

Тема 10

«Средства автоматизированного тестирования веб-ориентированных приложений»

Автоматизированное тестирование – основные понятия

Введение в автоматизацию

Начнём с определений.

Автоматизация тестирования (test automation) – набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения **НЕКОТОРЫХ** задач в процессе тестирования.



Инструментальное средство автоматизированного тестирования (test automation tool) – программа (или набор программ), позволяющая создавать, редактировать, отлаживать и выполнять автоматизированные тесты, а также **собирать статистику** их выполнения.



Что следует автоматизировать

Наибольший эффект автоматизация тестирования веб-ориентированных приложений даёт в следующих областях:

- проверка **ссылок**;
- проверка работоспособности **стандартного, типичного для множества проектов функционала**;
- проверка **стандартных элементов управления**;
- **базовая** проверка безопасности;
- тестирование **производительности** и **нагрузочное тестирование**;
- **смоук-тест для крупных систем**, где приходится выполнять большое количество примитивных трудоёмких задач;
- **регрессионное** тестирование;
- **конфигурационное тестирование** в контексте проверки работоспособности приложения при тех или иных настройках;
- часто повторяющиеся **тесты, простые для автоматизации**;
- длинные **утомительные для человека тесты**.

Добавьте что-нибудь?

Учитываемые факторы



На разработку автоматизированных тестов может уходить **в 5-10 раз больше времени**, чем на создание и однократное выполнение аналогичных ручных тестов.

Приступая к автоматизации, **следует учесть**:

- Затраты времени **на ручное выполнение тестов и на выполнение автоматизированных тестов**.
- Количество **повторений** тестов.
- Затраты времени на **отладку, обновление и поддержку** автоматизированных тестов.

Автоматизацией, как правило, **занимаются самые квалифицированные сотрудники**, которые в это время не могут решать иные задачи.

Преимущества автоматизации

Успешная автоматизация даёт следующие **преимущества**:

- **скорость** (компьютер работает быстрее человека; мы экономим время и, как следствие, деньги);
- **надёжность** (компьютер не допускает «человеческих ошибок»);
- **мощность** (можно выполнить действия, недоступные человеку);
- средства автоматизации тестирования **собирают числовую информацию** и представляют её в **удобной для понимания** человеком форме;
- средства автоматизации тестирования в сложных ошибочных ситуациях **способны выполнять «низкоуровневые действия»**, сложные для человека.



Недостатки автоматизации



Автоматизация тестирования обладает следующими **недостатками**:

- необходим **квалифицированный персонал**;
- необходима **грамотная стратегия** разработки и управления тестами, тестовыми данными и т.п.;
- необходимы **специальные инструментальные средства** (зачастую – **ОЧЕНЬ** дорогие);
- в случае серьёзных изменений в приложении многие автоматизированные **тесты становятся бесполезными** и/или требуют серьёзной переработки;
- **на автоматизацию часто возлагают неоправданные надежды, что ведёт к срывам сроков и прочим проблемам.**

Технология Record&Playback

Одной из наиболее **распространённых** и **простых** для понимания технологий автоматизации тестирования является технология **Record&Playback** («Записать и воспроизвести»).

Суть её заключается в том, что средство автоматизации тестирования позволяет **выполнить с тестируемым приложением некоторый набор действий, которые будут записаны на специальном языке программирования, а затем могут быть воспроизведены.**



Технология Record&Playback



Преимущества:

- создание «скелета» теста **ускоряется**;
- средство автоматизации само **собирает техническую информацию** о приложении;
- это **просто для понимания** новичками.

Недостатки:

- записанные тесты **содержат т.н. «hard-coded» («жёстко закодированные») значения**, которые приходится потом заменять вручную.
- средство автоматизации записывает ВСЁ, в т.ч. **МНОГО-МНОГО ЛИШНЕГО**;
- если приложение достаточно сильно изменилась, **тест придётся перезаписывать**.

Вывод:

- технология R&P **хороша в качестве помощника**, но она не выполнит за человека ВСЕ необходимые для автоматизации тестирования действия.

Data-Driven и Keyword-Driven тестирование

А автоматизации тестирования существует проблема **создания достаточно универсальных и используемых повторно тестов.**

Решить эти задачи помогают два подхода:

- **тестирование под управлением данными** (Data-Driven testing) – вынесение данных теста из самого теста;

и

- **тестирование под управлением ключевыми словами** (Keyword-Driven testing) – вынесение логики теста из самого теста.



Семейство инструментальных средств Selenium

Selenium



Selenium – набор программного обеспечения для **тестирования веб-ориентированных приложений**.

Входящие в этот набор инструментальные средства:

- **кроссплатформенные;**
- **бесплатные;**
- **очень широко распространены.**



Selenium

Основные продукты в составе Selenium:



Selenium IDE – плагин для браузера FireFox. Применяется для записи тестов с последующей передачей их в Selenium RC.



Selenium Remote Control (RC) – средство выполнения тестов под различными браузерами и операционными системами.



Selenium GRID – средство параллельного выполнения крупномасштабных тестов.

Полный список продуктов:
<http://seleniumhq.org/projects/>

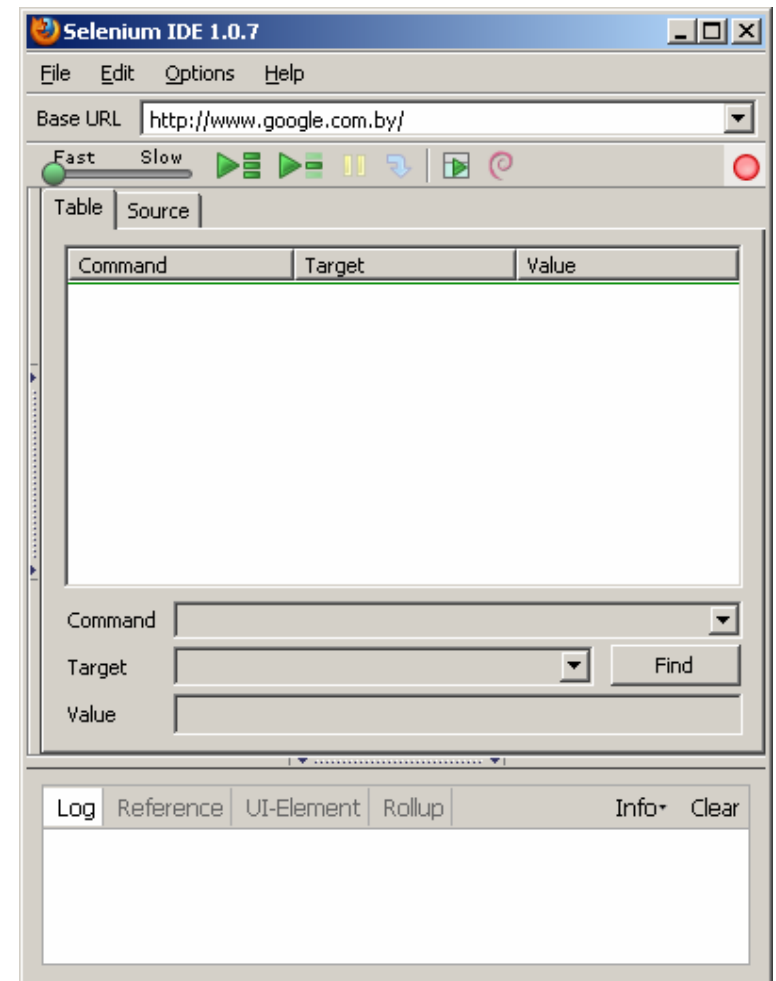
Selenium IDE

Selenium IDE

Selenium IDE – интегрированная среда для разработки и выполнения скриптов, представленная в виде плагина для браузера FireFox.

Свежая версия всегда доступна по адресу:

<http://seleniumhq.org/projects/ide/>



Возможности Selenium IDE

- В полной мере поддерживается технология **Record and Playback**.
- Широкие возможности по **идентификации элементов страницы** через идентификаторы, имена и локаторы XPath.
- **Автозаполнение** при наборе команд.
- Возможность выполнения тестов на **разной скорости**, возможность выполнения **отладочного прогона тестов**, возможность указания **точек остановки**.
- Возможность **сохранения тестов в виде HTML** или **экспорта** их в такие распространённые языки программирования как **PHP, Perl, Java, C#** и т.п.
- Поддерживается возможность **дополнительного расширения функционала с помощью плагинов**.

Selenium IDE поддерживает...

Операционные системы:

Windows,



Linux,



OS X,



Solaris



Языки программирования:

PHP,



Perl,



Java,



C#,



Python,



Ruby



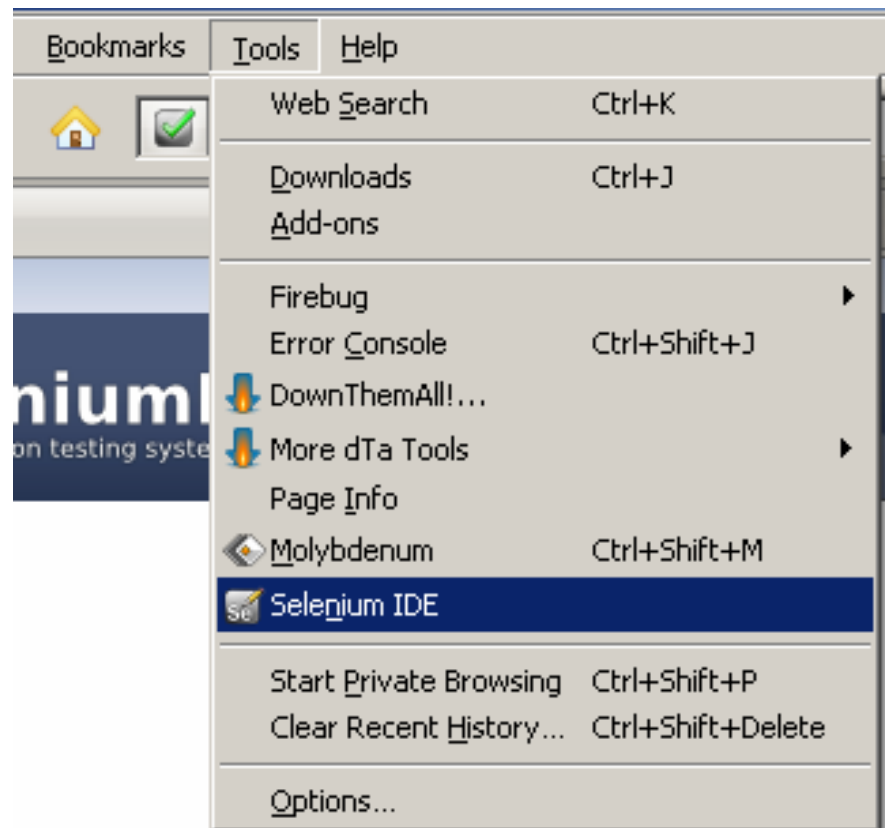
Браузеры:

Firefox, IE (*RC), Safari (*RC), Opera (*RC)



Использование Selenium IDE

После установки плагин **Selenium IDE** доступен в браузере FireFox в разделе «Инструменты» («**Tools**»):



Компоненты Selenium IDE

Поле «Command» содержит команду («что необходимо сделать»)

Поле «Target»
указывает
целевой
элемент («с
чем это
сделать»)

The image shows a screenshot of the Selenium IDE interface. It features three main input fields stacked vertically: 'Command', 'Target', and 'Value'. The 'Command' field has a dropdown arrow on its right side. To the right of the 'Target' field is a 'Find' button. Arrows from the surrounding text point to each of these three fields. At the bottom of the interface, there is a status bar with a progress indicator.

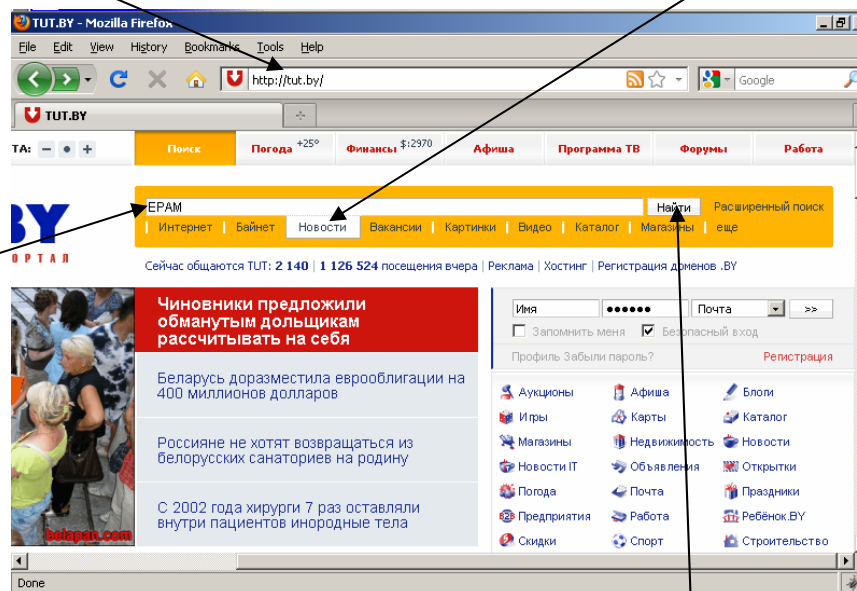
Поле «Value» содержит значение, с которым выполняется некоторая операция.

Пример для наглядности

1. Открыть
«http://tut.by»

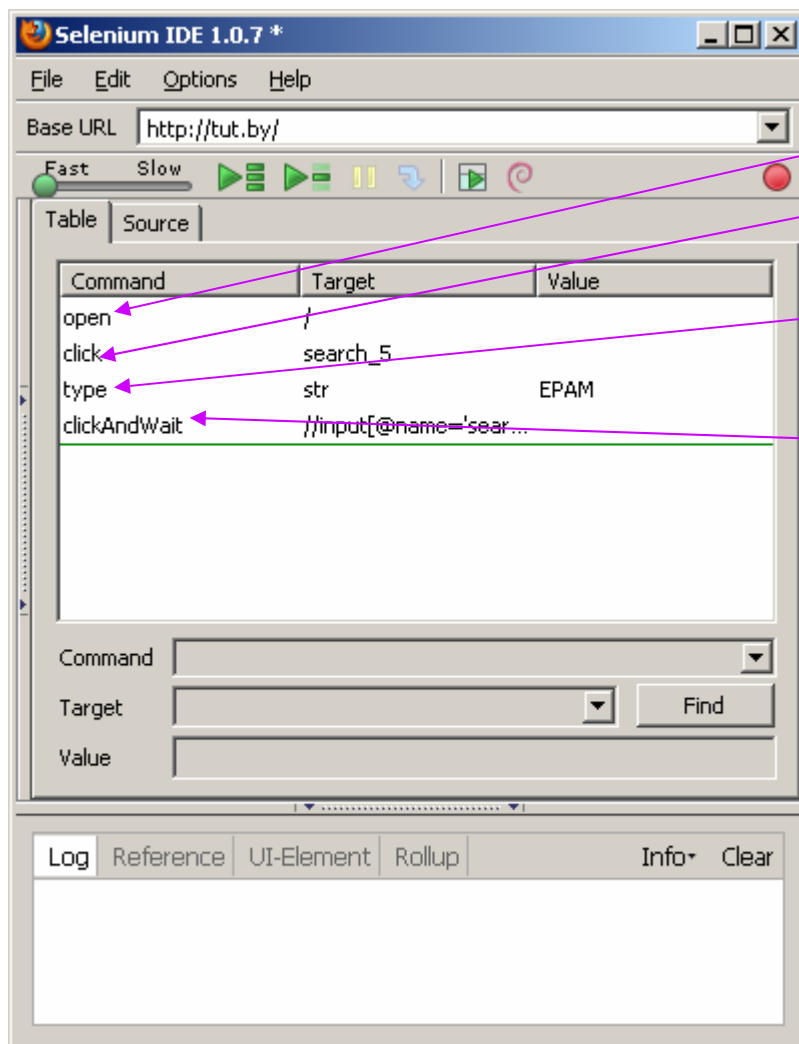
2. Кликнуть «Новости»

3. Ввести
«ЕРАМ»



4. Кликнуть
«Найти»

Пример для наглядности

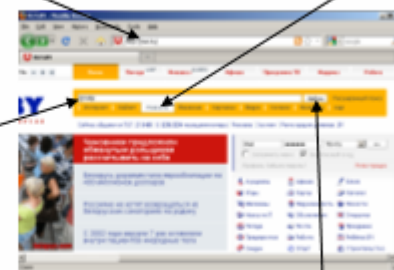


1. Открыть
«http://tut.by»

2. Кликнуть «Новости»

3. Ввести
«EPAM»

4. Кликнуть
«Найти»



Ключевые поля Selenium IDE

Итак, ещё раз о **самых главных полях**.

Command

Command

Target

Target

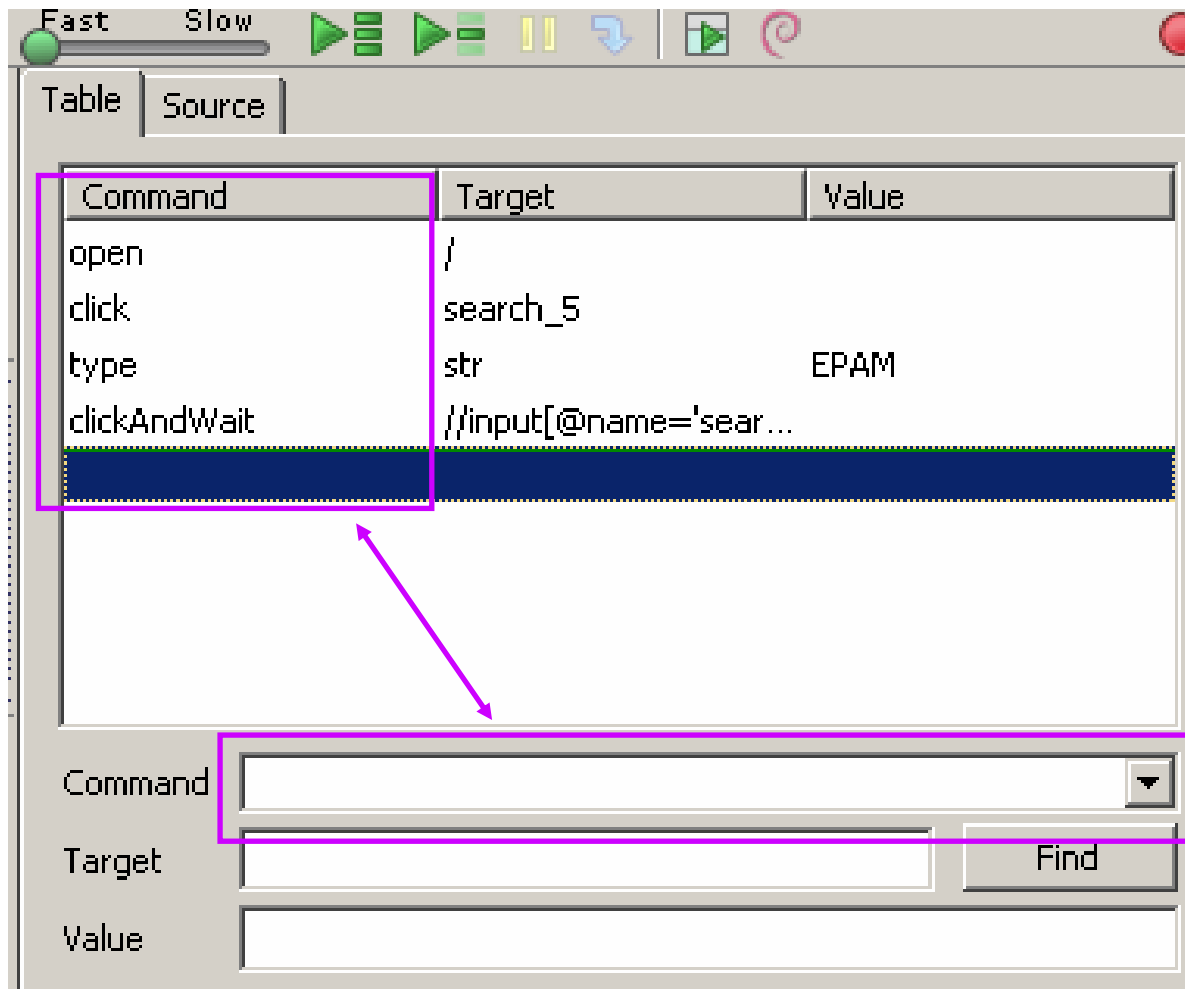
Find

Value

Value

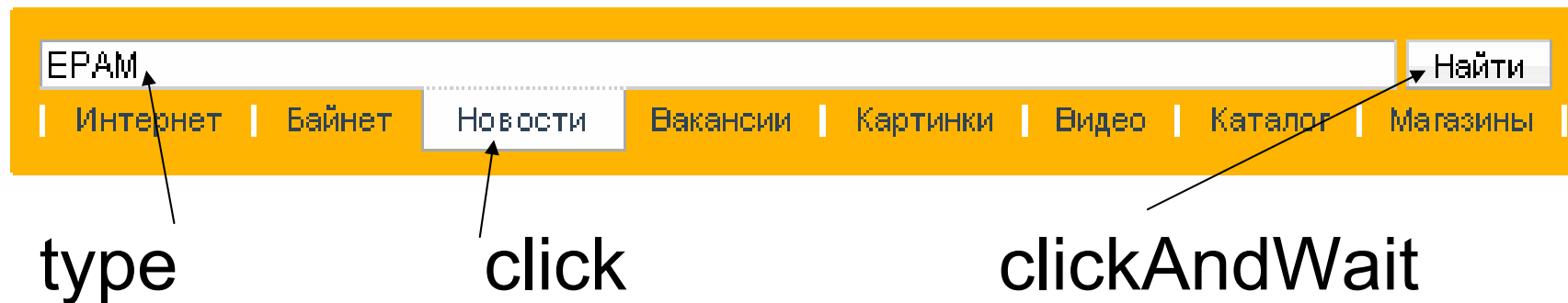
Ключевые поля: command

Поле «Command» содержит **указание** того, что **необходимо выполнить** на данном шаге теста.



Ключевые поля: command, действия

Примеры **действий** (actions):



Ошибка (невозможность) выполнения любой из этих команд приводит к остановке теста!

Ключевые поля: command, переменные

Пример работы с **переменными** (accessors):

The diagram illustrates the use of variables in a debugger's command window. It consists of two screenshots of the command window interface.

Left Screenshot:

	store	myvar	MyValue
type		str	\$(myvar)

Command: store
Target: myvar
Value: MyValue

Right Screenshot:

	store	myvar	MyValue
type		str	\$(myvar)

Command: type
Target: str
Value: \$(myvar)

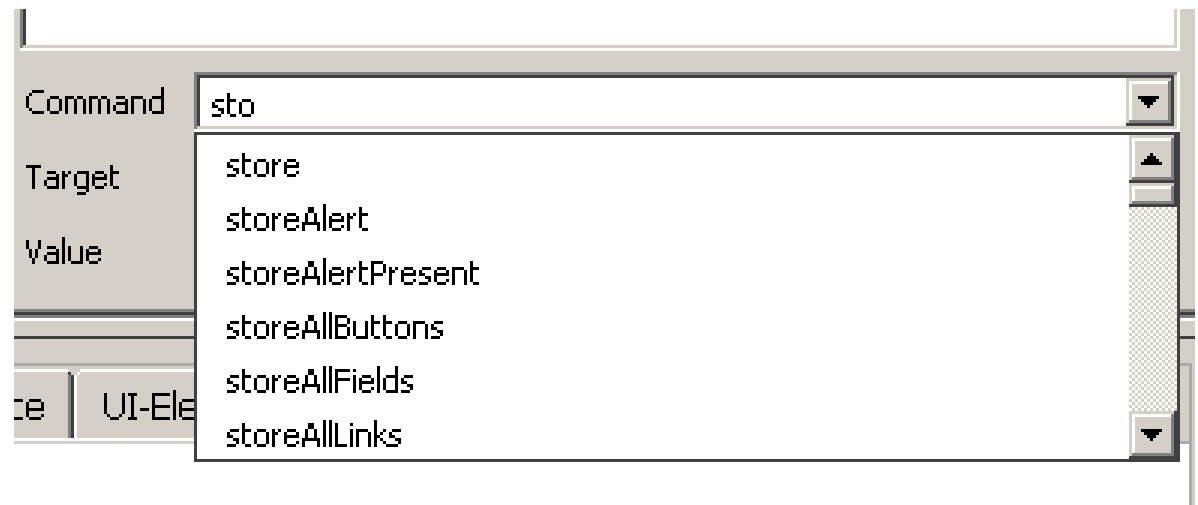
Annotations:

- Сохранение значения в переменную** (Saving the value in the variable) points to the 'store' command and the 'myvar' target in the left screenshot.
- Использование переменной** (Using the variable) points to the '\$(myvar)' value in the 'type' command of the right screenshot.

Ключевые поля: `command`, переменные

Пример часто используемых **accessors**:

- `store(expression, variableName)`
- `storeElementPresent(locator, variableName)`
- `storeText(locator, variableName)`
- `storeTitle(variableName)`



Ключевые поля: command, проверки

Проверки используются для **анализа состояния** веб-ориентированного приложения.

Например, можно проверять **наличие** того или иного **элемента** страницы, **значение** того или иного **поля** и т.п.

assertValue	str	EPAM
Command	assertValue	
Target	str	Find
Value	EPAM	

Ключевые поля: `command`, проверки

Пример некоторых проверок (asserts):

- `assertEditable(locator)`
- `assertElementPresent(locator)`
- `assertNotVisible(locator)`
- `assertText(locator, pattern)`

Ошибка (невозможность) выполнения любой проверки приводит к остановке теста!

Ключевые поля: command, проверки

Нижеприведённый пример показывает, как можно проверить **title** страницы:

assertTitle		Selenium Basics &...	
Command	assertTitle		
Target		Find	
Value	Selenium Basics — Selenium Documentation		

Ключевые поля: `command`, проверки

Существует ещё один тип проверок, команды которого начинаются с ключевого слова `verify`, например:


- `verifyTextPresent(pattern)`
- `verifyText(locator, pattern)`
- `verifyElementPresent(locator)`

Ошибка такой проверки **НЕ приводит** к остановке теста!

Ключевые поля: command, проверки

Нижеприведённый пример показывает, **КАК ЕЩЁ** можно проверить **title** страницы:

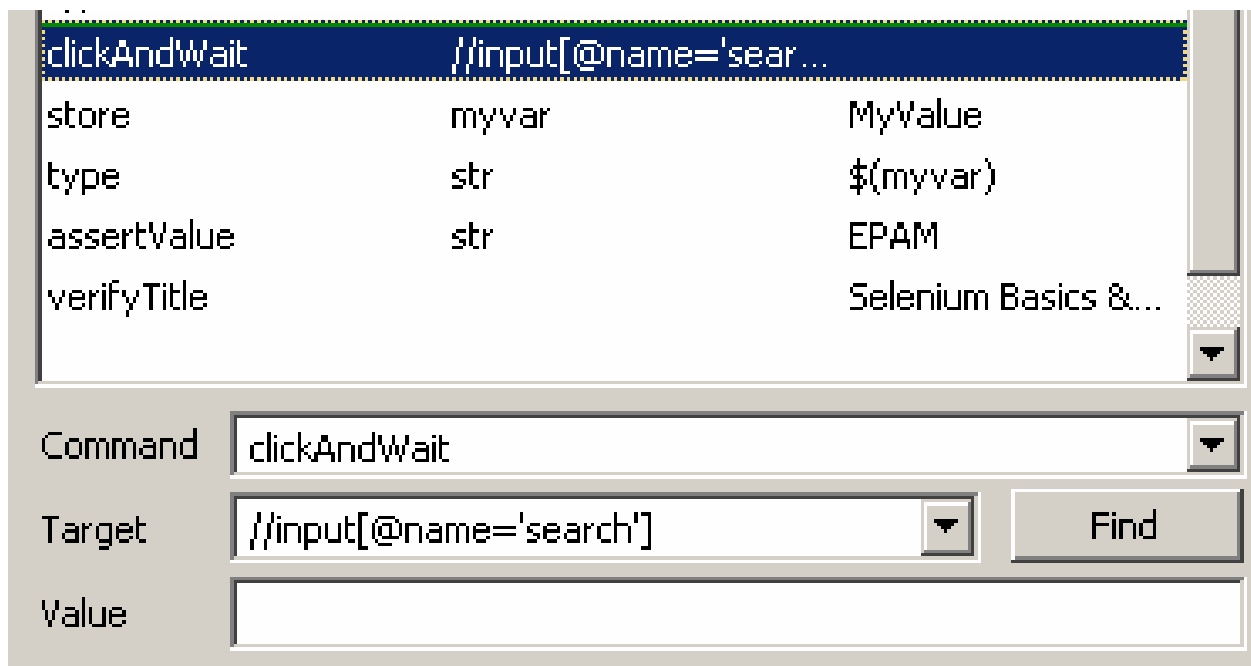
Command	Target	Value
verifyTitle		Selenium Basics &mdash...

Command	<input type="text" value="verifyTitle"/>		
Target	<input type="text"/>	<input type="button" value="Find"/>	
Value	<input type="text" value="Selenium Basics &mdash; Selenium Documentation "/>		

Ключевые поля: **command**, ожидание

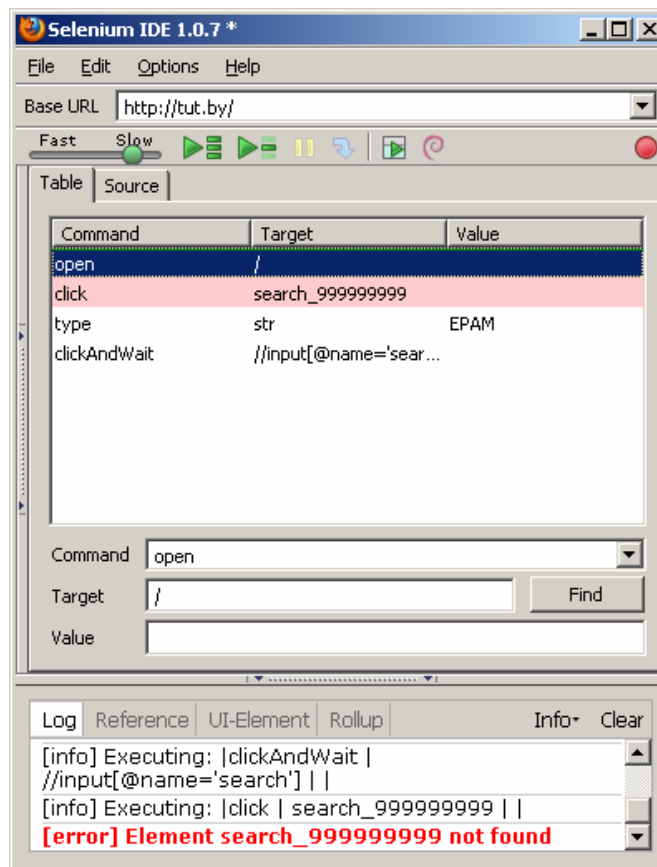
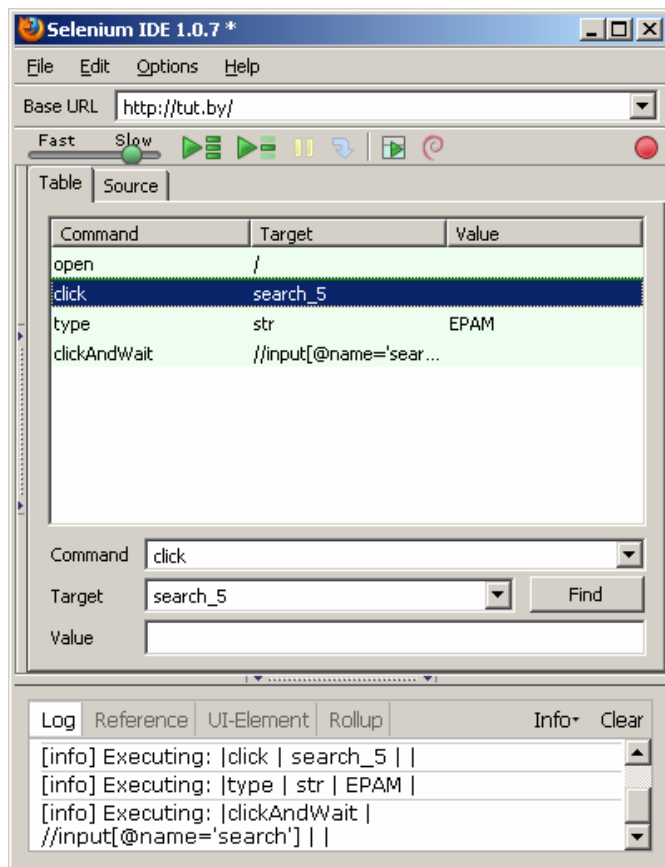
Существует ещё один класс команд, начинающихся с ключевых слов **waitFor**. Эти команды предназначены для работы с элементами, появление или изменение состояния которых **требует некоторого времени**.

Ошибка (невозможность) выполнения такой команды приводит к остановке теста!



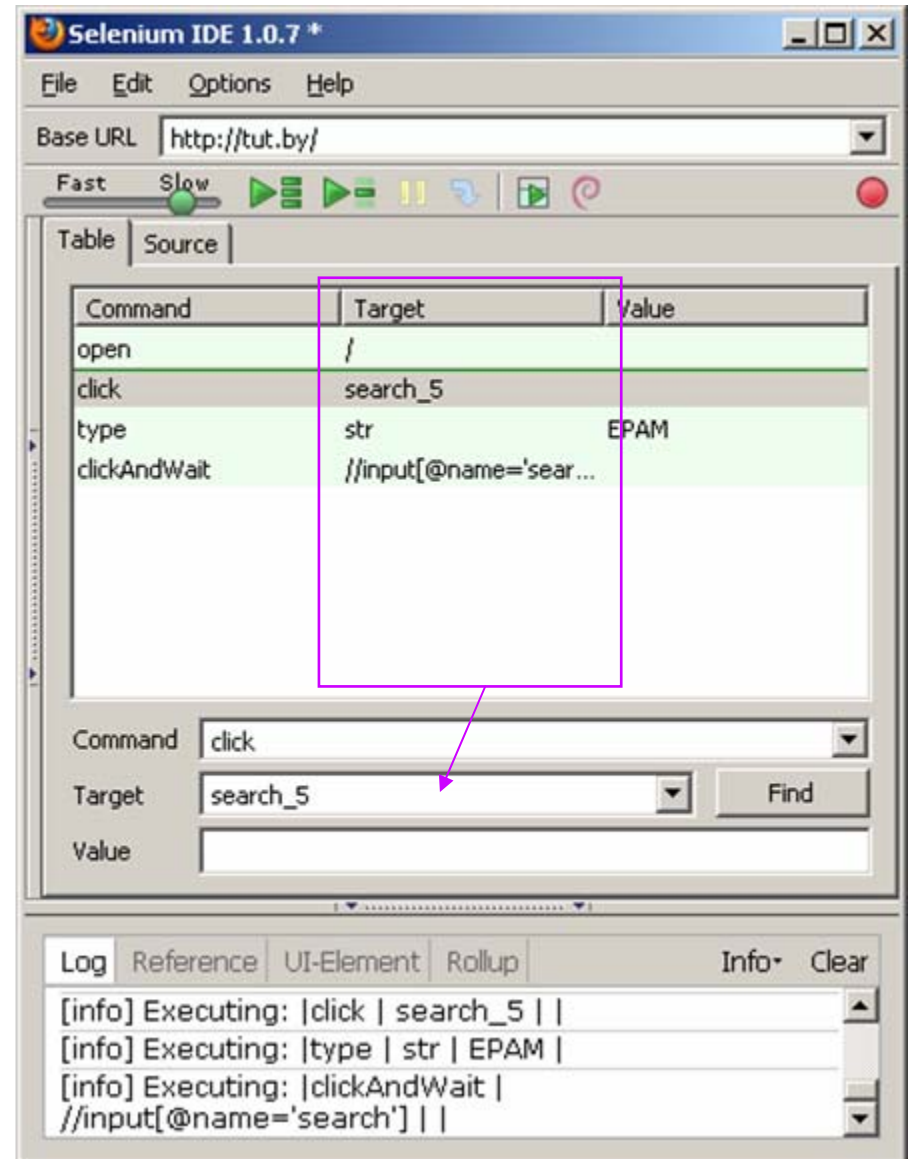
Ключевые поля: log (протокол)

За ходом и результатом выполнения тестов можно следить с помощью поля **Log**, в котором отражаются **все выполняемые Selenium IDE действия**.



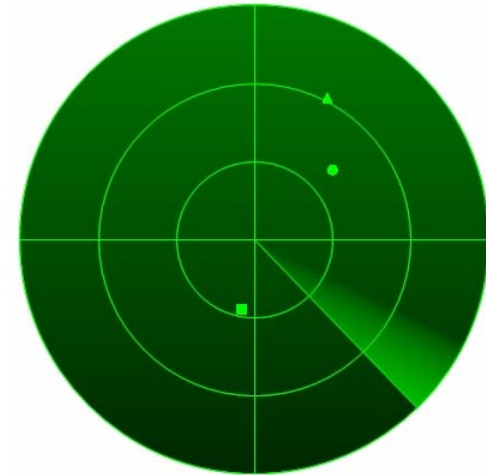
Ключевые поля: цель (target)

Как уже было сказано ранее, одним из важных полей является поле «**Target**», которое указывает, с каким элементом следует выполнить действие:



Ключевые поля: локатор (locator)

Указание на элемент, с которым необходимо выполнить действие, производится при помощи т.н. «локаторов» (англ. locator не является неточным аналогом русского «локатор», однако из-за созвучности прижилась такая транслитерация слова).



Итак, локатор **указывает на HTML-элемент**, с которым необходимо выполнить действие.

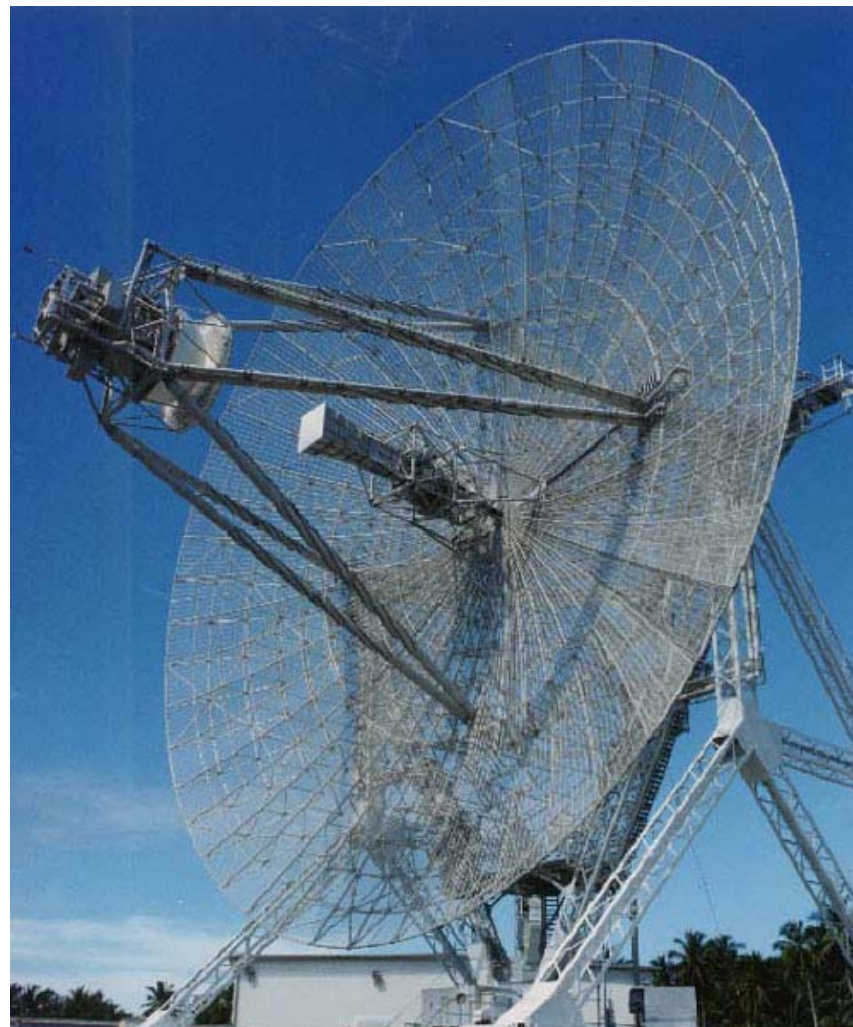
Формат локатора таков:

`LocatorType = Argument`

Ключевые поля: локатор (locator)

Локаторы бывают **следующих типов**:

- `id = ElementID`
- `name = ElementName`
- `link = LinkText`
- `xpath = XPath`



Ключевые поля: локатор (locator, id)

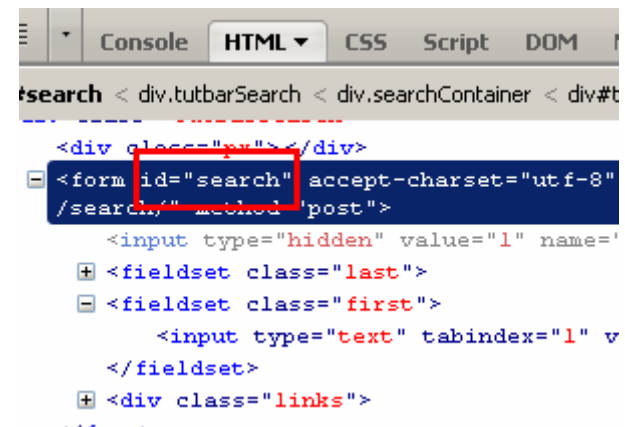
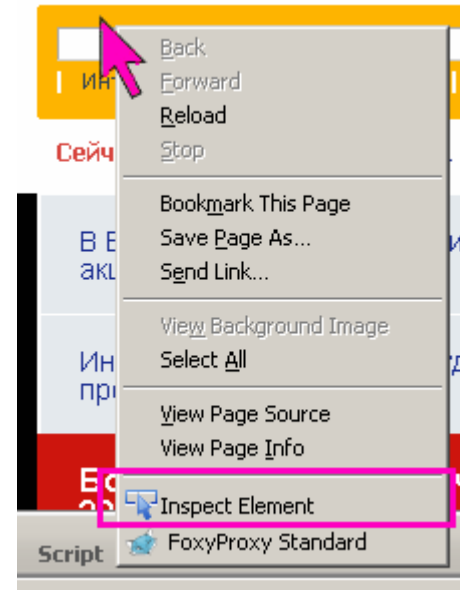
Локатор вида

`id = ElementID`

МОЖНО (ХОТЬ И НЕЖЕЛАТЕЛЬНО) записывать просто в виде `ElementID`.

Идентификатор элемента нужно смотреть в HTML-коде страницы.

Очень удобно для этого использовать функцию расширения Firefox Firebug "Inspect Element".



Ключевые поля: локатор (locator, name)

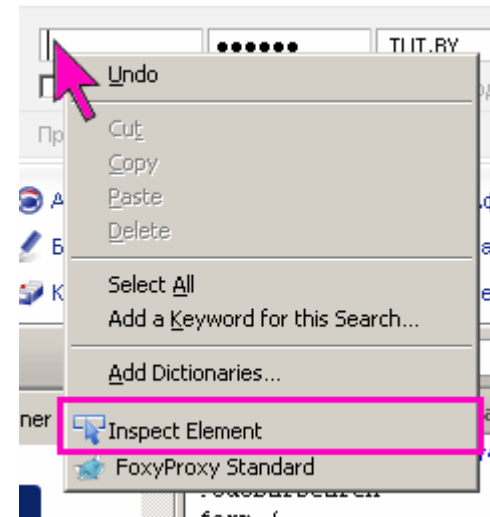
Локатор вида

`id = ElementName`

тоже можно (хоть и ТОЖЕ НЕЖЕЛАТЕЛЬНО) записывать просто в виде `ElementName`.

Имя элемента тоже нужно смотреть в HTML-коде страницы.

И опять очень удобно для этого использовать функцию расширения Firefox Firebug "Inspect Element".



```
.set>  
iv class="ulSubmit">  
iv class="ulFields">  
  <input class="fld" type="text" name="login" ta  
  <input class="fld" type="password" name="passw  
  <select tabindex="5" name="sid">  
div>  
dset>
```

Ключевые поля: локатор (locator, link)

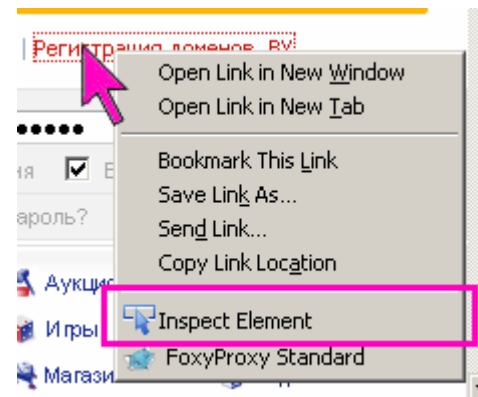
Локатор вида

`id = LinkText`

НЕЛЬЗЯ записывать просто в виде `LinkText`.

Текст ссылки чаще всего виден «невооружённым глазом», но если посмотреть его в коде — хуже не будет.

И опять же: очень удобно для этого использовать функцию расширения Firefox Firebug “Inspect Element”.



```
> | </span>  
title="Регистрация доменов  
www.hoster.by/regby/"> Регистрация доменов .BY </a>
```


Ключевые поля: локатор, краткая запись

Прежде, чем мы перейдём к рассмотрению **XPath-локаторов**, ещё раз напомним:

1. Лучше **НЕ** использовать сокращённую форму записи локаторов (**это может привести к неоднозначности определения элемента**), но если очень хочется...

2. Можно сокращать локаторы типа **id** или **name**, но **НЕЛЬЗЯ** сокращать локаторы типа **link**.

Command	
Target	id=regform
Value	



Command	
Target	regform
Value	

Command	
Target	name=user_password
Value	



Command	
Target	user_password
Value	

Command	
Target	link=Регистрация доменов .BY
Value	

Ключевые поля: локатор (locator, xpath)

Локатор вида

xpath = XPath

является **самым универсальным**. Он позволяет **гибко и при том однозначно** идентифицировать **любой элемент** на странице.

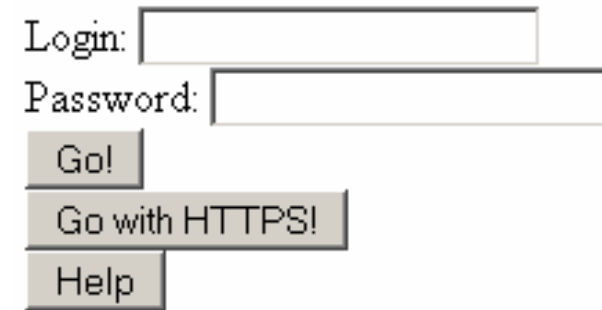
Подробно про XPath можно почитать здесь:

<http://www.w3schools.com/xpath/>

<http://www.w3.org/TR/xpath/>

Ключевые поля: локатор (locator, xpath)

Итак, допустим, перед нами есть
форма вида:



Login:

Password:

Go!

Go with HTTPS!

Help

```
<form name="frm1" id="frm1">  
  Login: <input type="text" name="ul" /><br />  
  Password: <input type="password"  
             name="up" /><br />  
  <input type="submit" name="do_login"  
           value="Go!" /><br />  
  <input type="button" name="do_login"  
           value="Go with HTTPS!" /><br />  
  <input type="button" value="Help" /><br />  
</form>
```

Ключевые поля: локатор (locator, xpath)

Сначала рассмотрим, как можно обратиться к самой форме:

- id=frm1
- name=frm1
- frm1

или с XPath

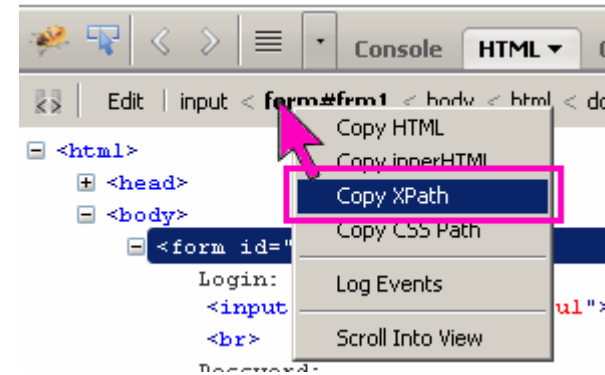
- //html/body/form[1]
- //form[1]
- //form[@id="frm1"]
- //form[@name="frm1"]
- //form[@name="frm1" or @id="frm1"]
- //form[@name="frm1" and @id="frm1"]



Ключевые поля: локатор (locator, xpath)

Как видно из примера, локаторы XPath позволяют обращаться к элементу по:

- его расположению в документе;
- указанию типа элемента и идентификационных данных;
- указанию идентификационных данных (причём с возможностью задавать условия).



Самым простым способом узнать XPath элемента является использование Firebug.

Однако, полученные XPath являются самыми «неуниверсальными», т.к. построены на основе прямого анализа структуры документа.

Ещё XPath записывает сам Selenium IDE – они получаются чуть-чуть лучше, но тоже требуют доработки.

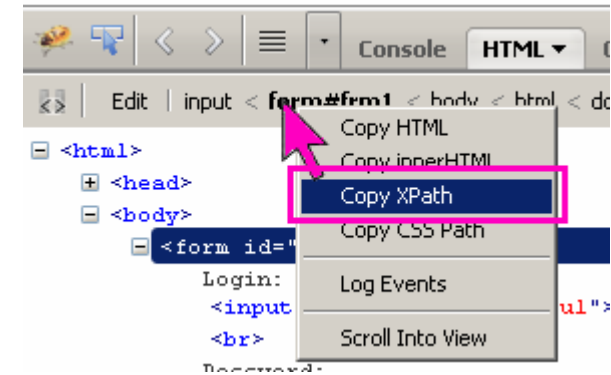
Ключевые поля: локатор (locator, xpath)

Чем **плох** XPath вида `//html/body/form[1]` ?

Тем, что как только форма будет перемещена куда-то на странице (вложена в div, таблицу и т.п.) – этот XPath станет «невалидным» (неверным).

Потому XPath следующих видов предпочтительнее:

- `//form[@id="frm1"]`
- `//form[@name="frm1"]`
- `//form[@name="frm1" or @id="frm1"]`



Ключевые поля: локатор (locator, xpath)

Сейчас давайте рассмотрим ещё **несколько примеров** использования XPath.

Будем обращаться к **элементам** формы:

1. `//input[@name="ul"]`
2. `//form[1]/input[2]`
3. `//input[@type="submit"]`
4. `//input[@name="do_login" and @type="button"]`
5. `//form[@id="frm1"]/input[5]`

Обратите внимание, что для кнопки «Help» обращение по номеру — **единственно возможное**: у неё нет ни имени, ни идентификатора.

Login:

Password:

```
<body>
  <form name="frm1" id="frm1">
    1 Login: <input type="text" name="ul" /><br />
    2 Password: <input type="password" name="up" /><br />
    3 <input type="submit" name="do_login" value="Go!" />
    4 <input type="button" name="do_login" value="Go with HTTPS!" />
    5 <input type="button" value="Help" /><br />
  </form>
</body>
```

Ключевые поля: локатор (locator, xpath)

Как вы могли заметить, мы активно использовали синтаксис

@IDTYPE="IDVALUE"
(например, //input[@name="ul"])

Всего существует пять основных вариантов IDTYPE:

- @id
- @name
- @type
- @class
- text()

(//*[text()="Some Text"] для элементов типа «Some Text»)

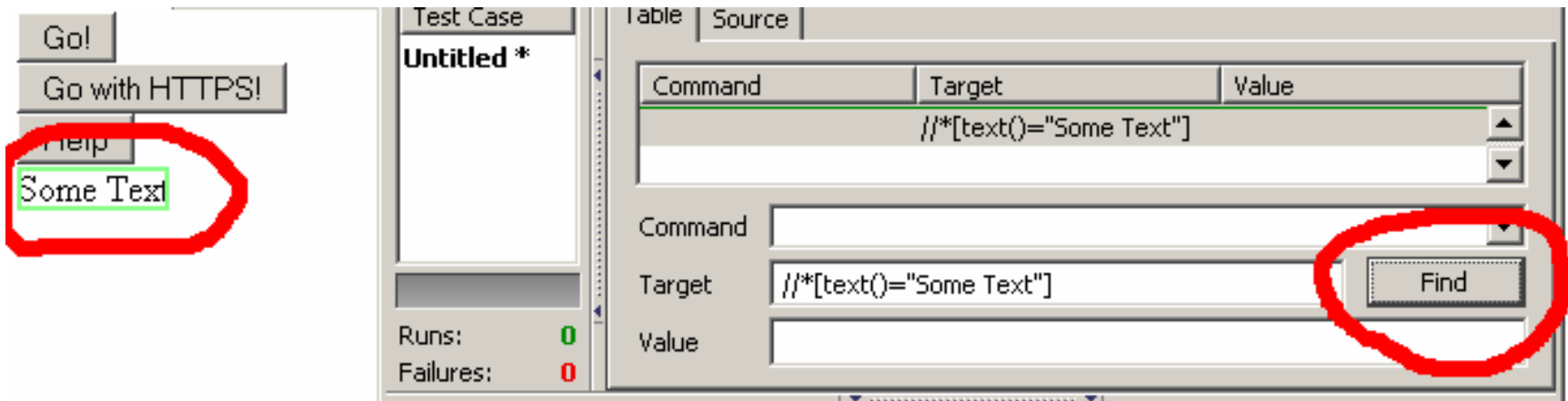


Ключевые поля: локатор (locator, xpath)

Как проверить, что вы написали «валидный» (правильный) локатор?

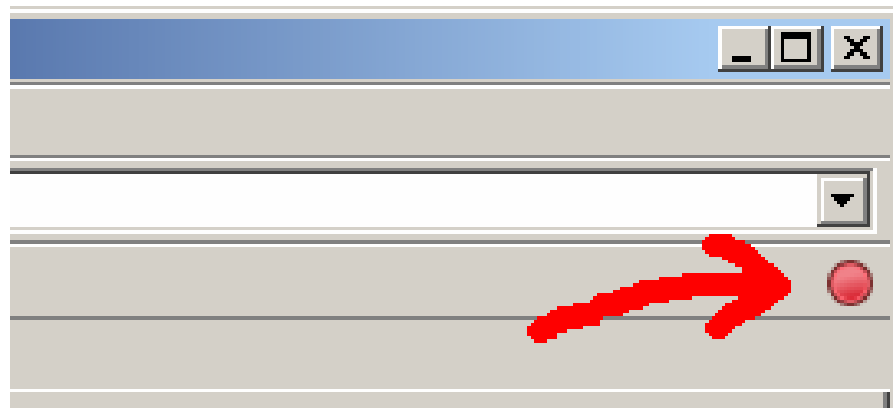
Selenium IDE позволяет после ввода локатора кликнуть кнопку «Find» и, в случае валидного локатора, подсветит соответствующий элемент зелёной рамкой.

В случае невалидного локатора в лог будет записана информация о невозможности обнаружить элемент.



Записываем и воспроизводим тест

Сразу же после запуска Selenium IDE находится в режиме записи теста, выключать и повторно включать который можно кнопкой в правом верхнем углу окна Selenium IDE.



Запись имеет смысл приостанавливать, если вы выполняете действия, не относящиеся к тесту.

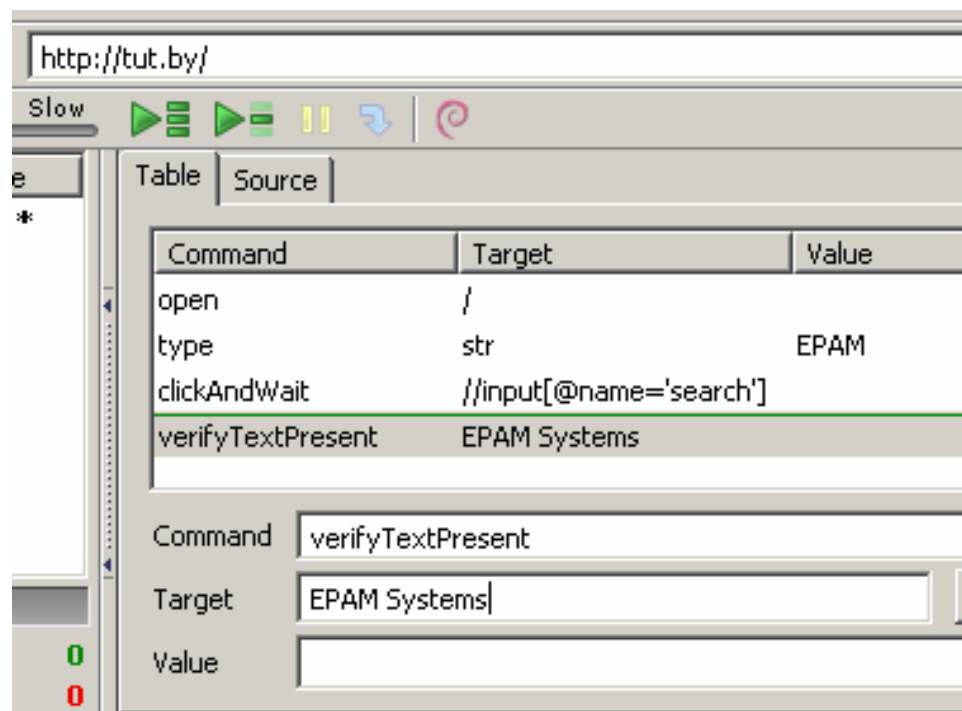
Записываем и воспроизводим тест

После включения записи вы можете просто делать с приложением то, что следует из шагов теста.

Selenium IDE всё запишет.

Итак, ещё раз примитивный тест:

1. Открыть tut.by
2. В строку поиска ввести «EPAM».
3. Кликнуть «Найти».
4. Проверить, что на странице есть текст «EPAM Systems».



Записываем и воспроизводим тест

Для выполнения записанного теста есть несколько элементов в левом верхнем углу экрана:

- регулятор скорости выполнения;
- кнопка выполнения всех тестов;
- кнопка выполнения текущего теста.

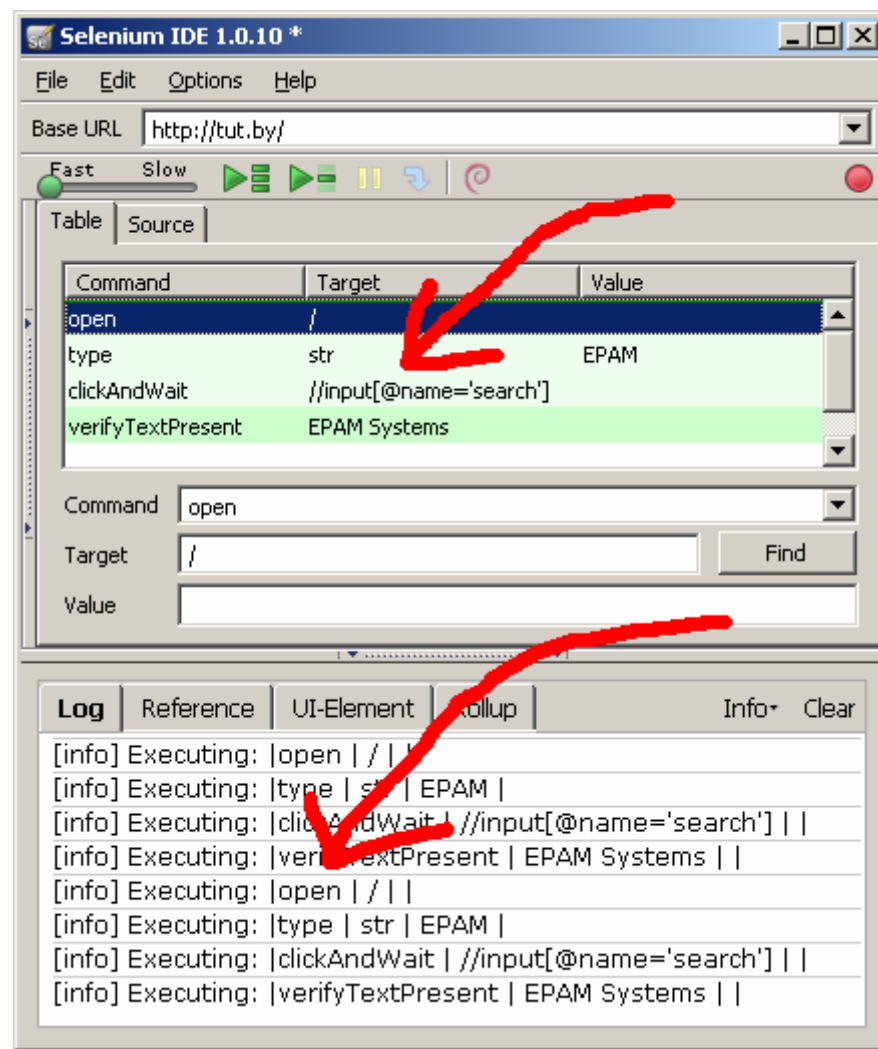
Рекомендуется после отладки выполнить тест на максимальной скорости, чтобы увидеть, где вы упустили «waitFor...»



Записываем и воспроизводим тест

В процессе воспроизведения теста Selenium IDE в реальном времени показывает, успешно ли выполнена та или иная команда:

- подсветкой команды зелёным или розовым в списке команд;
- сообщениями в логе.



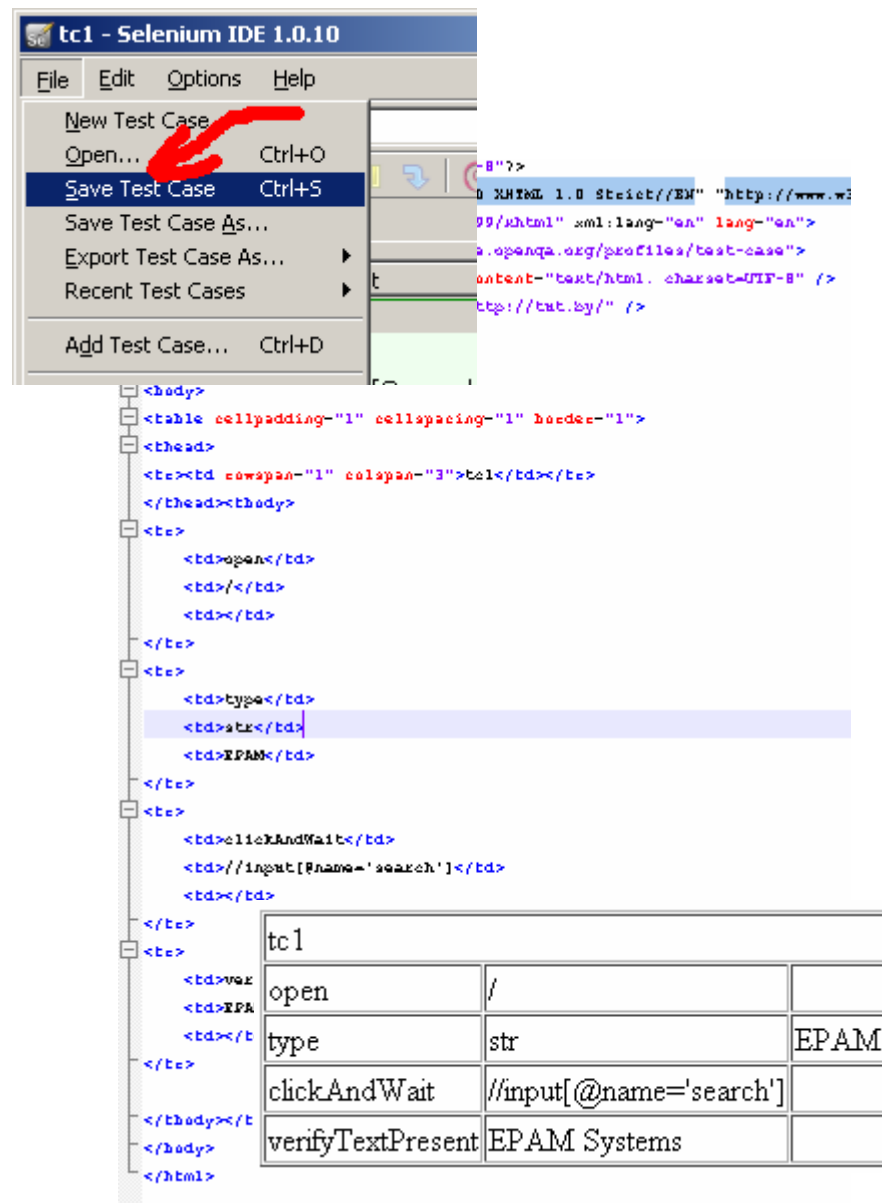
Сохраняем тест

Когда тест готов, его нужно **сохранить**.

Тесты, с которыми впоследствии можно будет продолжить работу в IDE, **сохраняются в виде обычной HTML-страницы** с определённой структурой.

Для ускорения просмотра их можно открывать **в браузере как обычные страницы**.

Несколько тестов (сценарий) можно **сохранить в виде «тест-сюта»**.



The screenshot shows the Selenium IDE 1.0.10 interface. The 'File' menu is open, and the 'Save Test Case' option is highlighted with a red arrow. The menu also shows 'New Test Case', 'Open...', 'Save Test Case As...', 'Export Test Case As...', 'Recent Test Cases', and 'Add Test Case...'. Below the menu, the HTML structure of the saved test case is displayed. It is an HTML document with a table containing the test steps. The table has three columns: the first column contains the step name, the second column contains the step arguments, and the third column is empty. The steps are: 'open' with argument '/', 'type' with argument 'str', 'clickAndWait' with argument '//input[@name='search']', and 'verifyTextPresent' with argument 'EPAM Systems'.

```
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr>
<td colspan="1" rowspan="1" colspan="3">test</td></tr>
</thead><tbody>
<tr>
<td>open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>str</td>
<td>EPAM</td>
</tr>
<tr>
<td>clickAndWait</td>
<td>//input[@name='search']</td>
<td></td>
</tr>
<tr>
<td>verifyTextPresent</td>
<td>EPAM Systems</td>
<td></td>
</tr>
</tbody></table>
</body>
</html>
```

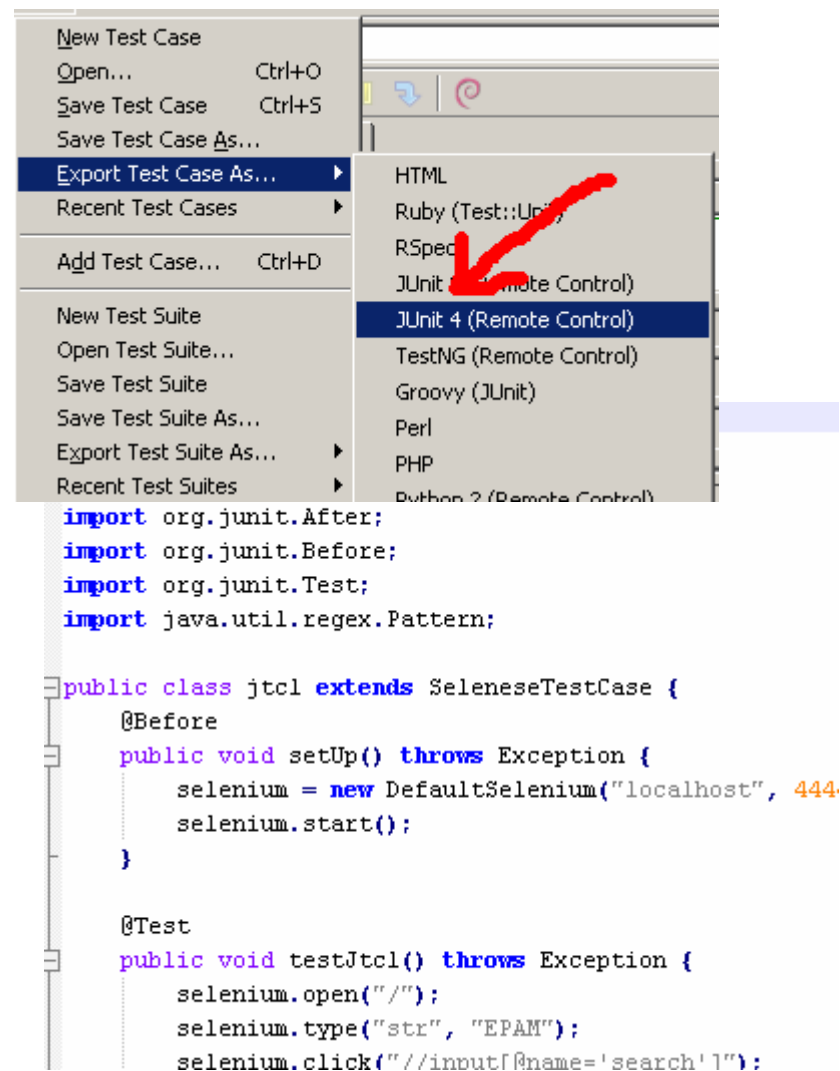
tc1		
open	/	
type	str	EPAM
clickAndWait	//input[@name='search']	
verifyTextPresent	EPAM Systems	

Экспортируем тест

У Selenium IDE есть один серьёзный **недостаток** — **тесты в нём «линейны» и примитивны**. Всё же здесь у нас нет полноценного языка программирования.

Поэтому для действительно серьёзной работы тест нужно **экспортировать в соответствующий формат**.

Далее мы будем рассматривать Selenium RC в работе с Java, а потому экспортируем наш тест в JUnit 4.



Промежуточное заключение



На этом наше знакомство с Selenium IDE завершено, и мы переходим к Selenium RC (Remote Control).

Тест для проверки изученного

1. Что такое «автоматизированное тестирование»?
2. В каких областях тестирования веб-ориентированных приложений автоматизация даёт наибольший эффект?
3. Какие факторы следует учитывать при решении о применении или неприменении автоматизации тех или иных тестов?
4. В чём основные преимущества и недостатки автоматизации тестирования?
5. Что такое технология «Record and Playback»? В чём её основные преимущества и недостатки?
6. Что такое тестирование под управлением ключевыми словами и тестирование под управлением данными? Чего позволяют достичь эти технологии?
7. Что такое Selenium IDE? Каковы его основные возможности?
8. Что такое «локатор»? Как его можно указывать?
9. Зачем в Selenium IDE присутствуют команды с эффектом ожидания (waitFor...)?
10. Как в Selenium IDE обратиться к элементу HTML-страницы, у которого нет ни id, ни имени, ни класса, ни типа и даже его содержимое (текст) нам неизвестно?

Модульное тестирование, JUnit

Определения

Модульное тестирование (unit-testing) – разновидность автоматизированного тестирования, при котором создаются тесты, отвечающие следующим требованиям:

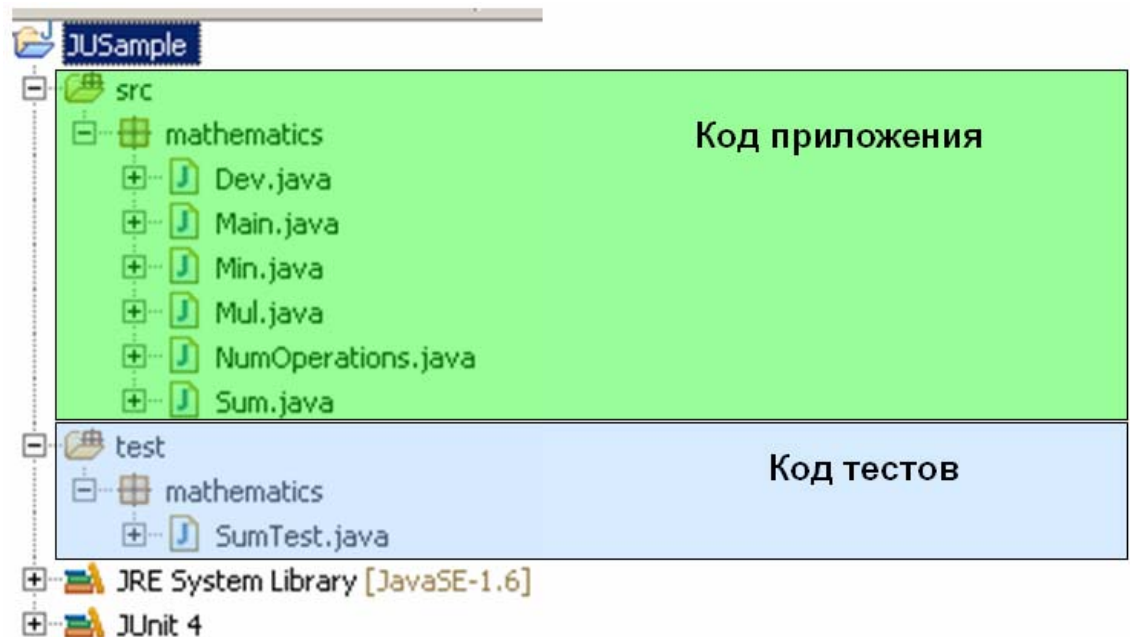
- тесты пишутся **на том же языке программирования**, на котором разрабатывается приложение;
- тестами **покрывается низкоуровневая атомарная логика** приложения;
- в классическом модульном тестировании все **100% модульных тестов должны проходить успешно**;
- модульные тесты (unit-test, юнит-тесты) должны быть **предельно компактными, независимыми и отделёнными от кода**.



Определения

Для запоминания. Модульные тесты:

- всегда проходят на 100%;
- компактны и не независимы;
- отделены от кода.



JUnit — один из самых известных фреймворков («наборов инструментальных средств») для модульного тестирования приложений, написанных на Java.



Поскольку наш курс не касается непосредственно программирования на Java, мы рассмотрим особенности JUnit безотносительно «программистских особенностей языка».

Итак...

JUnit: установка

Чтобы начать работать с JUnit, его нужно загрузить с сайта

www.junit.org



Хотя почти во всех современных IDE (включая **Eclipse**, который мы будем использовать), все необходимые библиотеки уже есть.

После этого следует лишь **создавать тесты...**

JUnit: тесты

Для указания, что **метод является модульным тестом** JUnit, используется аннотация **@Test**.

```
import org.junit.Test;
```

```
public void MyTest {
```

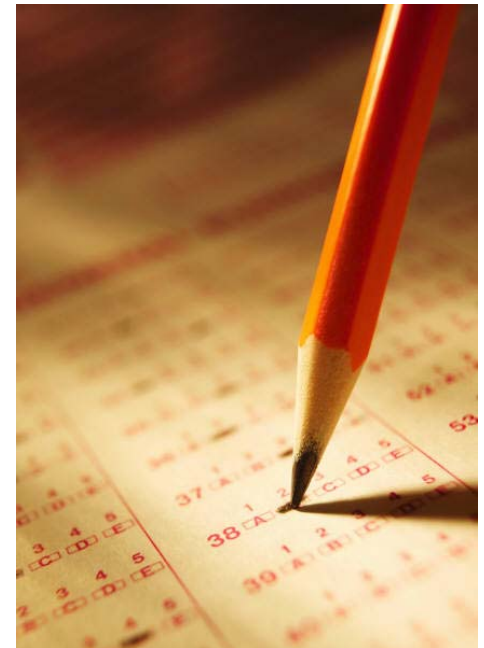
```
    @Test
```

```
    public void check_something() {
```

```
        ....
```

```
    }
```

```
}
```



JUnit: подготовка и завершение

Для указания, что метод должен выполняться перед каждым тестом или после каждого теста, используются аннотации **@Before** и **@After** соответственно.

```
public class MyTest {
```

```
    @Before
```

```
    public void prepareTestData() { ... }
```

```
    @Before
```

```
    public void setupConnection() { ... }
```

```
    @After
```

```
    public void freeConnection() { ... }
```

```
}
```

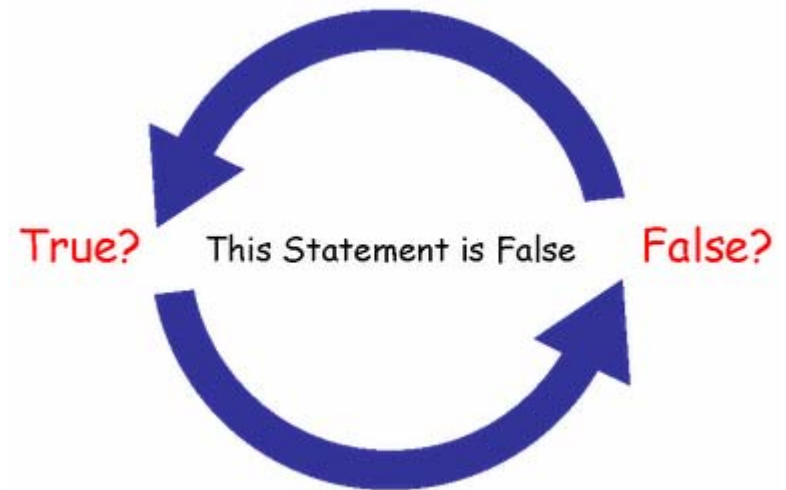


JUnit: проверки

Для **выполнения проверок** внутри тестов используются **методы assert***.

```
import org.junit.Test;  
import static org.junit.Assert.*;
```

```
public class MyTest {  
  
    @Test  
    public void test_something() {  
        ...  
        assertEquals( result, 3 );  
    }  
}
```



JUnit: проверки

Какие бывают **assert'ы**:

- `fail(String)` – приводит к «ошибке» теста.
- `assertTrue([message], boolean condition)` – проходит успешно, если аргумент равен true.
- `assertEquals([String message], expected, actual)` – проходит успешно, если аргументы равны.
- `assertEquals([String message], expected, actual, tolerance)` – проходит успешно, если аргументы равны (с указанием допустимой погрешности для дробных чисел).
- `assertNull([message], object)` – проходит успешно, если объект не создан (равен null).
- `assertNotNull([message], object)` – проходит успешно, если объект создан (НЕ равен null).
- `assertSame([String], expected, actual)` – проходит успешно, если обе переменные ссылаются на один и тот же объект.
- `assertNotSame([String], expected, actual)` – проходит успешно, если переменные ссылаются на РАЗНЫЕ объекты.

JUnit: исключения

Если мы ожидаем, что **тестируемый код может создать исключение**, мы **указываем его** (тогда тест пройдет нормально, иначе JUnit посчитает возникновение исключения ошибкой теста; но если исключение **НЕ** возникнет — это тоже будет расценено как ошибка теста):

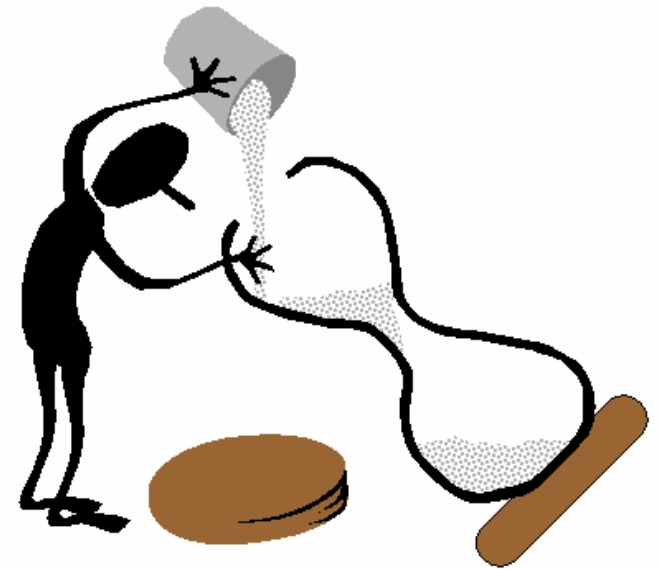
```
public class MyTest {  
  
    @Test(expected=OurException.class)  
    public void testException() {  
        openNonExistingFile(...);  
    }  
}
```



JUnit: тайм-аут

Если необходимо **ограничить время выполнения теста**, мы указываем **тайм-аут**, по истечении которого тест считается не пройденным:

```
@Test(timeout=5000)  
public void connectToNoWhere() {  
  
    ...  
  
}
```



JUnit: игнорирование тестов

Если некоторый тест нужно временно исключить из списка выполняемых, мы можем его игнорировать так:

```
public class MyTest {  
  
    @Ignore("Сырой тест")  
    @Test  
    public void increment() {  
        ...  
    }  
}
```



JUnit: сценарий

Организация тестов в сценарий (тест-сьют) происходит так:

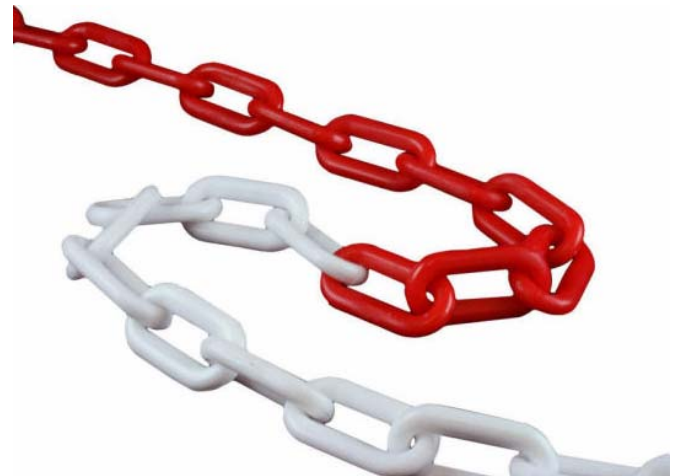
```
@RunWith(value=Suite.class)
```

```
@SuiteClasses(value={MyTest1.class, MyTest2.class})
```

```
public class SomeTests {
```

```
...
```

```
}
```



JUnit: тесты

Параметризация тестов (выполнение с некоторым набором параметров) происходит так:

@RunWith(value=Parameterized.class)

```
public class MyTest {  
    private int expected;  
    private int value;
```

@Parameters

```
public static Collection data() {  
    return Arrays.asList( new Object[][] {  
        { 3, 2 }, // expected, value  
        { 2, 3 }  
    });  
}
```

```
public MyTest (int expected, int value) {  
    this.expected = expected;  
    this.value = value;  
}
```

@Test

```
public void increment() {  
    assertEquals(expected, ClassToTest.increment(value));  
}  
}
```



JUnit: пример

Итак, допустим, у нас есть «примитивный калькулятор», который умеет выполнять **простейшие арифметические действия**. На самом верху иерархии есть **такой класс**:

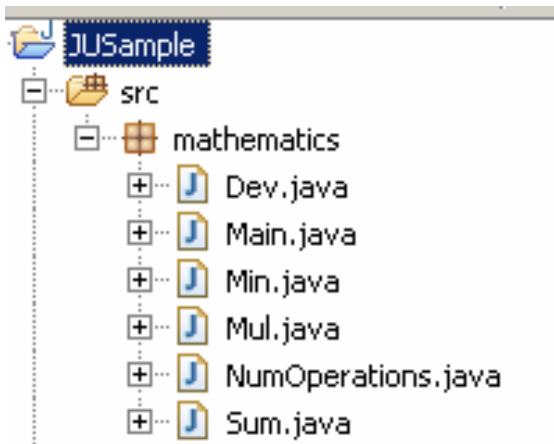
```
package mathematics;  
public abstract class NumOperations {  
protected Number a, b;  
abstract Number performOperation ();
```

```
    public void setA(Number A)  
    {  
        this.a = A;  
    }  
    public void setB(Number B)  
    {  
        this.b = B;  
    }  
    public Number getA()  
    {  
        return this.a;  
    }  
    public Number getB()  
    {  
        return this.b;  
    }  
}
```

JUnit: пример

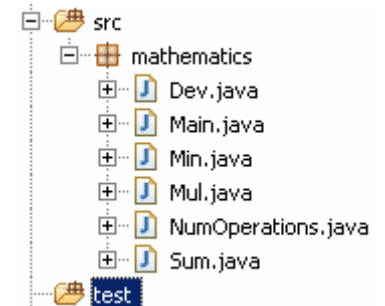
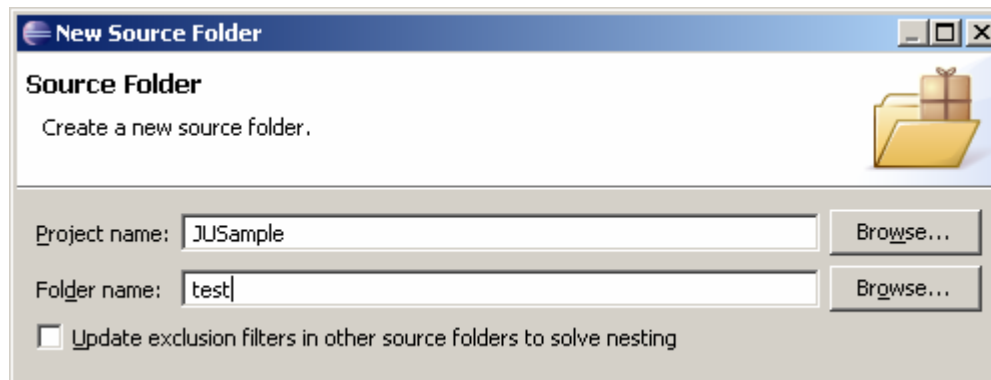
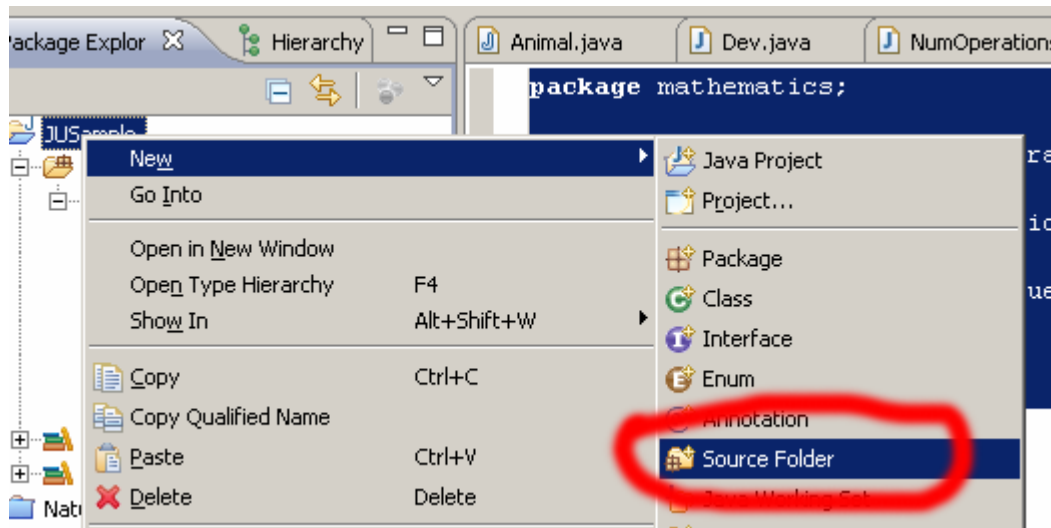
Затем у нас появляются **дочерние классы**, например:

```
package mathematics;  
public class Sum extends NumOperations{  
  
    public Number performOperation()  
    {  
        return this.a.doubleValue()+this.b.doubleValue();  
    }  
  
}
```



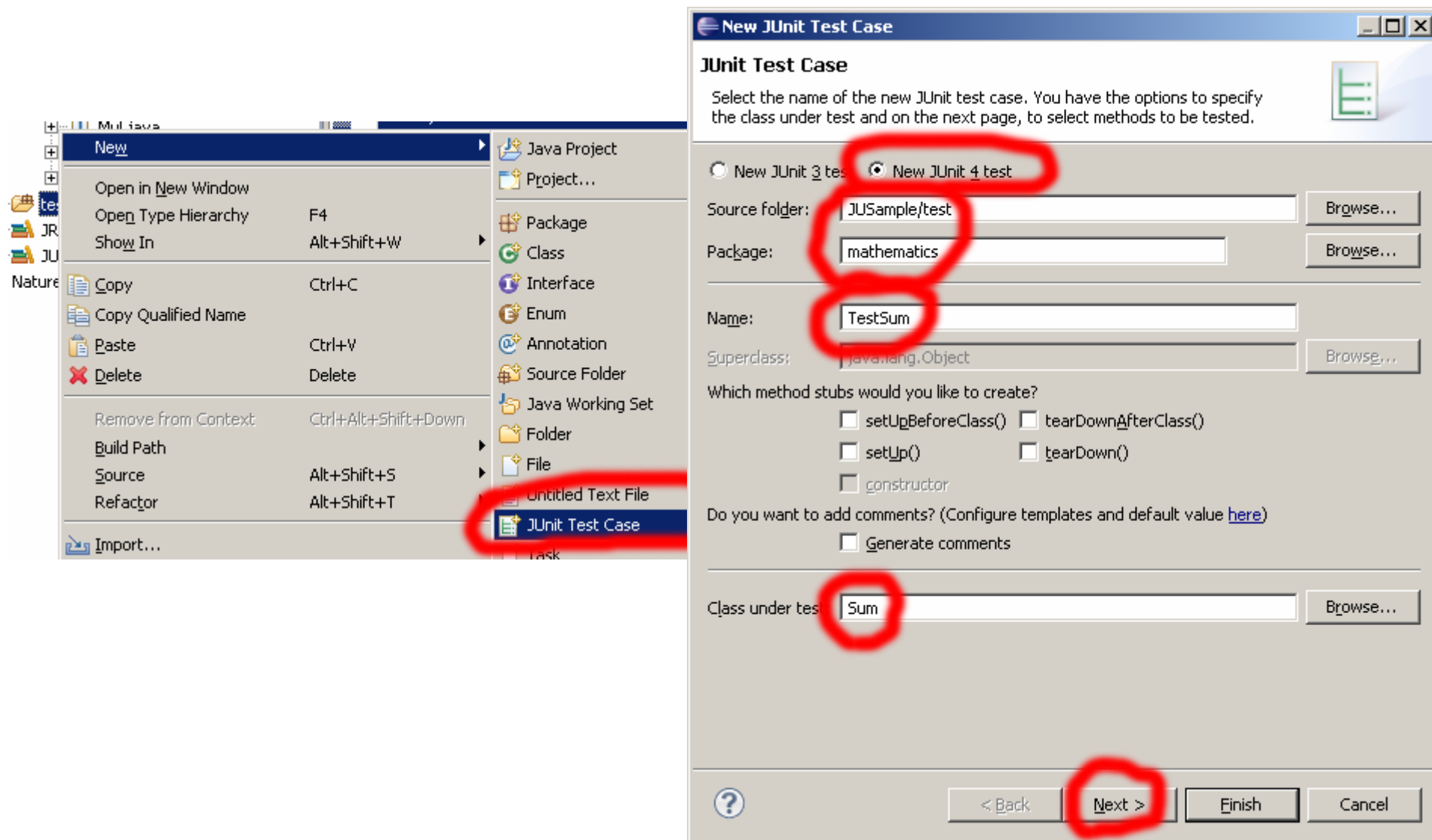
JUnit: пример

Нам нужно написать тесты, но мы помним, что их нужно размещать **отдельно от кода**, потому создадим отдельную папку («**source folder**»):



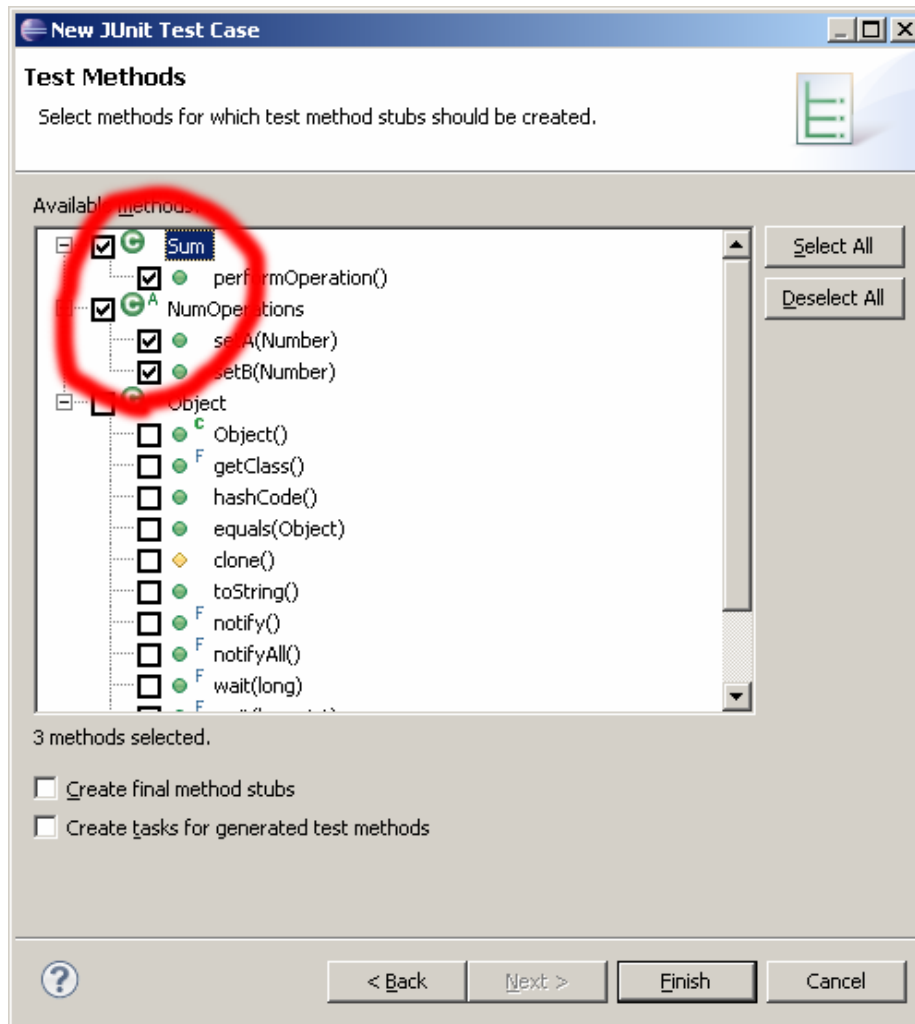
JUnit: пример

Теперь можно **создавать тест**:



JUnit: пример

Eclipse даже предлагает сразу отметить методы, которые мы собираемся тестировать:



JUnit: пример

Итак, мы получили такой автоматически сгенерированный код:

```
package mathematics;
import static org.junit.Assert.*;
import org.junit.Test;

public class TestSum {

    @Test
    public void testPerformOperation() {
        fail("Not yet implemented");
    }

    @Test
    public void testSetA() {
        fail("Not yet implemented");
    }

    @Test
    public void testSetB() {
        fail("Not yet implemented");
    }
}
```

JUnit: пример

После некоторых доработок получится:

```
package mathematics;
import static org.junit.Assert.*;
import java.util.Random;
import org.junit.Test;

public class TestSum {
    private static Sum sum = new Sum();
    private static Number a, b, c;

    @Test
    public void nat_prepareData() {
        Random r = new Random();
        TestSum.a = r.nextInt(1000000);
        TestSum.b = r.nextInt(1000000);
        TestSum.c = TestSum.a.doubleValue() + TestSum.b.doubleValue();
    }

    @Test
    public void testSetA() {
        TestSum.sum.setA(TestSum.a);
        assertEquals(TestSum.sum.getA(), TestSum.a);
    }

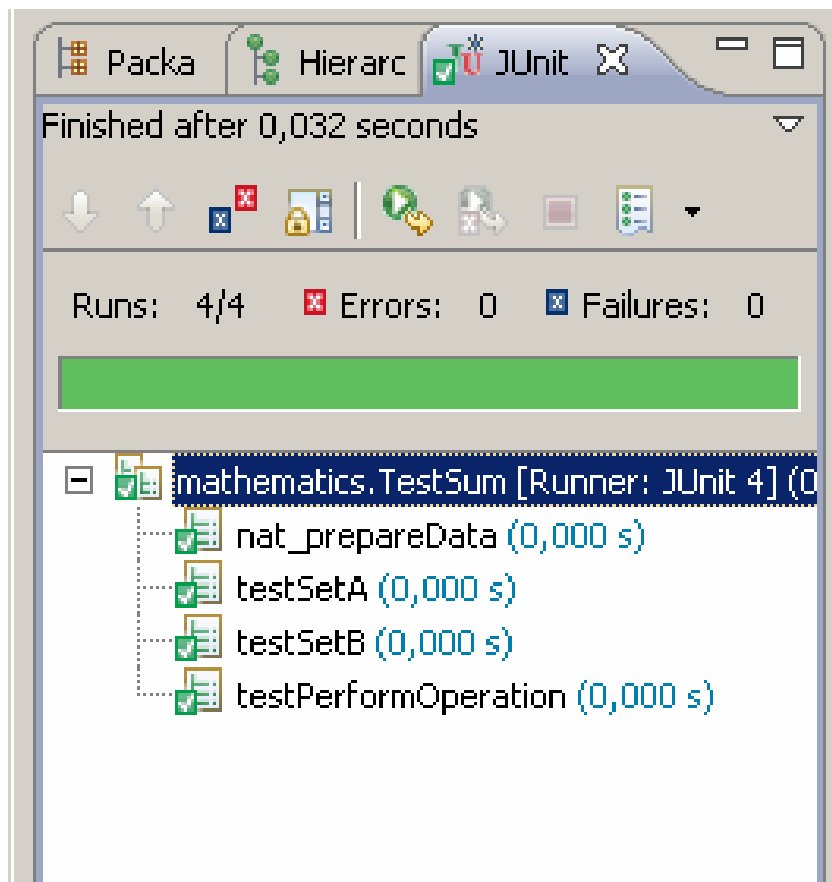
    @Test
    public void testSetB() {
        TestSum.sum.setB(TestSum.b);
        assertEquals(TestSum.sum.getB(), TestSum.b);
    }

    @Test
    public void testPerformOperation() {
        assertTrue(TestSum.sum.performOperation().doubleValue() == TestSum.c.doubleValue());
    }
}
```

JUnit: пример

Остаётся лишь запустить тест и наслаждаться результатом :)

(См. код этого примера в папке «Примеры/JUSample»)



Зачем всё это было нужно

Мы познакомились с идеей разработки модульных тестов потом, что тесты для **Selenium RC** на Java (а поддерживаются и другие языки) пишутся с использованием **JUnit**.

Важное замечание: **тесты с применением Selenium RC и HtmlUnit сами по себе могут НЕ ЯВЛЯТЬСЯ «модульными тестами» в классическом понимании этого термина**, т.к. они:

- НЕ работают напрямую с кодом приложения;
- могут проверять **ВЫСОКОУРОВНЕВУЮ** логику;
- могут НЕ проходить на 100%.



Однако модульное тестирование и JUnit являются технологической базой, на которой работает Selenium RC и HtmlUnit.

JUnit и Selenium RC

Зачем всё это было нужно

В завершении темы про Selenium IDE мы получили файл, **на основе которого создадим свои тесты:**

```
package com.example.tests;

import com.thoughtworks.selenium.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.regex.Pattern;

public class Jtc1 extends SeleneseTestCase {
    @Before
    public void setUp() throws Exception {
        selenium = new DefaultSelenium("localhost", 4444, "chrome", "http://tut.by/");
        selenium.start();
    }

    @Test
    public void testJtc1() throws Exception {
        selenium.open("/");
        selenium.type("str", "EPAM");
        selenium.click("//input[@name='search']");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("EPAM Systems"));
    }

    @After
    public void tearDown() throws Exception {
        selenium.stop();
    }
}
```

Подготовка к работе с Selenium RC

Но прежде, чем создавать и выполнять тесты, нужно **подготовить своё «рабочее место»**.

Загружаем **Selenium Remote Control** отсюда:

<http://seleniumhq.org/download/>

Из полученного архива распаковываем папки

selenium-server-НОМЕРВЕРСИИ

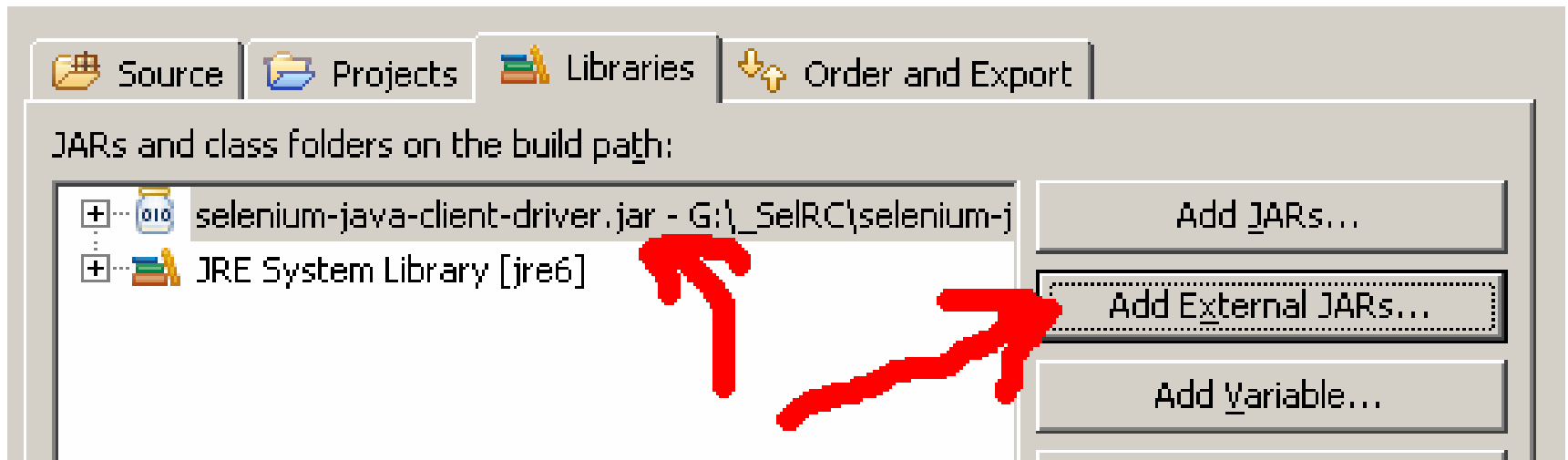
и

selenium-java-client-driver-НОМЕРВЕРСИИ



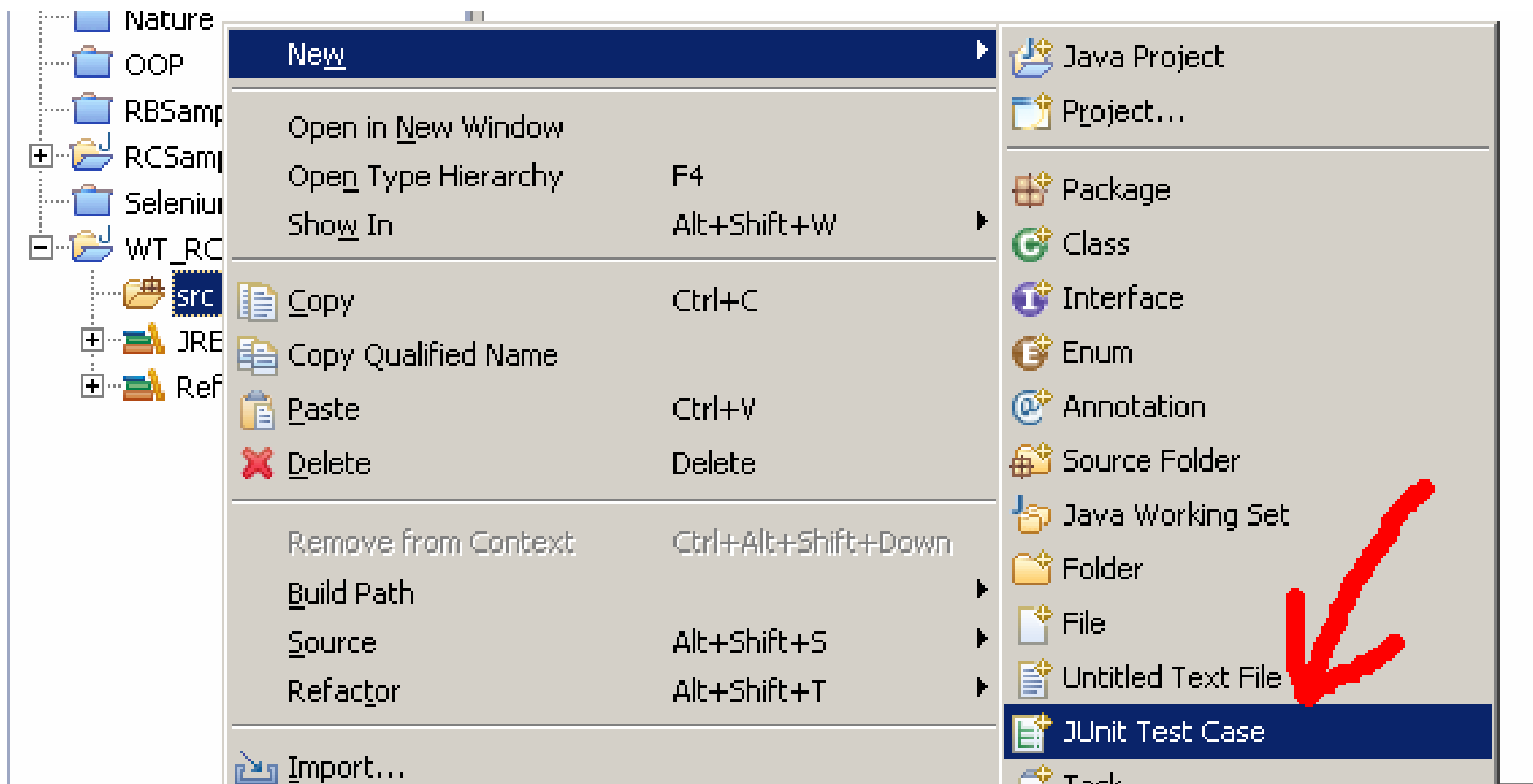
Подготовка к работе с Selenium RC

Создаём в **Eclipse** новый проект, куда подключаем библиотеку **selenium-java-client-driver**



Подготовка к работе с Selenium RC

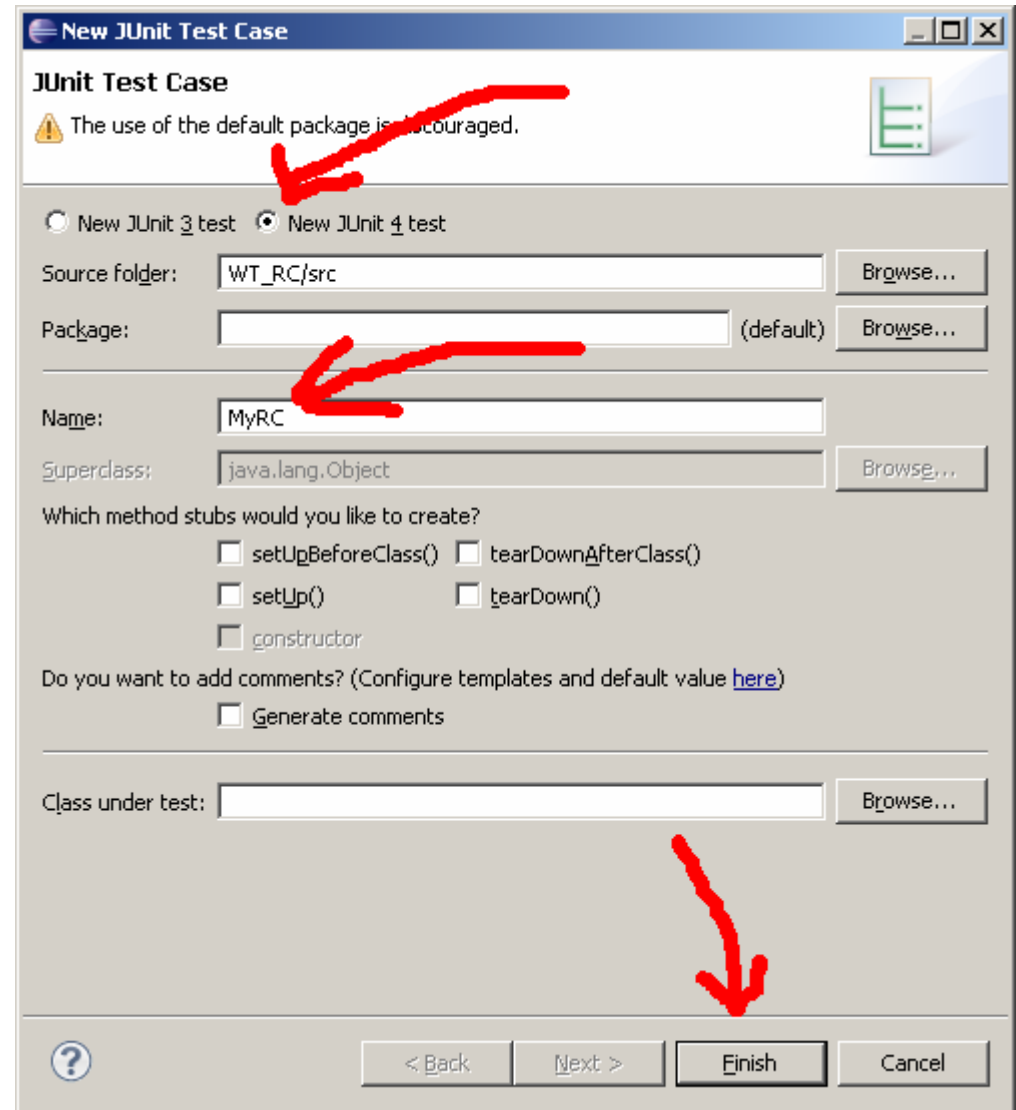
Создаём новый **JUnit** тест-кейс:



Подготовка к работе с Selenium RC

Важно выбрать 4-ю версию JUnit!

Указываем имя класса и завершаем создание тест-кейса.



Подготовка к работе с Selenium RC

В исходный код сначала пишем:

```
import static org.junit.Assert.*;
import com.thoughtworks.selenium.DefaultSelenium;
import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class MyRC {

    private DefaultSelenium selenium = null;
    private String base_url = "http://tut.by";

    @Before
    public void beforeClass()
    {
        selenium = new DefaultSelenium("127.0.0.1", 4444, "*firefox", base_url);
        selenium.start();
    }

    @After
    public void afterClass()
    {
        selenium.close();
        selenium.stop();
    }

}
```

Подготовка к работе с Selenium RC

А потом просто копируем и вставляем перед закрывающей скобкой описания MyRC тест, который сгенерировал для нас Selenium IDE:

```
@Test
public void testJtc1() throws Exception {
    selenium.open("/");
    selenium.type("str", "EPAM");
    selenium.click("//input[@name='search']");
    selenium.waitForPageToLoad("30000");
    assertTrue(selenium.isTextPresent("EPAM Systems"));
}
```

Подготовка к работе с Selenium RC

В итоге получится:

```
import static org.junit.Assert.*;
import static org.junit.Verify.*;
import com.thoughtworks.selenium.DefaultSelenium;
import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class MyRC {

    private DefaultSelenium selenium = null;
    private String base_url = "http://tut.by";

    @Before
    public void beforeClass()
    {
        selenium = new DefaultSelenium("127.0.0.1", 4444, "**firefox", base_url);
        selenium.start();
    }

    @After
    public void afterClass()
    {
        selenium.close();
        selenium.stop();
    }

    @Test
    public void testJtc1() throws Exception {
        selenium.open("/");
        selenium.type("str", "EPAM");
        selenium.click("//input[@name='search']");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("EPAM Systems"));
    }
}
```



Что мы получили

Разберём получившийся скрипт построчно (или, точнее, поблочно). Итак.

Блок `import`'ов предназначен для подключения к нашему исходному коду внешних библиотек. В первой строке мы подключаем библиотеку `DefaultSelenium`, в остальных – библиотеки `JUnit`.

```
import com.thoughtworks.selenium.DefaultSelenium;  
import static org.junit.Assert.*;  
import static org.junit.Verify.*;  
import org.junit.Test;  
import org.junit.Before;  
import org.junit.After;
```

Что мы получили

Методы класса **MyRC** как раз и будут **тестами**, ради которых всё затевалось.

```
public class MyRC {
```

```
...
```

```
}
```

Что мы получили

Эти две строки отвечают за **объявление** и **инициализацию** экземпляра класса **DefaultSelenium** и строки, содержащей **URL** тестируемого сайта.

```
private DefaultSelenium selenium = null;  
private String base_url = "http://tut.by";
```

Что мы получили

Метод, помеченный аннотацией **@Before** выполняется перед каждым тестом и подготавливает для нас «чистую» копию объекта **DefaultSelenium**.

```
@Before
public void beforeClass()
{
    selenium = new DefaultSelenium("127.0.0.1", 4444, "*firefox",
                                   base_url);

    selenium.start();
}
```

Что мы получили

Метод, помеченный аннотацией **@After** выполняется после каждого теста и «убирает рабочее место».

```
@After  
public void afterClass()  
{  
    selenium.close();  
    selenium.stop();  
}
```

Что мы получили

А это — наш **тест**.

```
@Test
public void testJtc1() throws Exception {

    // открываем главную страницу сайта
    selenium.open("/");

    // Вводим в элемент с именем "str" слово "EPAM"
    selenium.type("str", "EPAM");

    // Кликаем по элементу с именем "search"
    selenium.click("//input[@name='search']");

    // ожидаем загрузки новой страницы
    selenium.waitForPageToLoad("30000");

    // проверяем, что на этой странице присутствует искомый текст
    assertTrue(selenium.isTextPresent("EPAM Systems"));
}
```

Подготовка браузера и сервера

Тест уже **готов к запуску**, но пока **не готово то, с помощью чего он будет запущен**.

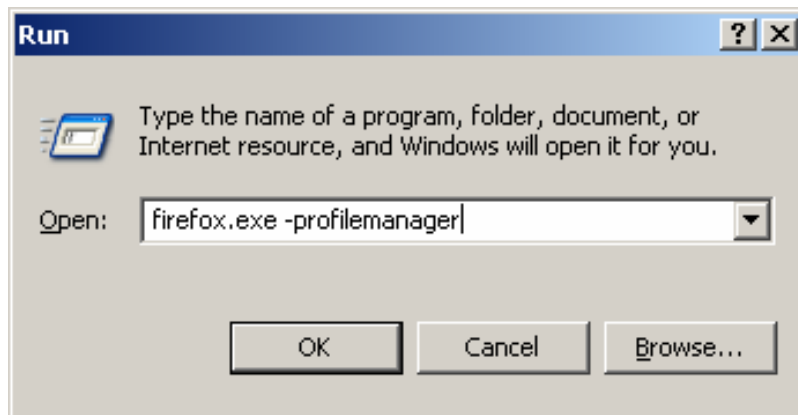
Нужно **настроить браузер и запустить сервер**.

Начнём с браузера.



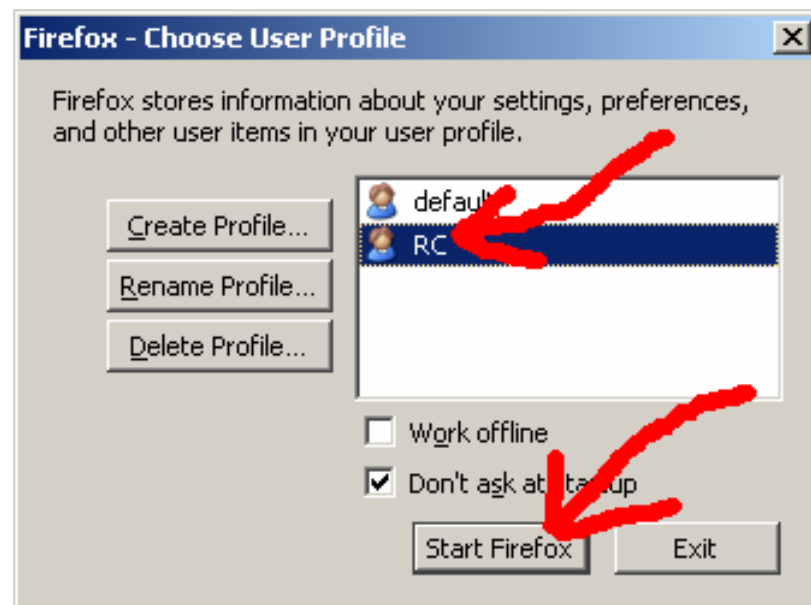
Настройка Firefox

1. Для начала – **закройте Firefox. ВСЕ ОКНА!** Если вы не уверены, что закрыли всё, проверьте наличие в системе процесса **firefox.exe** – его не должно быть.
2. Создайте **пустую папку** для хранения нового профиля Firefox. Например:
C:/MyFF
3. Из командной строки **выполните команду:**
firefox.exe –profilemanager



Настройка Firefox

4. Создайте **новый профиль**, назовите его, например **RC**
5. Укажите созданную папку (**C:/MyFF**) в качестве **места для хранения данных профиля**.
6. Выберите в **списке профилей** только что созданный и запустите Firefox.



Настройка Firefox

7. Удостоверьтесь, что с браузером всё в порядке :) . Если необходимо, **настройте прокси-сервер**.



8. Закройте Firefox и **УБЕДИТЕСЬ**, что он закрыт.

9. **Всё**, браузер настроен.

Запуск сервера Selenium RC

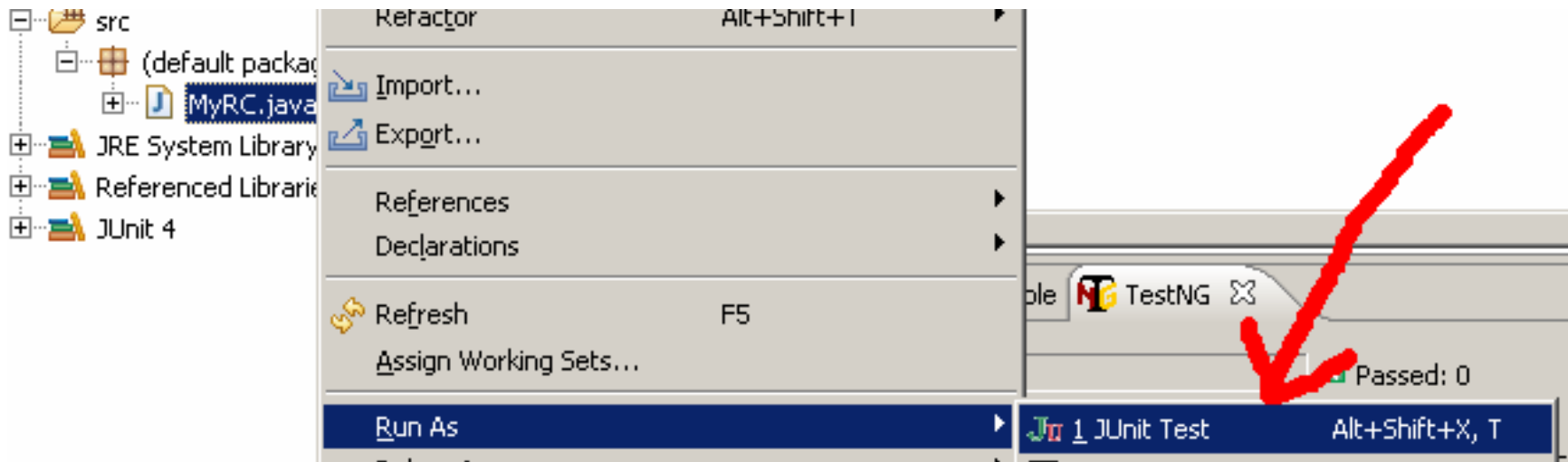
1. Перейдите в папку, в которую вы распаковали «selenium-server-НОМЕРВЕРСИИ».
2. Выполните из командной строки команду:
`java -jar selenium-server.jar -firefoxProfileTemplate "c:/MyFF"`

Вы должны увидеть примерно такое окно:

```
G:\_SelRC\selenium-server-1.0.3>java -jar selenium-server.jar -firefoxProfileTemplate "c:/MyFF"
18:48:47.218 INFO - Java: Sun Microsystems Inc. 16.3-b01
18:48:47.250 INFO - OS: Windows XP 5.1 x86
18:48:47.375 INFO - v2.0 [a2], with Core v2.0 [a2]
18:48:48.359 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
18:48:48.375 INFO - Version Jetty/5.1.x
18:48:48.375 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
18:48:48.421 INFO - Started HttpContext[/selenium-server,/selenium-server]
18:48:48.421 INFO - Started HttpContext[/,/]
18:48:48.890 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@1a7bf11
18:48:48.890 INFO - Started HttpContext[/wd,/wd]
18:48:48.968 INFO - Started SocketListener on 0.0.0.0:4444
18:48:48.968 INFO - Started org.openqa.jetty.jetty.Server@1ca318a
```

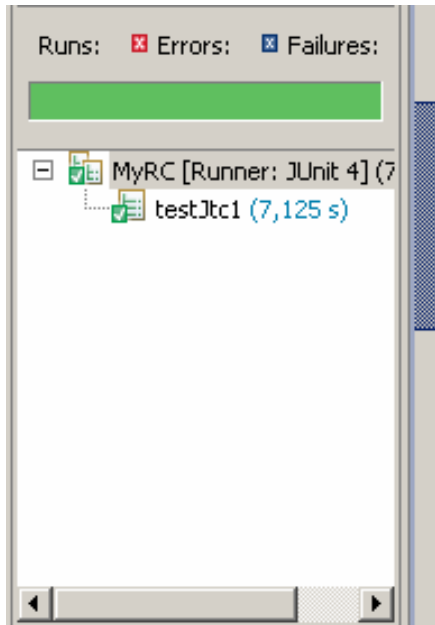
Долгожданный запуск теста

Теперь можно переключиться обратно в Eclipse и запустить тест. **Всё готового.**



Долгожданный запуск теста

Если вы всё сделали правильно, **вы увидите такое:**



Это значит, что **ваш тест прошёл успешно**, и теперь остаётся только потренироваться на практике писать более сложные тесты.

Расширение HtmlUnit

Краткое введение

Веб-ориентированные приложения можно тестировать и **без браузера**.

Или, по крайней мере, без «привычного обычным людям» браузера.

Мы уже знакомы с JUnit, поэтому нам легко будет понять, как работает **его расширение – HtmlUnit**.



Краткое введение

HtmlUnit — это специальное средство автоматизированного тестирования веб-ориентированных приложений. Фактически, это «конструктор для браузера» без графического интерфейса.

HtmlUnit «умеет» всё то же самое, что умеет браузер: открывать страницы, заполнять и отправлять формы, работать с куки и сессиями, выполнять JavaScript.

Единственное отличие от обычного браузера — это всё никак не отображается «на экране».

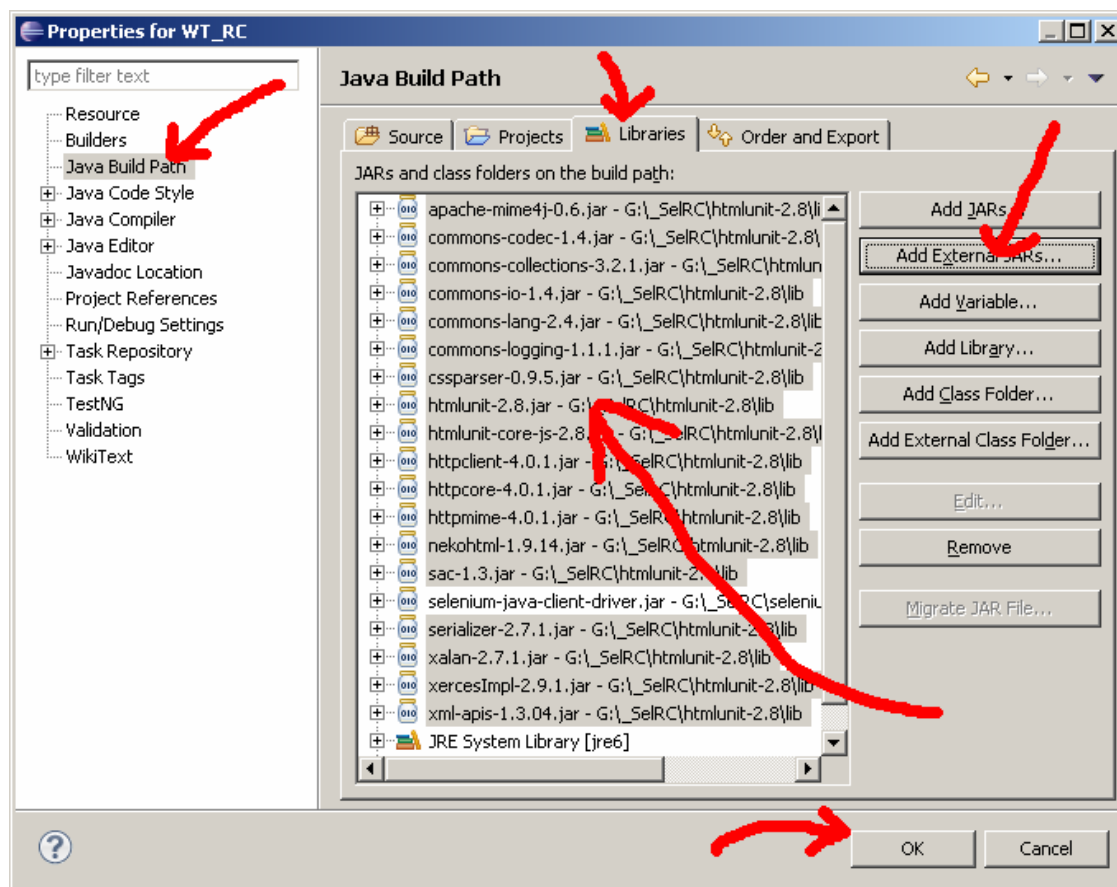


HtmlUnit – настройка

Загрузить последнюю версию HtmlUnit можно отсюда:

<http://htmlunit.sourceforge.net>

После распаковки архива добавьте все **jar-файлы** из папки **lib** в свой проект:



HtmlUnit – создание теста

Уже показанным ранее способом **создайте JUnit тест, внутри которого мы сейчас напишем код, выполняющий все те же действия, что и наш тест в Selenium RC.**

```
import static org.junit.Assert.*;

import org.junit.Test;

import com.gargoylesoftware.htmlunit.BrowserVersion;
import com.gargoylesoftware.htmlunit.DefaultCredentialsProvider;
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlForm;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import com.gargoylesoftware.htmlunit.html.HtmlSubmitInput;
import com.gargoylesoftware.htmlunit.html.HtmlTextInput;

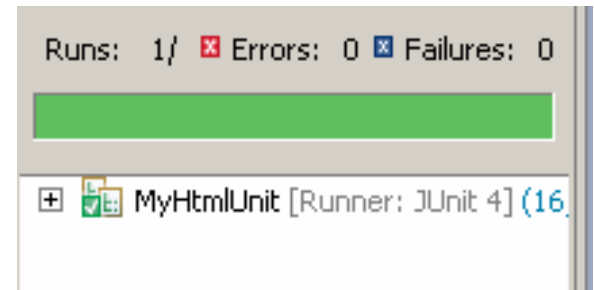
public class MyHtmlUnit {

    @Test
    public void testOne() throws Exception
    {
        WebClient webClient = new WebClient(BrowserVersion.FIREFOX_3, "proxy.com", 8080);
        DefaultCredentialsProvider credentialsProvider = (DefaultCredentialsProvider) webClient.getCredentialsProvider();
        credentialsProvider.addCredentials("username", "password");
        HtmlPage page = webClient.getPage("http://tut.by");
        HtmlForm form = page.getHtmlElementById("search", false);
        HtmlTextInput textField = form.getInputByName("str");
        HtmlSubmitInput button = form.getInputByName("search");
        textField.setValueAttribute("EPAM");

        page = button.click();

        String pageAsText = page.asText();

        assertTrue(pageAsText.contains("EPAM Systems"));
        webClient.closeAllWindows();
    }
}
```



HtmlUnit – создание теста

Уже показанным ранее способом **создайте JUnit тест, внутри которого мы сейчас напишем код, выполняющий все те же действия, что и наш тест в Selenium RC.**

```
import static org.junit.Assert.*;

import org.junit.Test;

import com.gargoylesoftware.htmlunit.BrowserVersion;
import com.gargoylesoftware.htmlunit.DefaultCredentialsProvider;
import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlForm;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import com.gargoylesoftware.htmlunit.html.HtmlSubmitInput;
import com.gargoylesoftware.htmlunit.html.HtmlTextInput;

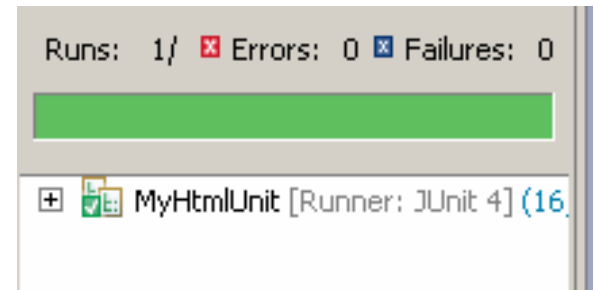
public class MyHtmlUnit {

    @Test
    public void testOne() throws Exception
    {
        WebClient webClient = new WebClient(BrowserVersion.FIREFOX_3, "proxy.com", 8080);
        DefaultCredentialsProvider credentialsProvider = (DefaultCredentialsProvider) webClient.getCredentialsProvider();
        credentialsProvider.addCredentials("username", "password");
        HtmlPage page = webClient.getPage("http://tut.by");
        HtmlForm form = page.getHtmlElementById("search", false);
        HtmlTextInput textField = form.getInputByName("str");
        HtmlSubmitInput button = form.getInputByName("search");
        textField.setValueAttribute("EPAM");

        page = button.click();

        String pageAsText = page.asText();

        assertTrue(pageAsText.contains("EPAM Systems"));
        webClient.closeAllWindows();
    }
}
```



HtmlUnit – комментарии

Итак, **что происходит** в тесте:

```
@Test
public void testOne() throws Exception
{
    // Создаём экземпляр веб-клиента
    WebClient webClient = new WebClient(BrowserVersion.FIREFOX_3, "proxy.com", 8080);

    // Создаём экземпляр класса, отвечающего за прокси-сервер (если прокси-сервер нужен)
    DefaultCredentialsProvider credentialsProvider = (DefaultCredentialsProvider)
webClient.getCredentialsProvider();

    // Задаём логин-пароль для прокси-сервера
    credentialsProvider.addCredentials("username", "password");

    // Получаем главную страницу сайта
    HtmlPage page = webClient.getPage("http://tut.by");

    // Получаем форму поиска
    HtmlForm form = page.getHtmlElementById("search", false);
```

HtmlUnit – комментарии

```
// Получаем поле ввода поисковой фразы
HtmlTextInput textField = form.getInputByName("str");

// Получаем кнопку отправки данных формы
HtmlSubmitInput button = form.getInputByName("search");

// Вводим текст для поиска в поле ввода
textField.setValueAttribute("EPAM");

// Отправляем данные формы и получаем новую страницу
page = button.click();

// Получаем текст страницы
String pageAsText = page.asText();

// Проверяем наличие в тексте страницы искомой фразы
assertTrue(pageAsText.contains("EPAM Systems"));

// Завершаем работу клиента
webClient.closeAllWindows();
}
}
```

Заключение

В материалах данной темы сознательно не приводится подробное описание всех возможностей особенностей рассмотренных инструментальных средств, т.к. **ЭТИ средства интенсивно развиваются, а документация по ним занимает сотни страниц.**

Более глубокое изучение крайне рекомендуется и будет реализовано:

- на **практике**;
- посредством просмотра **обучающих видеороликов** (см. папку «Видео»);
- вами лично в процессе самостоятельного обучения, ибо **только практикуясь лично можно достичь по-настоящему глубокого понимания темы.**

Тест для проверки изученного

1. Что такое «модульное тестирование»?
2. Каковы основные особенности модульных тестов?
3. Что такое JUnit? В чём отличие JUnit и HtmlUnit?
4. Что такое Selenium RC? Как он связан с JUnit?
5. В чём самое, на ваш взгляд, главное отличие HtmlUnit и Selenium RC?
6. Зачем в тестах Selenium RC и HtmlUnit перед каждым тестом создаётся «новый экземпляр браузера»? Можно ли использовать один, созданный перед началом всех тестов?
7. Какие тесты больше похожи на действия реального пользователя – в Selenium RC или в HtmlUnit?
8. В чём основной недостаток Selenium IDE в сравнении с Selenium RC?
9. Можно ли с помощью Selenium RC тестировать веб-ориентированное приложение, написанное без применения Java?
10. Если один и тот же тест написать с использованием Selenium RC и HtmlUnit – какой вариант, на ваш взгляд, выполнится быстрее? Почему?