецификационной функции.

Медиаторы для системы Банковского кредитного счета

Чтобы обеспечить возможность проверки соответствия между реализацией и спецификацией тестируемой системы, они должны быть связаны некоторым образом. В UniTesK для этого используются специальные компоненты — *медиаторы*. В JavaTESK медиаторы реализуются *медиаторными методами* — специального вида методами, помеченными ключевым словом **`mediator`**.

В проекте `AccountExample` уже есть готовый медиаторный класс `AccountMediator`, расположенный в пакете `ru.ispras.redverst.se.java.examples.account.model`, поэтому можно пропустить описание того, как создать новый медиаторный класс, и приступить к разработке тестового сценария (см. раздел *Разработка тестового сценария*).

Для того чтобы создать новый спецификационный класс, выделите мышкой в окне **Package Explorer** пакет, в который желаете поместить этот класс. После этого выберите пункт меню **File/New/File**. В появившемся окне **New File** в строке для ввода с надписью **File Name:** введите имя файла, в котором будет находиться код медиаторного класса. Имя файла должно иметь расширение **.sej**. Нажмите на кнопку **Finish**.

.....

После выполнения указанных действий должно появится окно редактора, содержащее код пустого медиаторного класса.

В дальнейшем для разработки теста будет использоваться существующий медиаторный класс `AccountMediator`.

Можно посмотреть код медиаторного класса `AccountMediator`. Для этого дважды нажмите левой кнопкой мыши на соответствующем узле.

Код медиаторного класса начинается с указания пакета, которому этот класс принадлежит. В данном случае это `jatva.examples.account`:

```
package jatva.examples.account;
mediator class AccountMediator implements AccountSpecifcation
{
    implementation Account targetObject = null;
    …
}
```

Медиаторный класс будет связывать некие спецификационный класс и реализационный класс. Спецификационный класс необходимо указать в заголовке медиаторного класса: после имени класса необходимо написать ключевое слово **`implements`**, а за ним – имя спецификационного класса. Теперь укажем реализационный класс. Для этого в теле медиаторного класса объявим поле с типом – реализационным классом – и модификатором **`implementation`**. Такое поле будем называть *реализационным полем*. Оно

будет содержать объект реализационного класса, над которым будут проводиться тестовые испытания.

Кроме того, тело медиаторного класса будет состоять из медиаторных методов и некоторых специальных блоков.

Рассмотрим медиаторный метод `deposit`, описывающий связь спецификационного метода `deposit` и реализационного метода `deposit`. Сигнатура медиаторного метода должна совпадать с сигнатурой соответствующего спецификационного метода, за исключением одного момента – ключевое слово **`specification`** заменяется на **`mediator`**:

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

Медиаторный метод состоит из различных блоков, описывающих связи состояний объекта реализационного класса и объекта спецификационного класса.

В теле медиаторного метода обязательно должен быть расположен *реализационный блок*. Он оформляется как блок, помеченный ключевым словом **`implementation`**. Реализационный блок описывает поведение медиатора в тот момент, когда необходимо получить реакцию соответствующего реализационного метода. В данном случае достаточно вызвать реализационный метод `deposit`. Он вызывается через реализационное поле (в данном случае оно называется `targetObject`).

Медаторный метод withdraw, описывающий связь спецификационного метода withdraw и реализационного метода withdraw, аналогичен медиаторному методу deposit. Также состоит из реализационного блока, в котором просто вызывается реализационный метод:

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

Кроме медиаторных методов, медиаторный класс может содержать *update-блок*. Он оформляется как блок, помеченный ключевым слово **`update`**. **`update`**-блоки также могут быть в медиаторных методах. Его цель – сделать состояние спецификационного объекта соответствующим измененному состоянию[1] реализационного метода. Соответственно

[1] *Состоянием* объекта (в широком смысле) называется такое его описание, которое однозначно соответствует поведению объекта. Для объектов (в смысле объектно-ориентированных языков программировния) состояние можно задавать значениями полей этого объекта (хотя такое описание не единственное).

update-блок вызывается после реализационного блока (когда состояние реализационного объекта может измениться). Сначала вызываются **update**-блоки классов, а потом – **update**-блок медиаторного метода, который в данный момент исполняется.

В классе `AccountMediator` медиаторные методы не содержат **update**-блоков. Однако сам класс содержит **update**-блок. Он будет вызываться каждый раз после вызова реализационного блока в каждом медиаторном методе класса `AccountMediator`:

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

Тестовый сценарий для системы Банковского кредитного счета

Настало время определиться с тем, как собственно будет проводиться тестирование. А проводиться оно будет по следующей схеме: на систему будут оказываться *тестовые воздействия*. Это значит, что сначала будет получена реакция спецификационного метода на тестовое воздействие, потом через медиатор будет получена реакция реализационного метода на то же самое тестовое воздействие и, наконец, эти две реакции будут сравнены согласно постусловию. Тестовое воздействие строится на основе *тестовых данных* (аргументов для спецификационных и реализационных методов). Тестовые данные необходимо будет придумать программисту, а вот само тестирование (когда написаны спецификации и медиаторы) проводится автоматически на основе *тестовых сценариев*. Специальный компонент инструмента JavaTESK, называемый *обходчиком*, на основе тестовых сценариев генерирует тестовые воздействия. По результатам тестовых воздействий на систему определяется достигнутое покрытие. Покрытие описывает качество проведенного тестирования согласно *критерию покрытия*. Критерий отражается в спецификациях при помощи специальных операторов. Примером покрытия может быть набор помеченных конструкций в реализационном методе. Если после всех тестовых воздействий все конструкции исполнились, то покрытие считается 100%-ным.

Поставим перед собой цель – достичь 100% покрытия *ветвей функциональности*[2] спецификационного класса `AccountSpecification`. Это означает, что мы должны разработать и выполнить набор тестов, покрывающих все ветви функциональности, определенные в постусловиях спецификационных методов с помощью операторов **`branch`**. В спецификационных методах на каждом пути в постусловии должен быть единственный оператор **`branch`**. Для класса `AccountSpecification` получились следующие ветви (их имена идут сразу же после ключевого слова **`branch`**) :

`Single` (определена в спецификационном методе `deposit`)

`TooLargeSum` (определена в спецификационном методе `withdraw`)

---

[2] под *ветвью функциональности* следует понимать выполнение программы с предопределенными значениями условий в конструкциях ветвления (**`if`**, **`while`**, **`for`**, **`switch`**)

`Normal` (определена в спецификационном методе `withdraw`)

Тестовые сценарии оформляются в виде специальных классов, называемых *сценарными*.

В проекте `AccountExample` уже есть готовый сценарный класс `AccountTestScenario`, расположенный в пакете `jatva.examples.account`, поэтому можно пропустить описание того, как создать новый сценарный класс, и приступить к выполнению теста и анализу результатов (см. раздел *Выполнение теста и анализ результатов*).

Для того чтобы создать новый сценарный класс, выделите мышкой в окне **Package Explorer** пакет, в который желаете поместить этот класс. После этого выберите пункт меню **File/New/File**. В появившемся окне **New File** в строке для ввода с надписью **File Name:** введите имя файла, в котором будет находиться код сценарного класса. Имя файла должно иметь расширение **.sej**. Нажмите на кнопку **Finish**.

.....

После выполнения указанных действий должно появится окно редактора, содержащее код пустого сценарного класса.

В дальнейшем для разработки теста будет использоваться существующий сценарный класс `AccountTestScenario`.

Можно посмотреть код сценарного класса `AccountTestScenario`. Для этого дважды нажмите левой кнопкой мыши на соответствующем узле.

Код сценарного класса начинается с указания пакета, которому этот класс принадлежит. В данном случае это `jatva.examples.account`:

```
package jatva.examples.account;
scenario class AccountTestScenario
{
    protected AccountSpecification objectUnderTest;
    ...
}
```

Сценарный класс должен содержать объект спецификационного класса. Над ним будут совершаться все тестовые воздействия. Обычно такое поле называют `objectUnderTest`, хотя его можно назвать и иначе.

Далее. Как и любой класс, сценарный класс должен содержать конструктор. В нём как раз будет инициализирован `objectUnderTest` и выбран обходчик:

```
  public AccountTestScenario()
  {
    objectUnderTest = mediator AccountMediator(targetObject = new
Account());
    objectUnderTest.attachOracle();
    setTestEngine( new jatva.engines.DFSMExplorer() );
  }
```

Первый оператор создает новый объект типа `AccountMediator`. Для этого используется конструкция **`mediator`**, похожая на привычную конструкцию **`new`**. Единственным отличием от **`new`** является то, что порядок аргументов в конструкторе `AccountMediator` может быть произвольным. Поэтому в скобках после `AccountMediator` надо указывать выражения вида:

`название_аргумента = значение_аргумента`

То, что `objectUnderTest` является объектом спецификационного класса, а проинициализировали мы его объектом медиаторного класса, не является ошибкой. Ведь медиаторный класс наследует спецификационный.

Вспомним, что медиаторному классу, кроме объекта спецификационного класса, нужен объект реализационного класса. Поэтому в качестве одного из аргументов при создании `objectUnderTest` надо указать этот объект. Для этого в качестве *`названия_аргумента`* надо написать имя реализационного поля, как оно объявлено в медиаторном классе `AccountMediator`, а в качестве *`значения_аргумента`* – конструкцию создания объекта. В результате получается:

```
targetObject = new Account()
```

Следующий оператор фиксирует, что для объекта `objectUnderTest` надо вызывать проверку инвариантов, пред- и постусловий (вызывать оракул). Если этого не сделать, то будет вызываться только проверка инвариантов.

Третий оператор фиксирует обходчик (TestEngine), который будет использоваться сценарным классом. В данном случае выбирается `jatva.engines.DFSMExplorer`. Далее в программе мы будем обращаться к обходчику только через оператор **`execute`**.

Настало время описать сценарные методы. Они служат для перевода обходчиком объекта `objectUnderTest` из одного состояния в другое. В основе перевода (*перехода*) – вызов (реализационного) метода, соответствующего сценарному, из текущего состояния. Состояние объекта `objectUnderTest` после работы вызванного метода – и есть то состояние, в которое осуществляется переход. В результате обходчик составляет граф, вершинами которого будут состояния объекта `objectUnderTest`, а метками дуг – сценарные методы.

В спецификационном методе определены 2 спецификационных метода: `deposit` и `withdraw`. Для каждого из них составим свой сценарный метод. Сигнатура каждого из них состоит из ключевого слова scenario и имени сценарного метода:

```
scenario deposit()
{
    if (objectUnderTest.balance < 10)
          execute objectUnderTest.deposit( 1 );
    return true;
}
```

Основной оператор сценарного метода – оператор **`execute`**. При его исполнении происходит вызов медиаторного метода (а он в свою очередь реализационного) и, если необходимо, проверка результата работы медиаторного метода (дополнительная к постусловию). Оператор состоит из ключевого слова **`execute`** и вызова медиаторного метода (с некоторыми аргументами, которые программист должен сам подобрать такими, на которых покрывались бы нужные операторы **`branch`** соответствующего спецификационного метода). Поскольку в спецификационном методе `deposit` написан всего один оператор **`branch`**, то достаточно одного вызова `objectUnderTest.deposit` . В качестве параметра выберем 1, поскольку на этом значении параметра достигается оператор **`branch`** `Simple` в спецификационном методе `deposit`.

Оператор execute поместим в условный оператор

```
if (objectUnderTest.balance < 10)
```

Это ограничит размер графа, который строит обходчик (при достижении `objectUnderTest.balance` значения 10 не будет вызываться `objectUnderTest.deposit` и не будет строиться новое состояние – новая вершина графа).

Последний оператор сценарного метода deposit возвращает true, поскольку состояние объекта `objectUnderTest` после выполнения execute не нуждается в дополнительной проверке.

Следующий сценарный метод – метод `withdraw`:

```
scenario withdraw()
{
    iterate( int i = 1; i < 6; i++; )
          execute objectUnderTest.withdraw( i );
    return true;
}
```

Новым, по сравнению со сценарным методом `deposit`, здесь является оператор **`iterate`**. Он используется для задания последовательности воздействий. Оператор **`iterate`**`(;;;)` имеет синтаксис похожий на синтаксис оператора языка Java **`for(;;)`** за исключением последнего выражения, которое задает условие фильтрации воздействий. В теле оператора **`iterate`** задаются воздействия, которые должны быть произведены в каждом достигнутом состоянии сценария. В данном сценарном методе `withdraw` не будем определять фильтрацию воздействий.

Ещё одно отличие оператора **`iterate`** от оператора **`for`** заключается в том, что в момент его выполнения тело этого оператора исполняется только один раз, а при следующем входе выполнение **`iterate`** будем продолжено со следующей итерации. В данном случае на каждой итерации вызывается (медиаторный) метод `withdraw`. Ограничение на i подобрано таким образом, чтобы покрыть все операторы **`branch`** и **`mark`** в спецификационном классе `AccountSpecification`.

Дополнительная, по сравнению с постусловием, проверка результата работы медиаторного метода `withdraw` не требуется – сценарный метод возвращает `true`.

Вспомним, что обходчик составляет граф, вершины которого – состояния объекта objectUnderTest. Причем составлять этот граф он будет до тех пор, пока с помощью сценарных методов удается построить новое состояние. Однако смысла повторять состояния, в которых поведение объекта `objectUnderTest` одинаковое, нет. Предлагается строить граф только из *обобщенных состояний*. Одному обобщенному состоянию соответствует несколько состояний объекта `objectUnderTest` с одинаковым поведением. Соответственно, по полученному сценарным методом (конкретному) состоянию объекта `objectUnderTest` надо построить обобщенное состояние и добавить в граф, если такого обобщенного состояния ещё не было. Для построения обобщенного состояния по (конкретному) состоянию в сценарном классе надо объявить *state-блок*. Он оформляется как блок, помеченный ключевым словом **`state`**:

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

В данном случае в качестве обобщенного состояния выбран объект класса `Integer`, представляющий собой баланс кредитного счёта.

Сценарный класс готов. Осталось написать метод `main`. Можно поместить его прямо в сценарный класс. Так и сделаем. В этом методе надо создать объект сценарного класса и вызвать у него метод `run`:

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

Выполнение теста для системы Банковского кредитного счета

Для запуска тестового сценария первым делом необходимо подключить особенности JavaTESK к проекту `AccountExample`. Для этого в окне **Package Explorer** в контекстном меню узла с надписью **AccountExample** (этот узел является родительским для всего проекта `AccountExample`) выберите пункт **Enable JavaTESK features**. При этом окно **Package Explorer** станет таким, каким оно показано на рисунке 5. А именно в проект `AccountExample` добавились узлы **javatesk-generated и jatt-tsbasis-2.0.SNAPSHOT.jar**.

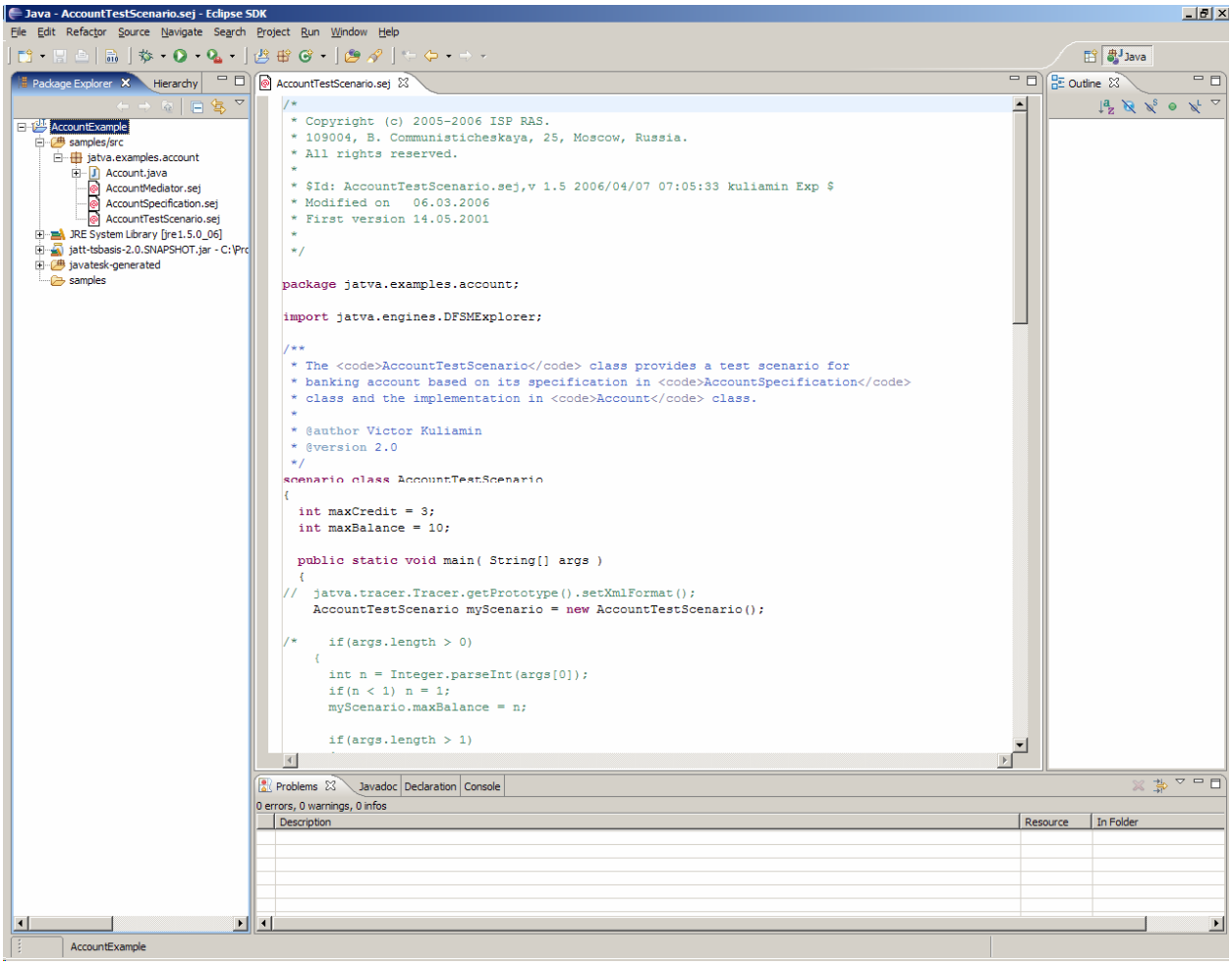

Рисунок 5. Окно среды Eclipse после Enable JavaTESK features.

После этого необходимо транслировать проект `AccountExample` полностью в Java. Для этого выберите в окне **Package Explorer** пакет **AccountExample**. Затем выберите пункт меню **Project/BuildAutomatically**. При этом слева от него должна появиться галочка. Если она уже есть, то нажимать на **BuildAutomatically** не нужно (галочка исчезнет; чтобы она появилась вновь, необходимо снова нажать на **BuildAutomatically**). Затем выберите пункт меню **Project/Clean…**. Появится окно, изображенное на рисунке 6.

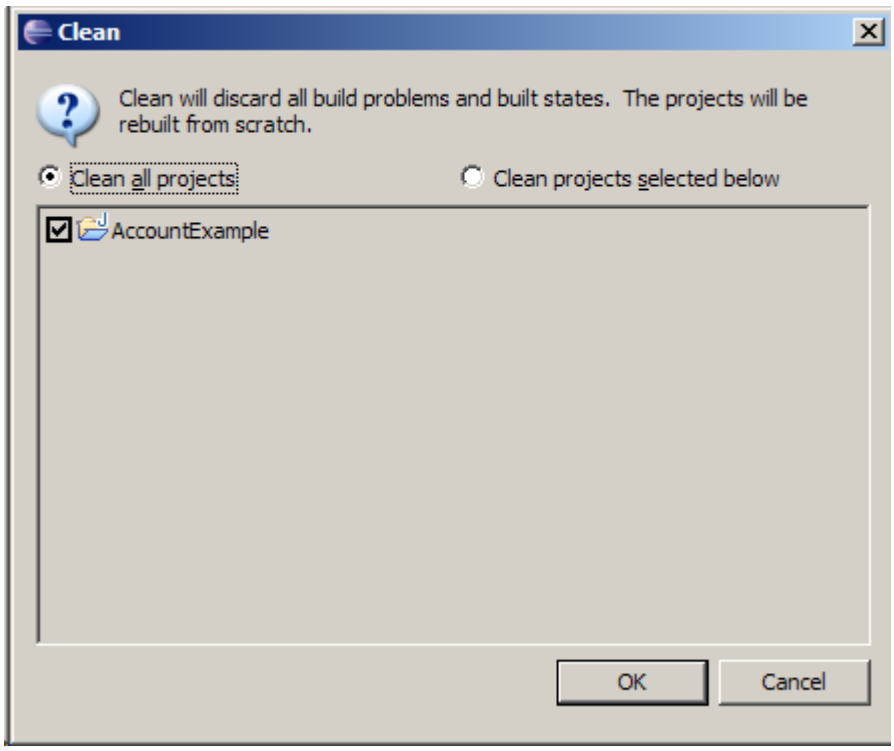

Рисунок 6. Сборка проекта.

Выберите пункт **Clean projects selected below**. При этом пространство окна между этим пунктом и кнопками станет доступным для изменения. Поставьте галочку у пункта **AccountExample** и снимите все остальные пункты. Нажмите кнопку **ОК**.

Если окно **Package Explorer**'а после этого не изменилось, значит, исходный текст ошибок не содержал. Однако при наличии ошибок окно может стать, например, таким, каким оно изображено на рисунке 7. Следует исправить все ошибки и пересобрать проект `AccountExample` так, как описано чуть выше.

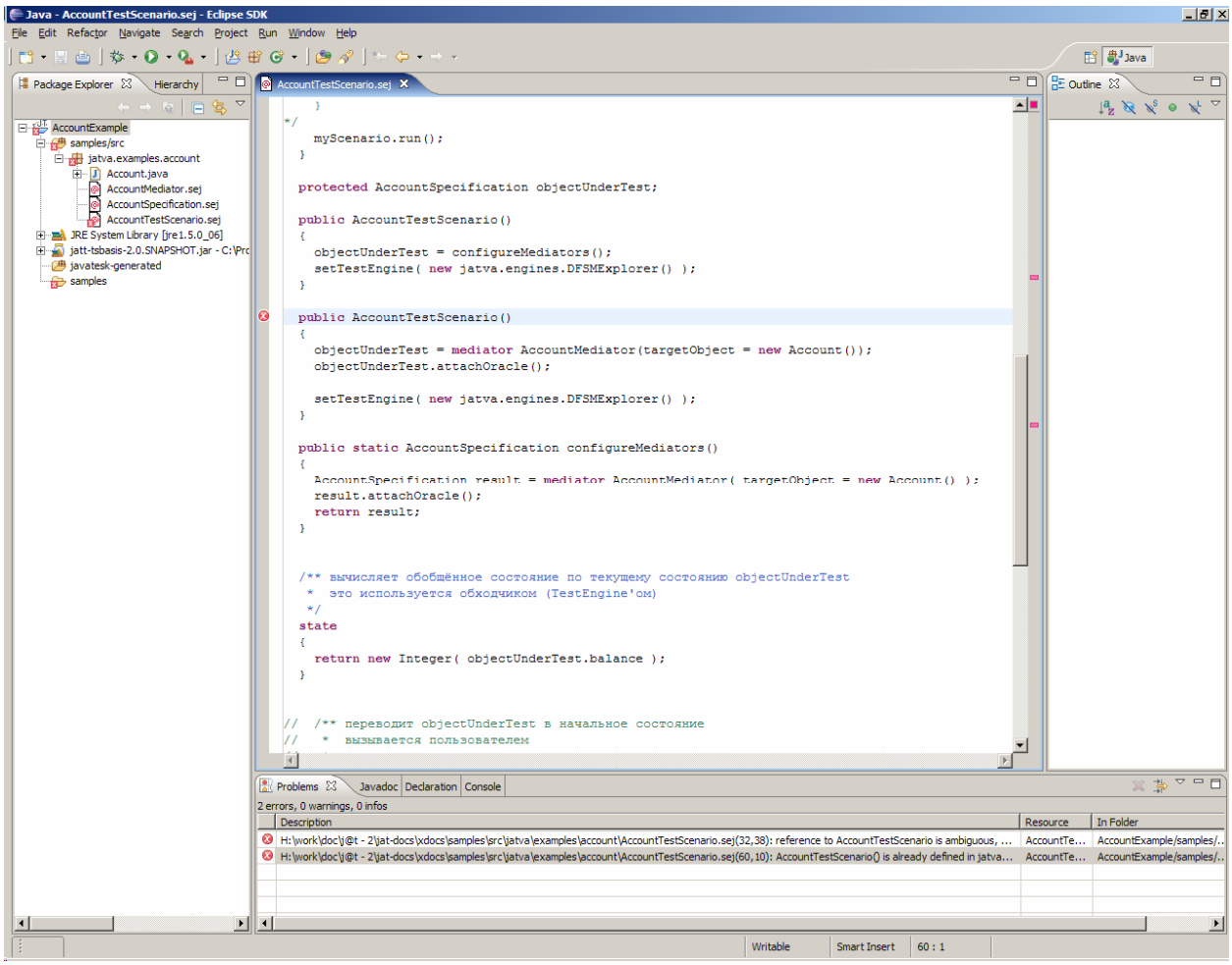

Рисунок 7. Окно среды Eclipse с выделенной ошибкой в коде.

Далее необходимо создать новую конфигурацию запуска. Для этого выберите меню **Run/Run…**. Появится окно, изображенное на рисунке 8.

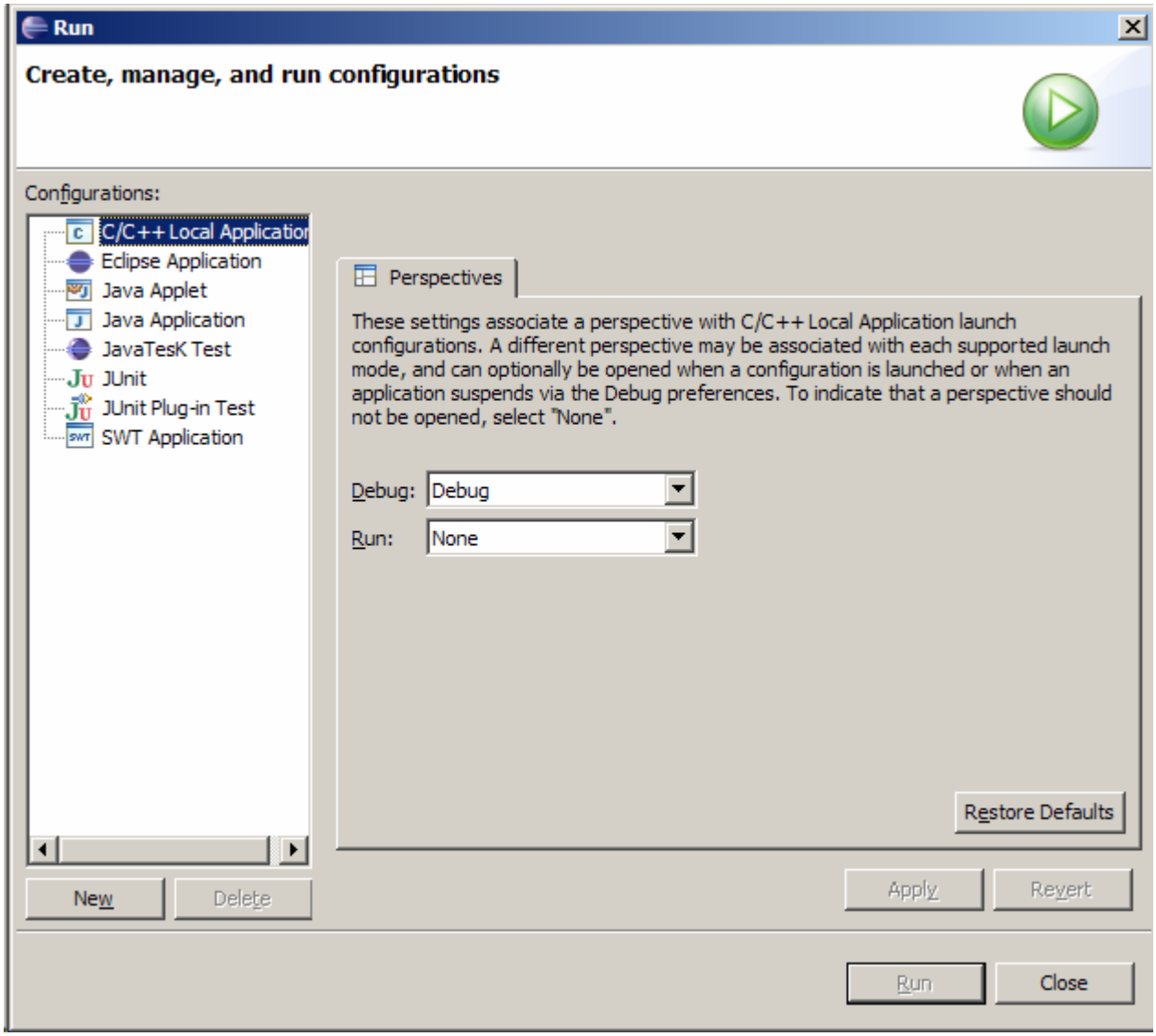

Рисунок 8. Создание новой конфигурации запуска.

В списке **Configurations:** выберите **JavaTesK Test** и нажмите кнопку **New**. Окно конфигурации запуска должно стать таким, каким оно изображено на рисунке 9.

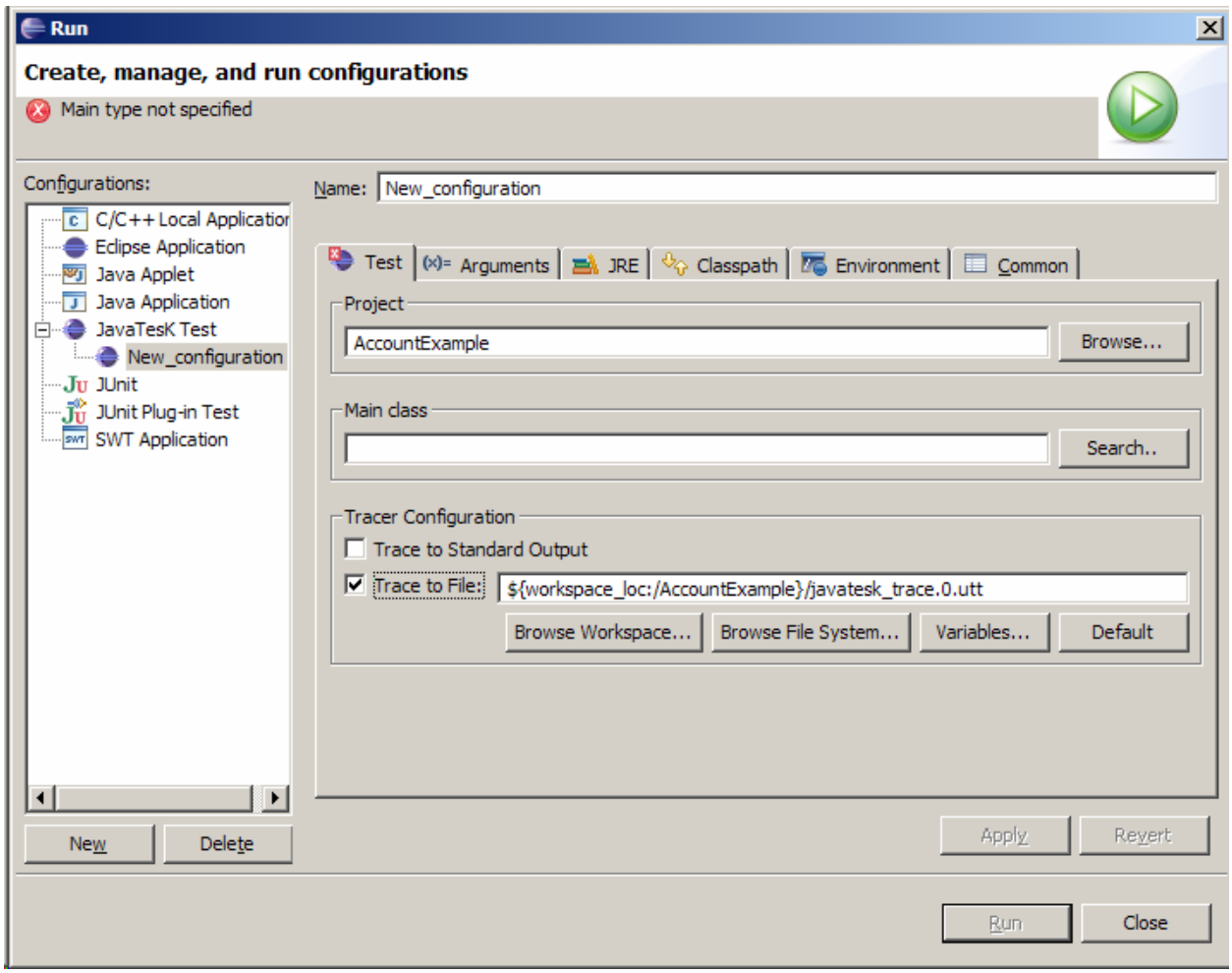

Рисунок 9. Создание новой конфигурации запуска JavaTESK.

В поле **Name:** можно написать некоторый текст, отличающий эти запуски тестового сценария. Можно оставить это поле таким, каково оно по умолчанию.

В поле **Main class** надо написать имя того класса, где расположен метод main. Для этого проекта в это поле следует написать `jatva.examples.account.AccountTestScenario`, поскольку именно в этот класс был помещен метод `main`.

Нажмите кнопку **Apply**. Нажмите кнопку **Run**.

Теперь в окне **Package Explorer** в контекстном меню узла **AccountExample** выберите **Refresh**. После этого окно среды должно быть таким, как на рисунке 10. Должны появиться или обновиться папки **jatva\examples\account**. Они содержат трассы, сгенерированные во время запуска теста. Каждой трассе будет соответствующий узел со значком, в виде бегущего человечка. Если на него нажать, откроется окно **UniTesK Trace Viewer**, где можно увидеть конечный автомат, построенный обходчиком. Пример такого автомата можно посмотреть на рисунке 11.

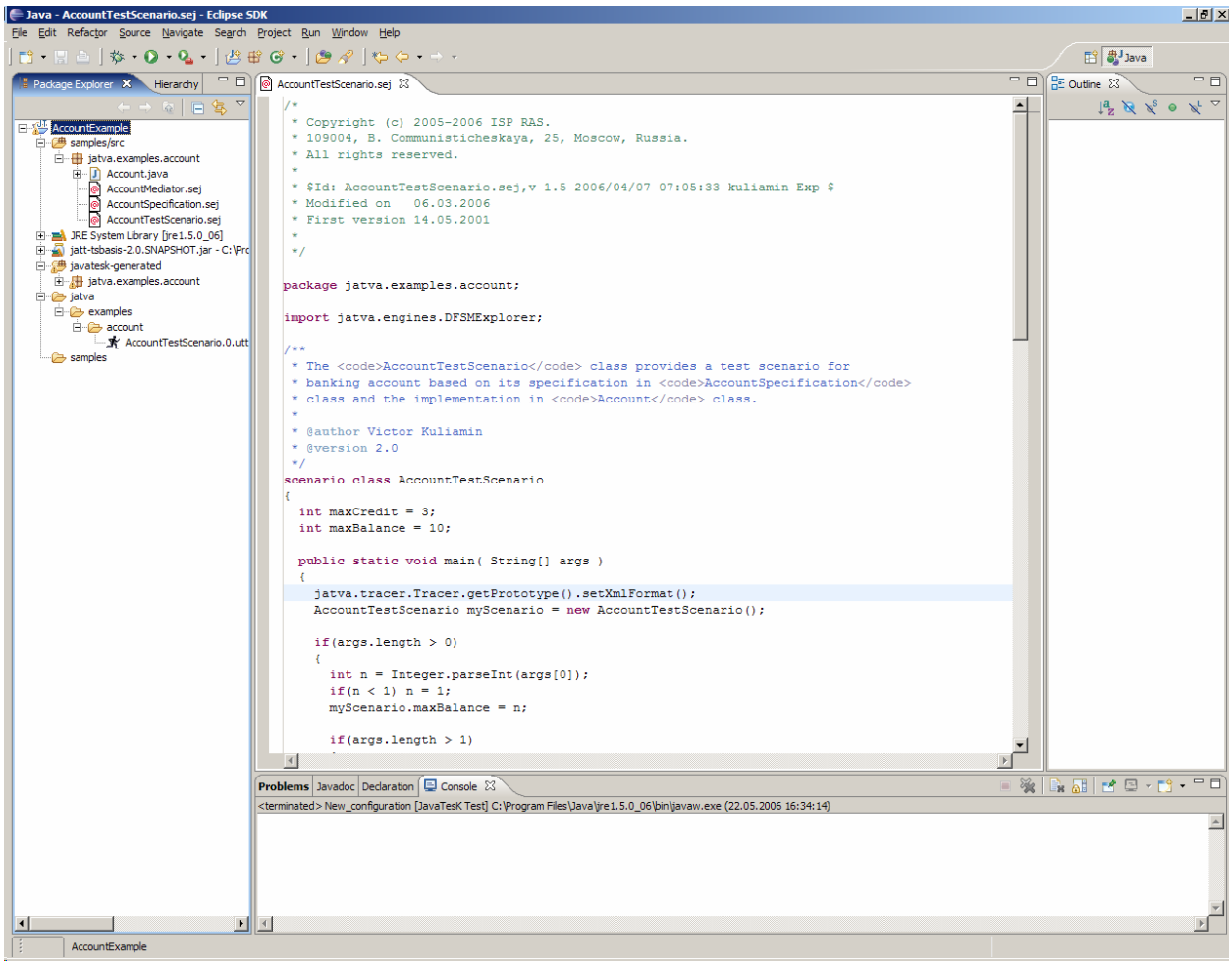

Рисунок 10. Окно среды Eclipse после исполнения теста.

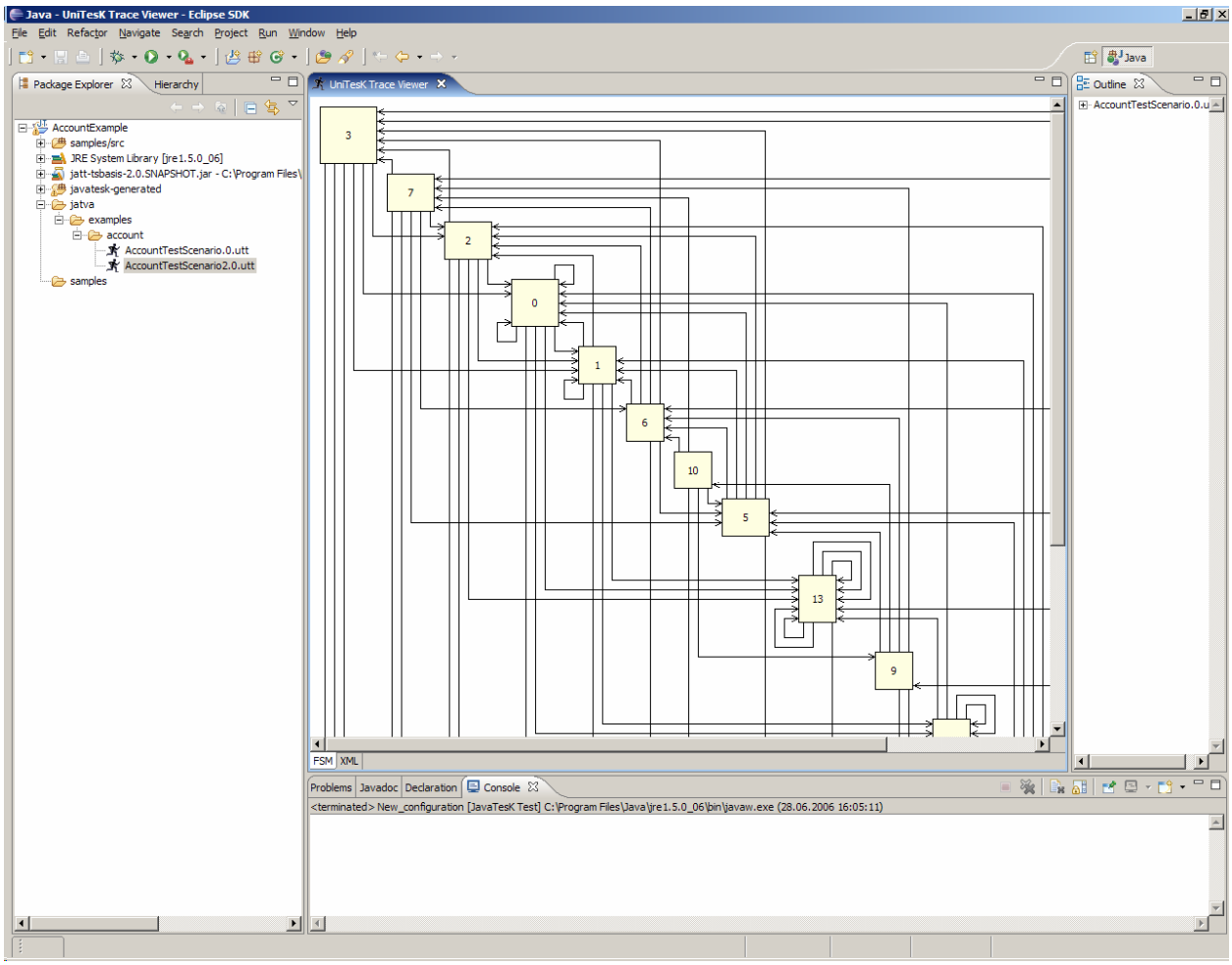

Рисунок 11. Окно UniTesK Trace Viewer с автоматом, построенным обходчиком

Анализ результатов выполнения теста для системы Банковского кредитного счета

Генерация тестовых отчетов

Предполагается, что выполнены все предыдущие пункты данного документа. Тогда для генерации тестового отчета необходимо сделать следующее. В окне **Package Explorer** выделите пункте, который соответствует трассе тестового запуска, для которого необходимо сгенерировать отчет. Выберите в его контекстном меню пункт Generate Report. При этом появится окно, изображенное на рисунке 12.

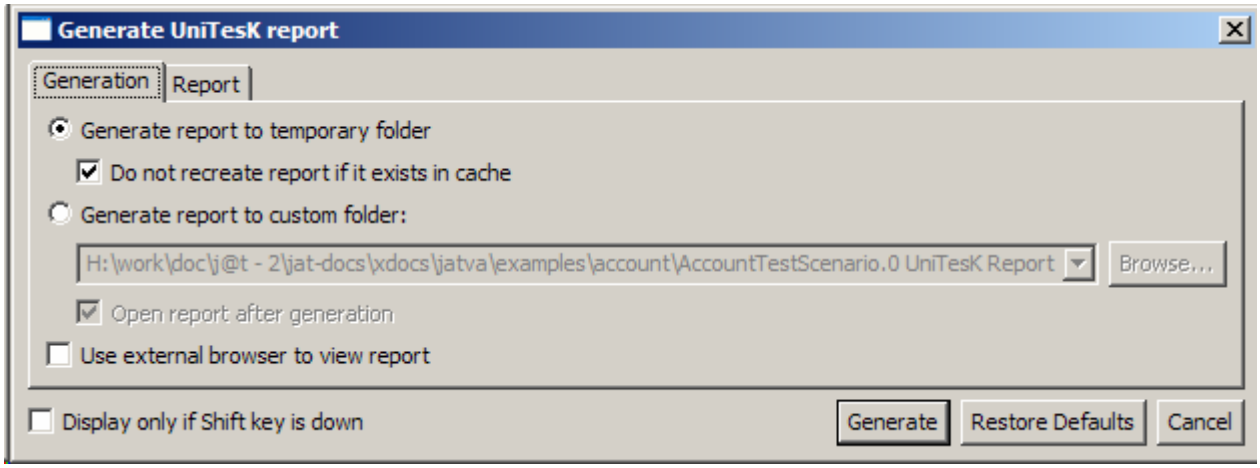

Рисунок 12. Окно генерации тестового отчета.

Нажмите кнопку Generate.

При этом откроется новая вкладка в основном окне с надписью UniTesK Report, как показано на рисунке 13. Будем далее эту вкладку называть *браузером тестового отчета*.

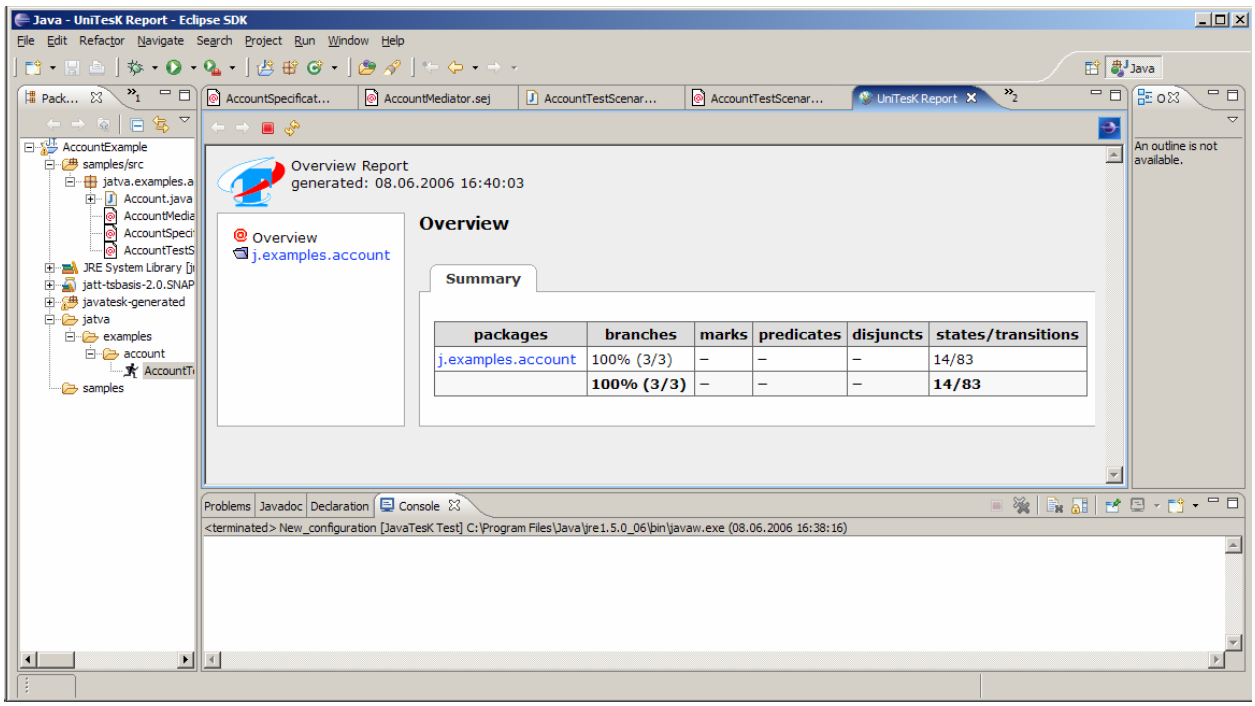

Рисунок 13. Окно отчёта Overview Report во встроенном браузере Eclipse.

Если в окошке Generate UniTesK report поставить галочку Use external browser to view report, то после нажатия кнопки Generate откроется окно внешнего по отношению к Eclipse браузера (здесь и далее это будет Internet Explorer), изображенное на рисунке 14.

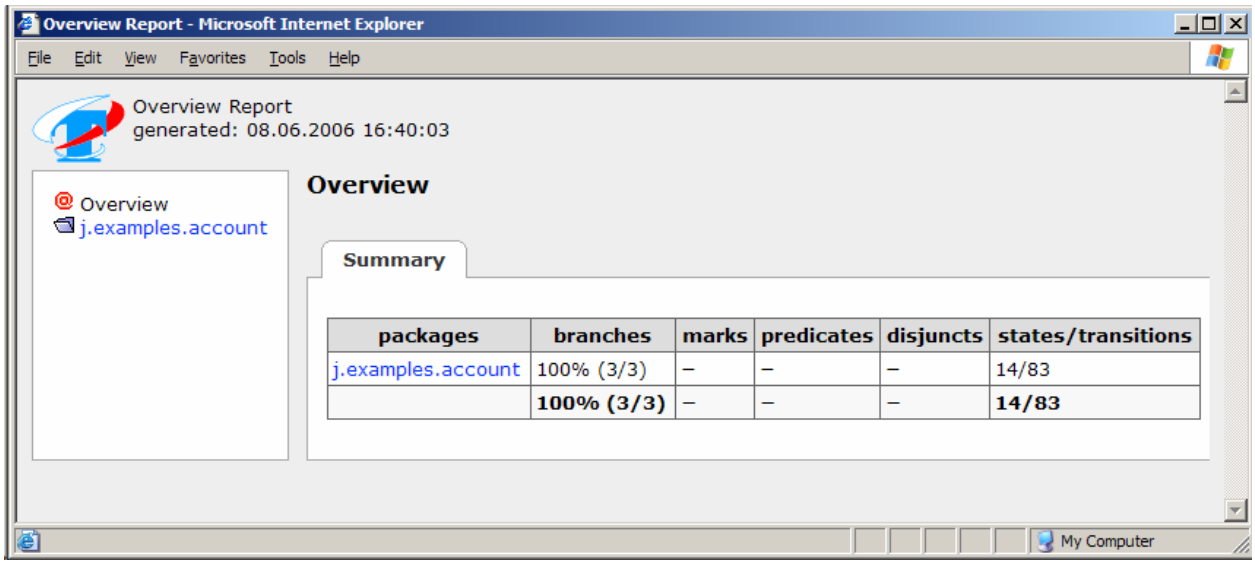

Рисунок 14. Окно отчета Overview Report в браузере Internet Explorer.

Информация на странице **UniTesK Report** представлена в виде таблицы со следующими столбцами:

`packages` – пакеты, в которых расположены спецификационные и/или сценарные классы

`failures` – количество обнаруженных ошибок

`branches` – уровень покрытия *ветвей функциональности*

`marks` – уровень покрытия *помеченных путей*

`predicates` – уровень покрытия *предикатов*

`disjuncts` – уровень покрытия *дизъюнктов*

`states/transitions` – общее число обобщенных состояний и переходов, реализованных в процессе тестирования

В нижней строке таблицы представлена суммарная информация по всем пакетам.

Для получения информации о покрытии спецификационных методов спецификационного класса `AccountSpecification` выберите в списке слева пакет `java.examples.account`, затем – спецификационный класс `AccountSpecification`.

В результате браузер откроет страницу **Specification Summary Report**, показаную на рисунке 15.

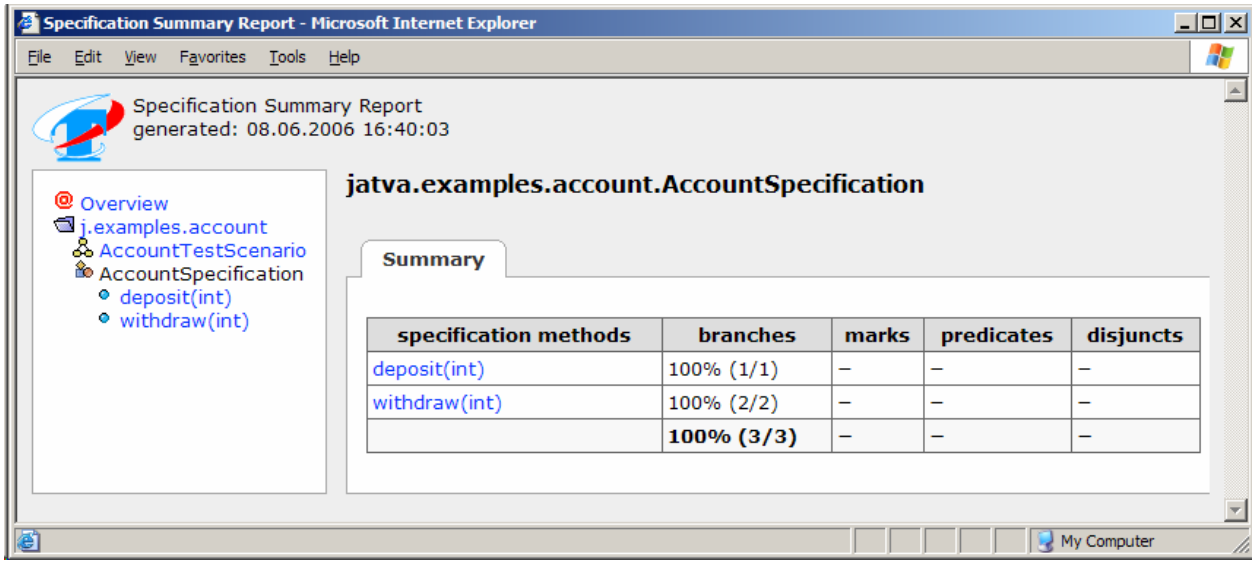

Рисунок 15. Specification Summary Report.

Информация на странице **Specification Summary Report** представлена в виде таблицы, в которой, по сравнению с предыдущей, отсутствует столбец **`states/transitions`**, а столбец **`packages`** заменен на **`specification methods`**, в котором перечислены спецификационные методы класса `AccountSpecification`.

В нижней строке таблицы представлена суммарная информация по спецификационному классу.

Заметим, что покрыты все ветви функциональности спецификационного класса `AccountSpecification`, потому что под столбцом **`branches`** написано **100%**. То есть мы достигли цели, которую поставили перед разработкой тестового сценария.

Для того, чтобы посмотреть покрытие спецификационного метода `deposit`, нажмите на его название в столбце **`specification methods`**. В результате браузер откроет страницу **Method Coverage Report**, показаную на рисунке 16.

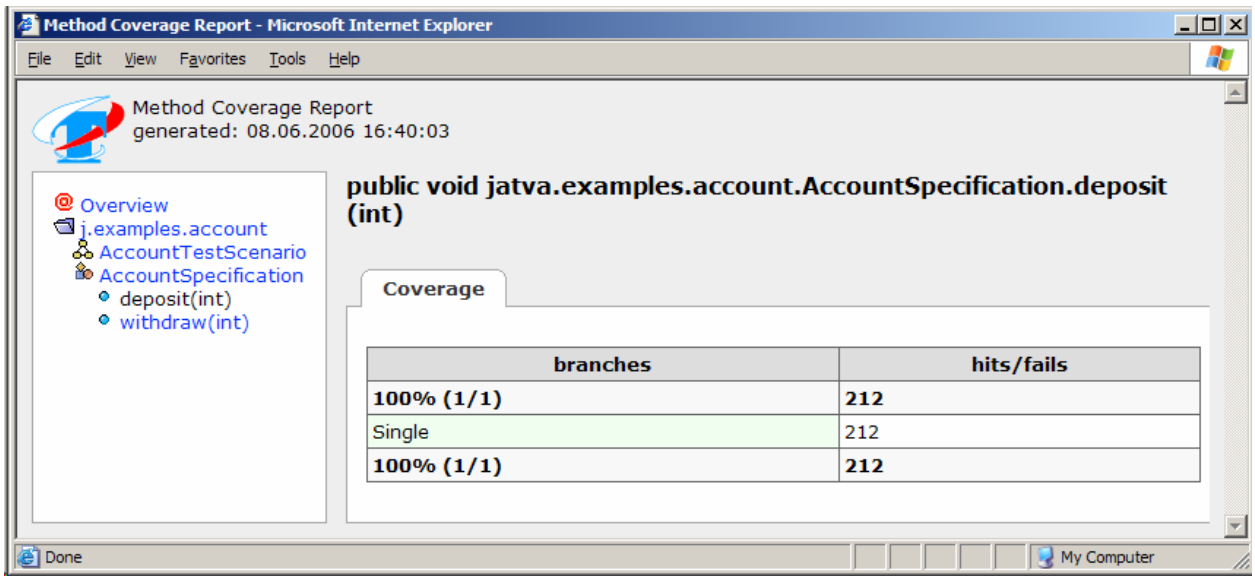

Рисунок 16. Method Coverage Report.

Основная информация на странице **Method Coverage Report** представлена в виде таблицы, содержащей тестовые ситуации. Для каждой из них указывается количество обращений к методу, при которых данная ситуация реализовалась, и количество обнаруженных при этом ошибок. Покрытые ситуации выделены зеленым цветом.

Из отчета о покрытии функции `jatva.examples.account.AccountSpecification.deposit` видно, что было сделано всего 212 вызовов функции, причем из них **212** были сделаны со значениями параметров, соответствующими ветви **Single**.

После таблицы представлены формулы предикатов (список **`predicates`**) и элементарных логических формул (список **`prime formulas`**).

Для получения информации об обобщенных состояниях и переходах, реализованных при выполнении тестового сценария `AccountTestScenario`, выберите сценарный класс `AccountTestScenario` в меню слева. В результате браузер откроет страницу **Scenario Certain Transitions Report**, показаную на рисунке 17.

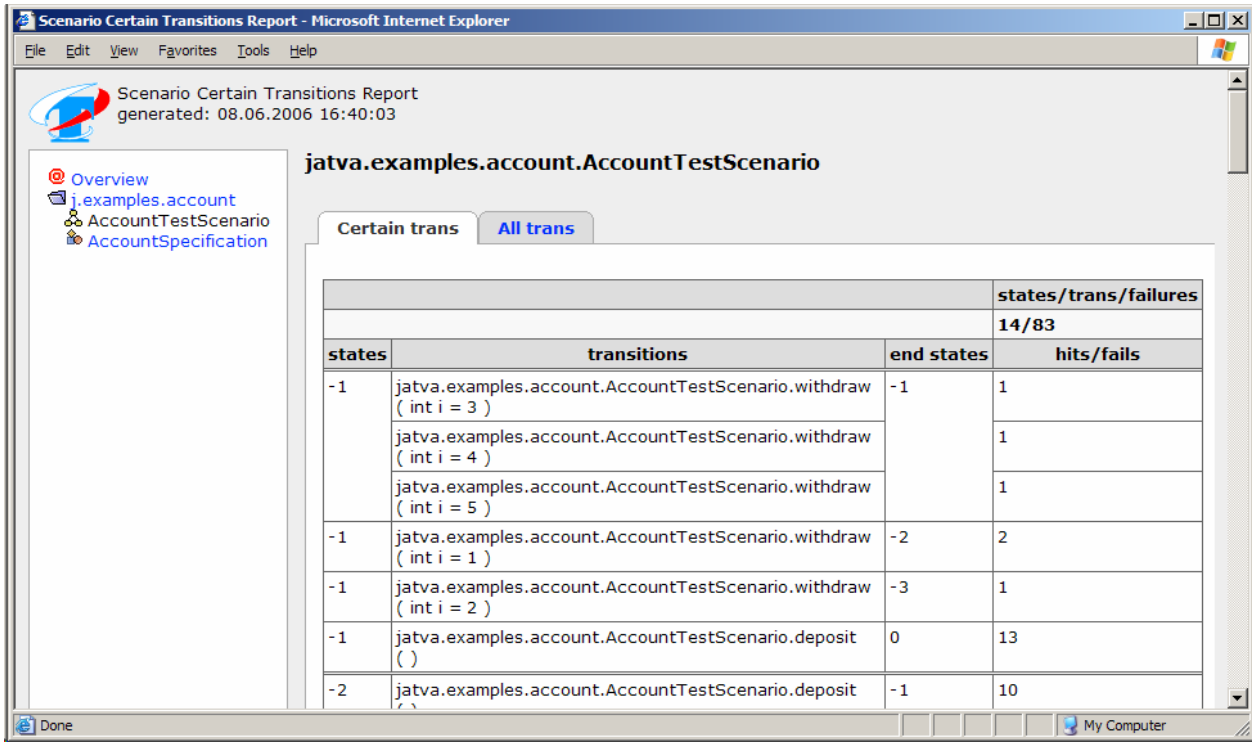

Рисунок 17. Scenario Certain Transitions Report.

Информация на странице **Scenario Certain Transitions Report** представлена в виде таблицы со следующими столбцами:

`states` – значение обобщенного состояния перед выполнением перехода

transitions – переход

`end states` – значение обобщенного состояния после выполнения перехода

`hits/fails` – число реализаций перехода и количество обнаруженных на нем ошибок

Так, при выполнении теста по сценарию для кредитного счета было десять переходов из состояния сценария **–1**, то есть при текущем значении баланса `-1`.

Переход, помеченный как **jatva.examples.account.AccountTestScenario.withdraw(int i = 1)** переводит сценарий в состояние **-2**. Переход совершается в состоянии `-1` со значением итерационной переменной `i` равным `1`. Данный переход был совершен **2** раза, нарушений при этом обнаружено не было.

Итоговый отчет

JavaTESK позволяет отобразить результаты выполнения нескольких тестовых сценариев на одной странице. Для этого создайте нужные трассы любым способом, выделите нужные в окне **Package Explorer** (например, зажмите клавишу **Ctrl** и нажимайте мышкой на узлы, соответствующие трассам, которые нужно включить в отчет). Затем из контекстного меню выделения (например, чтобы его получить, можно щелкнуть по выделению правой кнопкой мышки) вызовите **Generate Report**. Появится окно, изображенное на рисунке 12.

В результате будет сформирован отчет. На его стартовой странице будет отображено количество проверенных состояний и переходов и количество обнаруженных нарушений при выполнении каждого сценария теста.