Создание автоматизированных тестов

Краткое руководство

Система управления бизнес-процессами и эффективностью

Оглавление

Введение

Данная книга является кратким руководством по созданию автоматизированных тестов, построенных на базе языка программирования Java, библиотеке Selenide и фреймворке для автоматизации сборки проектов Apache Maven. Она предназначена для тех пользователей, которые самостоятельно осваивают автоматизированное тестирование.

Данный набор технологий был выбран по следующим причинам:

1. JAVA:
    a. один из самых популярных языков программирования;
    b. присутствует низкий порог вхождения;
    c. большая аудитория, следовательно, много готовых решений и документации.
2. Selenide:
    a. удобная, бесплатная библиотека, обертка для Selenium (самого распространённого инструмента для автоматизации действий веб-браузера);
    b. прочитав FAQ на официальном сайте, можно начать писать тесты уже через несколько часов;
    c. постоянно растущая аудитория в связи с популяризацией и простотой;
    d. лояльные разработчики.
3. Apache Maven:
    a. фреймворк для автоматизации сборки проектов;
    b. единственный инструмент, предоставляющий большое количество возможностей без дополнительных зависимостей.

Основная задача этого руководства – обозначить минимальный набор знаний, требующихся для начала написания простых автоматизированных тестов, а также научить пользователя использованию базовых возможностей популярных библиотек.

Текущее руководство предполагает, что пользователь уже владеет базовыми навыками языка программирования Java.

Данная книга является справочным руководством, в котором последовательно разбираются основные настройки и возможности популярных библиотек для автоматизированного тестирования. Таким образом, она позволит познакомиться и освоить основные приемы работы с библиотеками. Также в книге приведено множество примеров, практик, советов, следуя которым будет легче усваивать материал.

Глава 1. Структура HTML-документа

Ниже приведены основные термины и их определения, используемые в данной главе.

HTML (Hyper Text Markup Language, т.е. "язык разметки гипертекста") – это специальный язык разметки, который используется для создания так называемых веб-документов.

Гипертекст – это некоторый текст, содержащий ссылку на другой текст. В веб-сфере данные ссылки являются ссылками на другие веб-документы.

Язык разметки – это набор синтаксических правил, с помощью которых можно упорядоченно представить ту или иную информацию на экране. Важно отметить, что язык разметки не является языком программирования.

HTML-документ – это веб-документ, сформированный при помощи языка разметки HTML. По своей сути – это текстовый файл, в котором был использован язык разметки. Исходный код HTML-документа может быть просмотрен и отредактирован в любом текстовом редакторе (например, в Блокноте). Это одно из удобных свойств HTML-документа. Программа, позволяющая отображать HTML-документ на основе исходного кода разметки, называется **браузером**.

Основной составляющей языка разметки являются **теги** (**дескрипторы**). Дескрипторы с содержимым (или без него) также называют **элементами HTML-документа**.

1.1. Типы HTML-документов

В связи с развитием синтаксиса языка разметки HTML существует несколько популярных его спецификаций. Это, в свою очередь, ведет к тому, что разные спецификации могут быть по-разному восприняты браузером и, как следствие, это приведет к некорректному отображению веб-документа на экране.

Данные спецификации определяют тип документа. Как следствие, в зависимости от того, какая спецификация языка разметки используется в веб-документе, в начале исходного кода веб-страницы должен быть указан тип документа. Основными спецификациями HTML являются "HTML 5" и "HTML 4.01" (используется все реже). При использовании языка XHTML основными спецификациями являются "XHTML 1.1" и "XHTML 1.0". Также на синтаксические правила разных спецификаций накладываются различные ограничения.

Для явного указания типа документа используется тег <!DOCTYPE>. Этот тег указывается первой строкой исходного кода веб-документа.

Для того, чтобы использовать спецификацию HTML 5, необходимо в начале исходного кода веб-страницы добавить строку <!DOCTYPE html>.

Типичный пример исходного кода веб-страницы в таком случае выглядит так:

Example.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    Content of the test HTML 5 page
  </body>
</html>
```

Идея создания языка XHTML состояла в том, чтобы добавить в язык разметки жесткие правила (например, написание названий тегов в нижнем регистре, написание значений атрибутов тегов в кавычках и т.п.).

Спецификация XHTML 1.0 имеет разделение на виды синтаксиса: строгий (strict) и переходный (transitional). Это сделано для того, чтобы предоставить разработчику возможность плавно перейти к более строгому (strict) синтаксису XHMTL, используя переходный (transitional).

1.2. Основы DOM

DOM (Document Object Model, т.е. "объектная модель документа") – это некоторый программный интерфейс, с помощью которого скрипты (программы) могут взаимодействовать с содержимым документов, созданных при помощи языка разметки (HTML, XML, XHTML). Ниже рассмотрим HTML DOM.

HTML DOM – это некоторое формальное представление содержимого веб-страницы в виде дерева. Данное представление использует объекты (элементы), с которыми, в свою очередь, взаимодействует клиентский язык программирования (например, JavaScript). Составляющие объектной модели документа называются **вершинами**.

Например, сам веб-документ – это вершина, а все элементы документа – это дочерние вершины, которые имеют другие дочерние вершины и т.д. Таким образом, веб-документ может быть представлен в виде дерева. Это и есть HTML DOM.

Используя такую структуру, очень удобно влиять на элементы страницы посредством объектно-ориентированных языков программирования, в частности, JavaScript.

Рассмотрим в качестве примера следующий исходный код веб-страницы:

Example.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>My header</h1>
    <a href =”#”>My link</a>
  </body>
</html>
```

Соответствующая структура дерева в HTML DOM будет выглядеть следующим образом (Рис. 1):

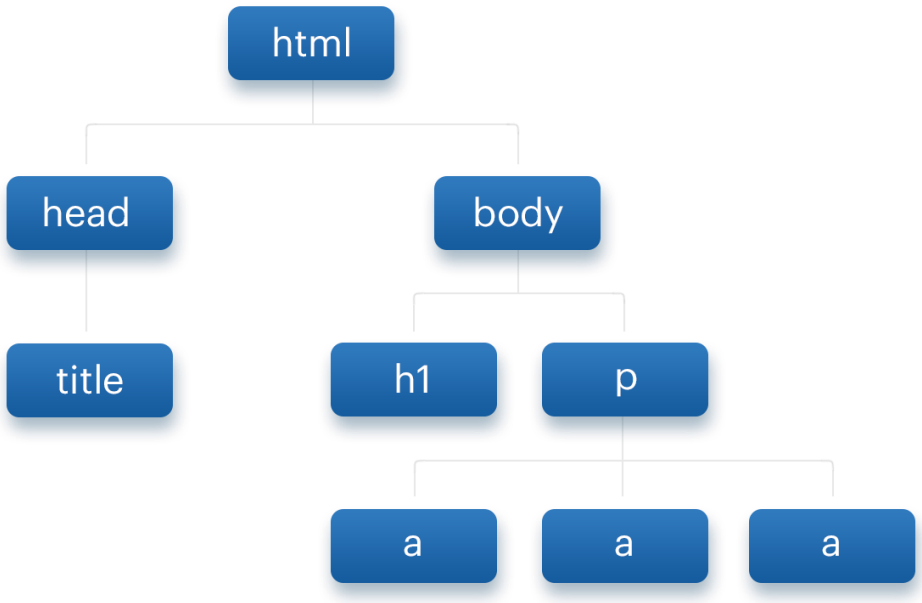

Рис. 1 Структура дерева HTML DOM

HTML DOM объекты состоят из:

- Свойств
- Методов доступа
- Событий

Свойство – это определенный параметр объекта, имеющий точное значение (например, атрибут тега).

Метод доступа – это специальная функция, обеспечивающая взаимодействие с объектом.

Событие – это любое "действие", которое может быть реализовано по отношению к странице (например, нажатие по элементу, перемещение указателя на элемент и т.д.).

1.3. HTML-теги

Тэг (дескриптор) HTML – это основная составляющая языка HTML. Тег представляет собой набор символов, помещенных в угловые скобки **<>**. Внутри данных скобок указывается название тега и дополнительные атрибуты с соответствующими значениями. Например:

Example.html

```
<div>
   <a href='/home'> Home </a>
</div>
```

Не обязательно всегда указывать атрибуты тега. Кроме того, не все атрибуты могут принимать значения. Например:

Example.html

```
<select name='country'>
    <option>USA</option>
    <option>Germany</option>
    <option selected>Ukraine</option>
</select>
```

Тут тег <select> имеет атрибут name со значением 'country'. Остальные теги либо не имеют атрибутов, либо атрибуты указаны без значения. Для ряда тегов присутствие некоторых атрибутов является обязательным, например, в теге ссылки <a> атрибут href является обязательным.

1.3.1. Классификация тегов

Теги делятся на два типа: **двойные (парные)** и **одиночные (непарные)**.

Двойные (парные) теги – это теги, для которых необходимо указывать открывающий и закрывающий тег. Закрывающий тег состоит из угловых скобок, названия тега и символа "/" перед названием тега. Например:

Example.html

```
<div>
   <b>Text</b>
</div>
```

Одиночные (непарные) теги – это теги, которые не требуется закрывать, т.е. у них нет закрывающего тега. Например:

Example.html

```
<img src='/img/picture.jpg' alt='My picture'>
<br />
```

В данном случае в теге был использован атрибут, который определяет расположение необходимого файла изображения.

В общем и целом, синтаксис тегов можно изобразить следующим образом:

Example.html

```
<tagName attr_1='val_1' ... attr_N='val_N'> [content] </tagName>
```

Важно отметить, что внутри двойных тегов (вместо [content]) возможно вложение одних тегов в другие. При этом необходимо следовать некоторым простым правилам.

При некорректной расстановке тегов или других синтаксических ошибках браузер также предпримет попытку отобразить страницу корректно – это предусмотрено программной частью браузера.

Пример исходного кода:

Example.html

```
<!DOCTYPE html>
<html>
<head>
   <title>My First Home Page</title>
 </head>
<body>
   <h1>Hello, page! </h1>
   This is my first web page.
   And this is my first link: <a href='http://google.com'>google</a>
</body>
</html>
```

Первый тег в данном исходном коде обозначает тип документа. В данном случае это HTML 5.

Следующие за ним теги <html>, <head>, <body> называются **тегами верхнего уровня**. Они определяют основную структуру HTML-кода.

Тег <html> обозначает начало кода с разметкой. Далее структура HTML-кода условно имитирует составляющие живых существ – <head> ("голова") и <body> ("тело").

В "голове" <head> указывается информация служебного и вспомогательного характера. Например, заголовок окна браузера в качестве заголовка страницы, кодировка, описание страницы, дополнительная информация для поисковых систем, браузеров и т.п. То есть, в этой части HTML-документа указывают данные, которые явно не будут отображены на странице.

В "теле" <body> указывается только та информация, которая должна быть явно отображена на странице – текст, картинки, ссылки и другие элементы.

По характеру представления на экране теги условно делятся на следующие категории:

- форматирование текста
- ссылки
- изображения
- таблицы
- списки
- формы
- фреймы

Также теги можно разделить на блочные (block) и строковые (inline). Основными блочными тегами являются: , <p>, <h1>. Основными строчными тегами являются: , <a>, <b>, <u>, <i>.

Внимание! Для корректного отображения страницы на экране не следует помещать блочные теги в строковые.

1.4. HTML-таблицы

В языке HTML предусмотрена возможность создания таблиц. Таблицы в HTML синтаксически являются довольно громоздкими конструкциями, потому следует внимательно относиться к их созданию.

Для создания таблицы в HTML используется минимум три тега: <table>, <td> и <tr>.

Тег **<table>** используется для создания самого контейнера для будущей таблицы. Фактически, это рамка. Данный тег может принимать ряд необязательных атрибутов, рассмотренных ниже.

Тег **<tr>** используется для создания строки таблицы. Данный тег не может принимать какие-либо атрибуты.

Тег **<td>** используется для создания ячейки таблицы внутри уже существующей строки. Данный тег может принимать ряд необязательных атрибутов.

Ниже представлен простой пример. Таблица 2x2, в ячейках которой записаны цифры 1, 2, 3, 4.

Example.html

```
<table>
  <tr>
    <td>1</td>
    <td>2</td>
  </tr>
  <tr>
    <td>3</td>
    <td>4</td>
  </tr>
</table>
```

1.5. Формы

Для взаимодействия пользователя с веб-сервером в HTML предусмотрены **формы**. Формы позволяют пользователю вводить необходимую информацию и получать результат после обработки внесенных данных веб-сервером. Например, поиск, авторизация и т.п.

Именно с помощью форм возможна отправка запросов по протоколу HTTP методом POST (при переходе по обычной гиперссылке используется метод GET). Данные передаются на сервер в виде пар name-value, которые формируются для каждого поля формы.

Создать форму на веб-странице можно с помощью тега <form>. Сам по себе данный тег является блочным элементом веб-страницы. Этот тег имеет несколько важных, но необязательных атрибутов.

В атрибуте action можно указать адрес скрипта, который будет обрабатывать отправленные данные. Адрес может быть задан в виде абсолютного или относительного пути. Если этот атрибут не задан, то по умолчанию форма отправит данные по адресу текущего веб-документа.

В атрибуте method можно указать метод передачи данных согласно протоколу HTTP. Значениями данного атрибута могут быть GET или POST. По умолчанию, если этот атрибут не указан или значение не совпадает с двумя вышеуказанными методами, будет использован GET-метод. Ниже представлены несколько примеров, которые иллюстрируют использование этих атрибутов.

Example.html

```
<form> ... </form>
<form method='post'> ... </form>
<form action='/contact.php' method='post'> ... </form>
```

Простое текстовое поле создается при помощи тега <input> и обязательного атрибута type со значением text. Например:

Example.html

```
<input type='text' name='login'>
```

В тегах элементов формы также необходимо указать атрибут name, который позволит веб-серверу различать данные с разных элементов форм. Значение этого атрибута может быть произвольным, но, желательно, чтобы оно было логически связано с предназначением поля.

Для создания кнопки отправки данных необходимо создать аналогичный тег, но со значением submit атрибута type. Например:

Example.html

```
<input type='submit'>
```

Существуют также несколько вариаций простого текстового поля, например, поле для пароля (type='password') или поле для электронного адреса (type='email'). Поле для пароля идентично простому текстовому полю, однако для безопасности, увидеть набранный в нем текст нельзя. Поле для электронного адреса также идентично простому текстовому полю, но имеет полезное свойство – браузеры автоматически проверяют корректность введенного в него электронного адреса. Данная возможность добавлена в спецификации HTML 5.

Ниже представлена несложная форма для регистрации пользователя. Она содержит одно простое текстовое поле, поле для электронного адреса и два поля для пароля:

Example.html

```
<form>
   Enter login: <input type='text' name='login'/> <br />
   Enter email: <input type='email' name='email' /> <br />
   Enter password: <input type='password' name='password' /> <br />
   Confirm password: <input type='password' name='password_repeat' /> <br />
   <input type='submit' />
</form>
```

Также в тегах input можно разместить следующие атрибуты:

- **id** – должен иметь уникальное значение в рамках веб-страницы. Используется скриптами для однозначного нахождения конкретного элемента в объектной модели документа;
- **placeholder** – шаблонный текст-подсказка в поле, который исчезает при введении данных;
- **value** – значение поля по умолчанию. Для кнопки отправки данных – это надпись на кнопке.

Глава 2. Локаторы

2.1. XPath-локаторы

XPath (XML Path Language) – язык запросов к элементам XML или XHTML-документа. XML имеет древовидную структуру. В документе всегда имеется корневой элемент. У элемента дерева всегда существуют предки (исключение – корневой элемент) и могут существовать потомки. Каждый элемент дерева находится на определенном уровне вложенности. У элементов на одном уровне бывают предыдущие и следующие за ним элементы.

Строка XPath – это фактический путь к элементу в дереве, где каждый уровень разделяется "/". В результате обработки выражения XPath получается объект, который может быть:

- набором узлов (node-set) – неупорядоченный набор узлов без дубликатов;
- булевым значением (boolean) – true или false;
- числом (number) – число с плавающей точкой;
- строкой (string) – последовательность UCS символов.

Например, HTML-документ:

Example.html

```
<html>
<body>
<div id="first">
   <span>span внутри div</span>
</div>
<div id="second"></div>
<span>некоторый текст</span>
</body>
</html>
```

Результатом выполнения XPath-запроса "/html/body/*/span" будет узел . Запрос "/html/body/div" вернет несколько элементов (div#first и div#second).

Если в запросе первым символом стоит "/", то путь адресации считается абсолютным (от корня документа). Корень документа всегда является контекстом по умолчанию.

Контекст – это текущий полученный узел или набор узлов, относительно которых рассчитывается следующий шаг.

2.1.1. Оси

Оси – это основа запросов XPath и их обязательная часть:

- **ancestor::** – возвращает множество предков;
- **ancestor-or-self::** – возвращает множество предков и текущий элемент;
- **attribute::** – возвращает множество атрибутов текущего элемента;
- **child::** – возвращает множество потомков на один уровень ниже;
- **descendant::** – возвращает полное множество потомков;
- **descendant-or-self::** – возвращает полное множество потомков и текущий элемент;
- **following::** – возвращает необработанное множество элементов ниже текущего;
- **following-sibling::** – возвращает множество элементов на том же уровне, следующих за текущим;
- **namespace::** – возвращает множество, имеющее пространство имён (присутствует атрибут xmlns);
- **parent::** – возвращает предка на один уровень назад;
- **preceding::** – возвращает множество обработанных элементов, исключая множество предков;
- **preceding-sibling::** – возвращает множество элементов на том же уровне, предшествующих текущему;
- **self::** – возвращает текущий элемент.

Для наиболее часто используемых осей существуют сокращения:

- **attribute::** – можно заменить на "@";
- **child::** – часто просто опускают;
- **descendant::** – можно заменить на ".//";
- **parent::** – можно заменить на "..";
- **self::** – можно заменить на ".".

Для приведенного выше примера "/html/body/*/span”", полный синтаксис будет иметь вид: **"/child::html/child::body/child::*/child::span"**.

Чаще XPath-запрос начинают с ".//" или "//", это делает путь к элементу относительным. Символы ".//" в начале запроса возвращают полное множество потомков, которые являются дочерними для корня документа, т.е. все элементы на текущей странице. В данном случае точку в начале запроса можно опустить, потому что корневой элемент уже является контекстом.

Приведенный выше пример можно изменить следующим образом: "//div/span".

В то время, как XPath-запрос "//span" вернет оба элемента span.

Важную роль в построение запросов играют **предикаты** – это необязательная часть, заключаемая в квадратные скобки, в которой могут содержаться оси, условия проверки, функции и операторы. В ходе обработки предиката из полученного на шаге набора узлов исключаются узлы, не прошедшие условия проверки. Например: запрос, необходимый для поиска элемента span, находящийся внутри любого элемента с id = "first", будет выглядеть так: **"//*[@id="first"]/span"**.

2.1.2. Правила использования XPath

Ниже приведены основные правила, которых следует придерживаться при использовании XPath:

1. Запрещается использовать плагины или копирование XPath из кода страницы средствами браузера или веб-разработчика. Например, плагин к Firefox отображает одну ссылку следующим образом: **//header/div/ul/li[2]/a**. Такая ссылка не содержит полезной и понятной пользователю информации.
2. Xpath должен быть как можно короче и понятнее. Приветствуется использование его схожести с языком программирования для составления простых и информативных выражений.
3. Вместо длинной цепочки слешей ("/"), необходимо использовать отношения элементов: предок, потомок, сестринский элемент.
4. Обязательно использование логических операций and, or, not.
5. Начинать XPath нужно всегда с "//".
6. Не следует использовать фильтры с номером элемента ([2]).

2.1.2.1. Рекомендуемые к использованию команды

Далее указаны команды, которые необходимо использовать для написания грамотных и удобных локаторов.

1. **text**() – возвращает текст, содержащийся в элементе. Данная команда является непопулярной, однако, в любом веб-приложении используются элементы, содержащие текст. Значения полей id и class у этих элементов могут меняться. При этом в большинстве случаев текст остаётся неизменным, это не потребует правок локаторов, основанных на тексте.

Внимание! Не применять поиск по тексту, если приложение является мульти язычным. При смене языка, локаторы перестанут работать.

Example.html

```
<a id=”__vy85” class=”sbtn sbtn-grraised sbtn-large”>Contact us</a>
```

Как указано в примере кода выше, id явно сгенерирован и к нему нельзя привязаться. Поле class не может быть использовано, так как Selenium не разрешает использовать сложносоставные имена в локаторе className. В итоге, текст является единственным решением проблемы: **//a[text()=’Contact us’]**

2. **contains(*параметр, искомое*)** – возвращает элемент, если он содержит искомое. Данную команду можно использовать в связке с командой text(),

если необходимо составить сложный локатор и точный текст неизвестен. Например: **//div[@class='buttons' and contains(text(),'Save')]**

Результатом работы вышеуказанного примера будет кнопка, содержащая текст "Save". Если в тестируемом веб-приложении существует несколько страниц, содержащих кнопку сохранения, но выполняющих разную работу и, соответственно, содержащих разный текст (Сохранить файл, Сохранить диаграмму, Сохранить отчёт и т.д.), данный локатор выдаст их все.

Ниже представлен пример кода:

Example.html

```
<div id="__v22469" class="btn btn-grflat intercomBtn btn-img" enabled="enabled">
  <div class="selector-node"></div>
```

Полезной функцией является поиск по отдельным словам, содержащимся в поле. Например, для поиска данного элемента можно использовать следующую строку: **//div[contains(@class,'intercomBtn')]**

В данном примере поиск осуществляется по уникальной последовательности символов.

3. **starts-with (параметр, искомое)** – возвращает элементы, поля которых начинаются с искомой последовательности символов. Можно переписать рассмотренный выше пример с кнопками "Сохранить" с использованием данной функции: **//div[@class='buttons' and starts-with(text(),'Save')]**

В данном примере результат будет аналогичен результату, полученному при использовании функции contains().

Далее представлены команды, основанные на отношениях элементов (предок, родитель, ребёнок, потомок, сестринский элемент). При грамотном использовании все они позволяют осуществлять очень гибкий и функциональный поиск.

Формат использования: **//начальный элемент/отношение::тег(фильтр) конечного элемента.**

1. **sibling** – возвращает сестринский элемент – элемент, который расположен на том же уровне, что и начальный (не потомок и не предок). Выделяют 2 типа:
   a. **preceding-sibling** – сестринский элемент, который расположен до (выше) указанного;

b. **following-sibling** – сестринский элемент, расположенный после (ниже) указанного.

Например:

Example.html

```
<div id="__vz4019" class="input has-floating-label" enabled="enabled">
  <input id="__vx4019_input" class="nativeinput" type="text" value="">
  <div class="floating-label input-floatinglabel-text">Тема</div>
  <div class="hint-label"></div>
   <hr class="primary-backgroundColor">
```

Необходимо ввести текст в input, но в данном случае имеется ряд проблем: id динамический, несколько классов и сгенерированных id со словом "input" на странице. Единственным выходом будет использовать элемент с уникальным для страницы текстом: **//div[text()='Тема']/preceding-sibling::input**

В данном случае сначала ищется уникальный элемент с текстом, а затем от него находится предшествующий сестринский элемент. Для нахождения сестринского элемента используется фильтр-уточнение. Например:

Example.html

```
<div id="__we1848" class="btn btn-grflat listViewMoreActionsButton btn-img" enabled="enabled">
<div id="__we1870" class="btn btn-grflat btn-img" enabled="enabled">
  <div class="selector-node"></div>
```

Необходимо кликнуть кнопку, которая не содержит текста, но содержит иконку. В данном примере можно использовать элемент выше, который содержит уникальное название класса: **//div[contains(@class,'listViewMoreActionsButton')]/following-sibling::div**

В первую очередь необходимо найти элемент, у которого есть уникальное слово в названии класса и после этого можно найти следующий сестринский элемент с тегом div. При условии, если последующих сестринских элементов с тегом div будет несколько, вернется только самый первый.

2. **parent** и **child** – родитель и наследник (ребенок). Речь идет о непосредственном **прямом** родителе или наследнике, а не о предке или потомке.

Ранее рассмотренный пример:

Example.html

```
<div id=”__vx4019” class=”input has-floating-label” enabled=”enabled”>
   <input id=”__vx4019_input” class=”nativeinput” type=”text” value=””>
   <div class=”floating-label input-floatinglabel-text”>Тема</div>
   <div class=”hint-label”></div>
    <hr class=”primary-backgroundColor”>
```

Допустим, что необходимо найти непосредственно элемент с id=”__vz4019”. Для всех элементов в данном примере он является родителем (parent), следовательно, его можно получить через любой дочерний элемент:

//div[text()=’Тема’]/parent::div

Обращение к родительскому элементу и тег можно заменить двумя точками:

//div[text()=’Тема’]/..

Далее, так как все элементы в примере дети, то можно любого из них найти от родителя:

Example.html

```
<div id=”__vt8903” class=”listitem Folder” enabled=”enabled”>
   <div class=”content”>
      <div class=”txtNode”>
         <div class=”name”>
            <span class=”nameSpan”>Folder name</span>
            <div class=”team list-icon”></div>
            <div class=”priv list-icon”></div>
            <div class=”publ list-icon”></div>
            <div class=”star list-icon”></div>
         </div>
```

//div[contains(@class,’has-floating’)]/child::input – необходимо найти родителя, а затем ребенка с тегом input.

3. **descendant** – потомок, но в отличие от child, это может быть потомок любой вложенности. Непосредственный потомок (ребёнок) входит в это понятие.

Например, необходимо получить папку именно с определенным именем, но верстка организована так, что сам текст не содержится именно в элементе класса "Folder". Поэтому вначале необходимо найти класс, а потом отфильтровать ту папку, у которой в потомках есть нужный текст:

//div[@class='listitem Folder']/descendant::span[text()='Folder name']

Здесь сначала находится класс папки, потом среди его потомков ищется тег span и нужный текст. В случае, если на странице существует несколько элементов с подобным текстом, найдётся именно папка, как изначально и требовалось.

Вместо **descendant** можно использовать двойной слеш **//**. Это означает любой вложенный элемент. Пример выше можно переписать:

//div[@class='listitem Folder']//span[text()='Folder name']

4. **ancestor** – предок любой удалённости. Parent входит в это понятие. Элемент папки по тексту из предыдущего примера можно найти следующим образом:

//span[text()='Folder name']/ancestor:: div[@class='listitem Folder']

Использовать в одном локаторе несколько отношений можно, но крайне нежелательно:

//div[@class='One]/child::div[@class='Two']/descendant::input[@class='Three]

Такой локатор без труда будет работать, но он сложно читается и в большинстве случаев существует возможность подобрать другой локатор. Совсем недопустимо использование в одном локаторе обратных отношений, т.е. сначала поиск потомка, а потом его предка или наоборот.

В данном разделе были рассмотрены основные команды и отношения, которые могут пригодиться при написании локаторов. С остальными командами и отношениями можно ознакомиться в соответствующей документации.

2.2. CSS-локаторы

CSS (Cascading Style Sheets) – это язык иерархических правил (таблиц стилей), используемый для представления внешнего вида документа, написанного на HTML или XML (включая различные языки XML, например, SVG и XHTML). CSS описывает, каким образом элемент должен отображаться на экране, на бумаге, голосом или с использованием других медиасредств.

CSS является одним из основных языков свободной веб-разработки, который стандартизован спецификацией W3C.

Стандарт CSS делится на уровни:

1. **CSS1** – в настоящее время устарел.
2. **CSS2.1** – рекомендован для применения.
3. **CSS3 –** разбит на более мелкие модули, развивается на пути стандартизации.

2.2.1. Основные виды селекторов

Основные виды CSS селекторов:

1. – любые элементы.
2. div – элементы с тегом.
3. #id – элемент с id.
4. .class – элементы с классом.
5. [name="value"] – селекторы на атрибут.
6. :visited – "псевдоклассы", остальные разные условия на элемент.

Селекторы можно комбинировать, для этого их необходимо записать последовательно без пробела:

- .c1.c2 – элементы одновременно с двумя классами "c1" и "c2".
- a#id.c1.c2:visited – элемент "a" с данным "id", классами "c1" и "c2", и псевдоклассом "visited".

Отношения

В CSS3 предусмотрено четыре вида отношений между элементами:

- div p – элементы p, являющиеся потомками div.
- div > p – только непосредственные потомки.
- div ~ p – правые соседи: все p на том же уровне вложенности, которые идут после div.

- div + p – первый правый сосед: p на том же уровне вложенности, который идёт сразу после div (если есть).

Фильтр по месту среди соседей

При выборе элемента можно указать его место среди соседей. Для этого необходимо использовать определённые псевдоклассы:

- :first-child – первый потомок своего родителя.
- :last-child – последний потомок своего родителя.
- :only-child – единственный потомок своего родителя, соседних элементов нет.
- :nth-child(a) – потомок номер "a" своего родителя (например :nth-child(2) – второй потомок). Нумерация начинается с 1.
- :nth-child(an+b) – расширение предыдущего селектора через указание номера потомка формулой, где a, b – константы, а под n подразумевается любое целое число.

Данный псевдокласс будет фильтровать все элементы, которые попадают под формулу при каком-либо n. Например: -:nth-child(2n) даст элементы номер 2, 4, 6..., (четные).

- :nth-child(2n+1) даст элементы номер 1, 3..., (нечётные).
- :nth-child(3n+2) даст элементы номер 2, 5, 8 и так далее.

Фильтр по месту среди соседей с тем же тегом

Существуют аналогичные псевдоклассы, которые учитывают не всех соседей, а только соседей с тем же тегом:

- :first-of-type
- :last-of-type
- :only-of-type
- :nth-of-type
- :nth-last-of-type

Они имеют в точности тот же смысл, что и обычные :first-child, :last-child и так далее, но во время подсчёта игнорируют элементы с другими тегами, чем тот, к которому применяется фильтр.

Селекторы атрибутов

На атрибут целиком:

- [attr] – атрибут установлен.
- [attr="val"] – атрибут равен val.

На начало атрибута:

- [attr^="val"] – атрибут начинается с val, например, "value".
- [attr|="val"] – атрибут равен val или начинается с val-, например, равен "val-1".

На содержание:

- [attr*="val"] – атрибут содержит подстроку val, например, равен "myvalue".
- [attr~="val"] – атрибут содержит val как одно из значений через пробел.

Например, [attr~="delete"] верно для "edit delete" и неверно для "undelete" или "no-delete".

На конец атрибута:

- [attr$="val"] – атрибут заканчивается на val, например, равен "myval".

Другие псевдоклассы

- :not(селектор) – все, кроме подходящих под селектор.
- :focus – в фокусе.
- :hover – под мышью.
- :empty – без детей (даже без текстовых).
- :checked, :disabled, :enabled – состояния INPUT.
- :target – этот фильтр сработает для элемента, ID которого совпадает с анкором #... текущего URL.

Например, если на странице есть элемент с id="intro", то правило: target { color: red } подсветит его в том случае, если текущий URL имеет вид http://...#intro.

2.2.2. Правила использования

В данном разделе объясняется как, когда и какие селекторы следует использовать.

1. **Класс.**

В CSS очень удобно использовать класс элемента (если он уникален), применяя просто знак "." (точка). Например:

```
<div id=”c__dx351” class=”fbBtn btn-grflat btn font_button” enabled=”enabled”>
```

Для элемента с данным тегом можно указать локатор "**.fbBtn**". Если элементов с таким словом в классе несколько, то можно указать еще один из классов в строке, например, "**.fbBtn.btn-grflat**". Необходимо понимать, что все слова, перечисленные через пробел в теге class, это все классы CSS, назначенные данному элементу (т.е. найти его можно по любому из них).

2. **Id.**

В случаях, когда id уникален, используется такая запись: #myid.

3. **Значения атрибутов.**

Легко найти элемент по значению его атрибута, если он не обладает уникальным классом или id.

```
<input id=”c__db231_input” class=”input__nativeinput” type=”text” placeholder=”Эл. почта” name=”email” value=””/>
```

Как видно из примера, в его описании почти нет уникальных элементов, вполне оправдано использовать атрибут name и его значение. Получается **input[name=’email’]** Этот вариант достаточно короток и понятен. Квадратные скобки, как и у **xpath**, означают фильтр. Если условий несколько, то можно использовать запись в стиле **input[name=’email’][placeholder=’Эл. почта’]** – в данном случае это тот же элемент.

4. **Поиск по частичному совпадению значения атрибута.**

А конкретнее, использования аналогов функций "СОДЕРЖИТ", "НАЧИНАЕТСЯ С", "ОКАНЧИВАЕТСЯ НА". Input из вышеуказанного примера можно найти и таким образом: **input[placeholder*=’Эл’]**, запись "*=" означает "СОДЕРЖИТ", аналог *contains()* в **xpath**. В данном случае ее можно заменить функцией "НАЧИНАЕТСЯ С" следующим образом: **input[placeholder^=’Эл’]**, т.е. "^=" является аналогом *starts-with* в **xpath**. И, наконец, **input[placeholder$=’почта’]**, дает выражение "$=", означающее "ОКАНЧИВАЕТСЯ НА" и не имеющее аналогов в **xpath**.

5. **Непосредственный наследник и потомок.**

Для того, чтобы добраться до непосредственного наследника (аналог child в xpath), используется запись в стиле div>div

```
<div id=”c__da135” class=”signInUsingEmailLbl font_title view” enabled=”enabled”>
   <div class=”lbl-cnt”>Вход</div>
</div>
```

В данном примере для того, чтобы получить div с текстом "Вход", можно использовать следующую запись: **div.signInUsingEmailLbl>div** – сначала находим "родителя" по его классу (signInUsingEmailLbl), затем его "ребенка" с тегом div.

```
<div id=”c__da135” class=”signInUsingEmailLbl font_title view” enabled=”enabled”>
   <div class=”lbl-cnt”>Вход</div>
   <div class=”ex-cnt”></div>
</div>
```

Для того, чтобы получить потомка любой вложенности (аналог // или descendant в xpath), нужно поставить пробел. Чтобы найти элемент с классом "ex-cnt" в примере выше, используется **div.signInUsingEmailLbl div.ex-cnt** (находим родителя, а после – любого потомка с классом ex-cnt).

Выше были рассмотрены самые полезные функции в **CSS**. Они достаточно легки для запоминания, просты в написании и понимании.

Минусом **CSS** является то, что поиск идет только сверху вниз. Нет никакой возможности найти предка. Найти можно только потомков (в **xpath** функции ancestor и parent).

Итак, использовать CSS необходимо, если:

- есть уникальный класс (.fbBtn) или id;
- есть возможность определить элемент по атрибуту или его части (input*=’email’);
- есть возможность определить элемент более короткой записью, чем в **xpath**.

В других случаях, когда нам нужен предок, нужно движение по иерархии вверх (или же вверх-вниз), нужны сложные условия для получения элемента, необходимо использовать **xpath**.

2.3. Поиск локаторов в браузере

Google Chrome предоставляет встроенный инструмент для отладки, называемый **Chrome DevTools**, включающий удобную функцию, которая может оценивать или проверять селекторы XPath/CSS без использования каких-либо сторонних расширений.

Работа с панелью "Элементы"

Нажмите клавишу **F12** на клавиатуре, чтобы открыть **Chrome DevTools**. Панель элементов должна быть открыта по умолчанию. Нажмите сочетание клавиш **Ctrl + F**, чтобы включить поиск DOM на панели. Введите критерии выбора XPath или CSS (Рис. 2).

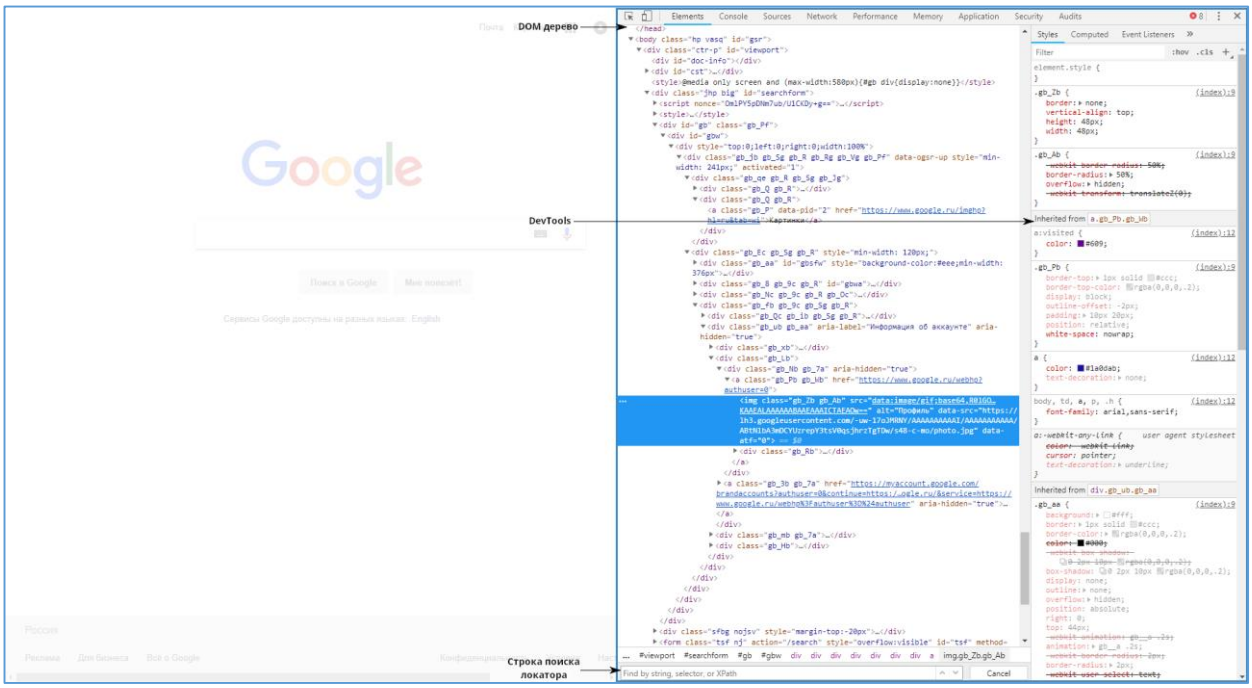

Рис. 2 Google Chrome. Панель "Элементы"

Если есть согласованные элементы, они будут выделены в DOM. Однако, если в DOM есть соответствующие строки, они также будут считаться допустимыми. Например, заголовок селектора CSS должен соответствовать всем (встроенные CSS, скрипты и т.д.), которые содержат заголовок слова, вместо соответствия только элементам.

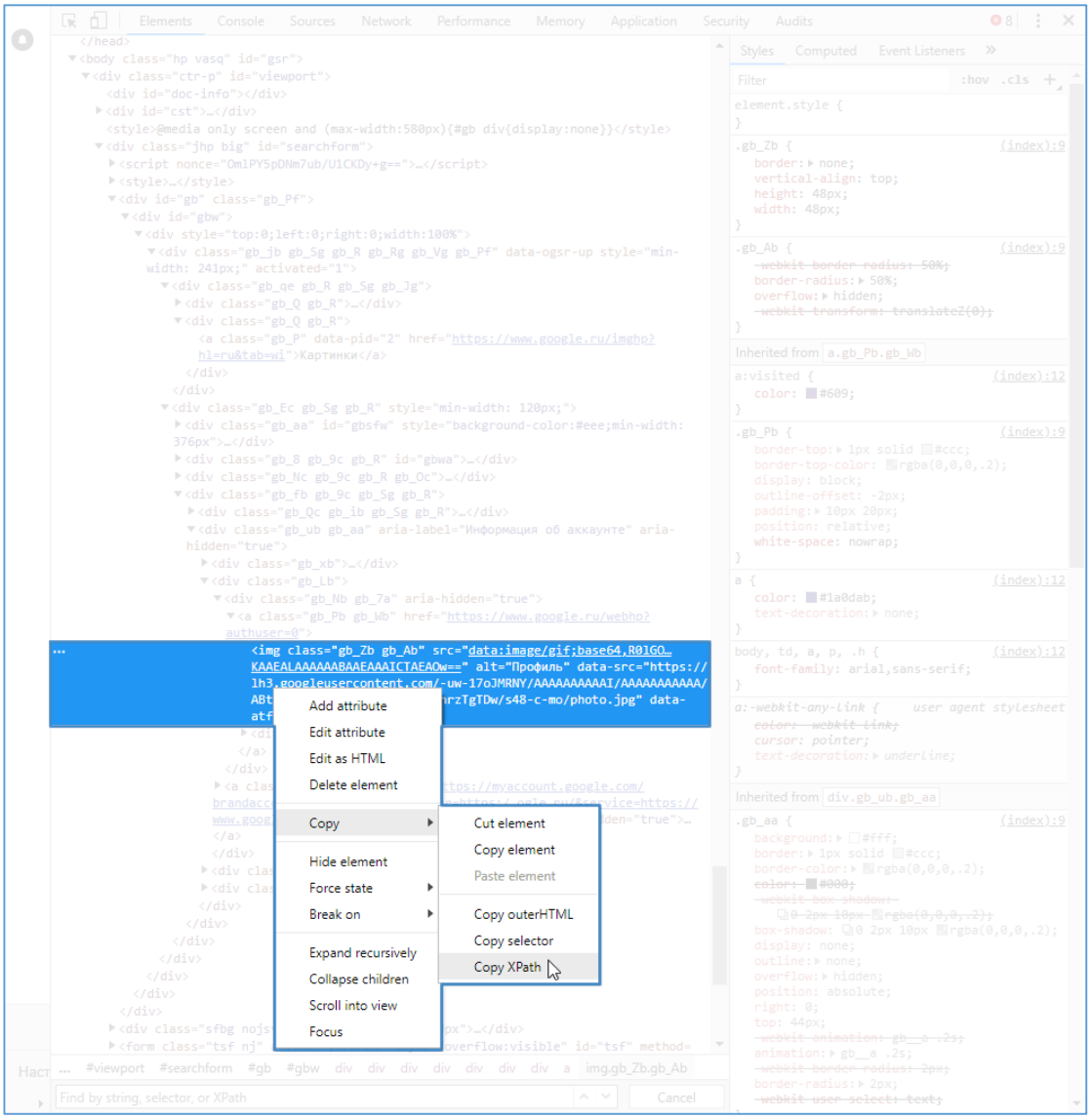

Рис. 3 Google Chrome. Контекстное меню на панели "Элементы"

Глава 3. Настройка окружения

Для создание тестового проекта понадобится JAVA, Maven, Selenide, TestNG.

3.1. JAVA

Java – это достаточно сильно типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре с помощью виртуальной Java-машины (JVM).

3.1.1. JDK, JRE, JVM

Виртуальная машина Java (JVM) – это виртуальная машина, которая запускает байт-коды Java. JVM не понимает Java typo, поэтому файлы *.java компилируются в файлы *.class, содержащие байт-код, понятный для JVM. Благодаря этой особенности программы, написанные на Java, могут быть легко перенесены на различные системы, в которых установлена JVM. Существует множество конкретных реализаций JVM для разных систем (Windows, Linux, MacOS и т.д.).

Java Runtime Environment (JRE) предоставляет библиотеки, виртуальную машину Java и другие компоненты для запуска апплетов и приложений, написанных на языке программирования Java. Кроме того, две ключевые технологии развертывания являются частью JRE:

- **Java Plug-in** – позволяет апплетам работать в популярных браузерах;
- **Java Web Start** – развертывает автономные приложения по сети. Он также является основой технологий Java 2 Platform, Enterprise Edition (J2EE) для разработки и развертывания корпоративного программного обеспечения.

JRE не содержит таких инструментов и утилит, как компиляторы или отладчики для разработки апплетов и приложений.

Java Development Kit (JDK) является надмножеством JRE и содержит все, что есть в JRE, а также такие инструменты, как компиляторы и отладчики, необходимые для разработки апплетов и приложений.

3.1.2. Установка JDK

Для установки JDK необходимо выполнить описанные ниже действия.

1. Перейдите на официальный сайт Oracle.
2. Нажмите "Загрузить JDK".
3. После скачивания файла необходимо запустить его и принять лицензионное соглашение.
4. Загрузите последнюю версию Java JDK для вашей версии (32 или 64 бит) java для Windows.
5. После завершения загрузки запустите файл .exe для установки JDK. Следуйте указаниям установочной программы.
6. После завершения установки нажмите "Закрыть".
7. Щелкните правой кнопкой мыши на "Мой компьютер" и выберите "Свойства".
8. Выберbте "Расширенные настройки системы".
9. Нажмите "Переменные среды".
10. Добавьте в переменную среды Path-путь к папке **bin** в папке, куда установили JDK.
11. Перейдите в командную строку и введите команды java –version

Если всё было сделано верно и JDK установлен правильно, в консоли появится версия установленной JDK.

3.2. Apache Maven

Apache Maven – это фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM, являющемся подмножеством XML. Проект Maven издаётся сообществом Apache Software Foundation, где формально является частью Jakarta Project.

Maven обеспечивает декларативную, а не императивную (в отличие от средства автоматизации сборки Apache Ant) сборку проекта. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

3.2.1. Основные концепции

3.2.1.1. Соглашения по конфигурации

Maven поддерживает принцип соглашения по конфигурации, заключающийся в том, что рассматриваемые аспекты нуждаются в конфигурации тогда и только тогда, когда этот аспект не удовлетворяет некоторой спецификации. Как следствие, это позволяет сократить количество требуемой конфигурации без потери гибкости. Одним из следствий применения данного принципа является то, что отсутствует необходимость указывать пути к файлам в явном виде, что упрощает содержимое pom.xml. Однако, почти все стандарты, на которые опирается Maven, могут быть изменены индивидуальной конфигурацией.

3.2.1.2. Архетипы

Maven использует принцип Maven-архетипов (англ. Archetypes). **Архетип** – это инструмент шаблонов, каждый из которых определён паттерном или моделью, по аналогии с которой создаются производные.

Стандартная структура каталогов – одна из реализаций принципа архетипов в Maven. Следующая структура показывает важнейшие каталоги для проекта на Java:

- Корневой каталог проекта: файл *pom.xml* и все дальнейшие подкаталоги
  - *src*: все исходные файлы
    - *src/main*: исходные файлы собственно для продукта
      - *src/main/java*: Java-исходный текст
      - *src/main/resources*: другие файлы, которые используются при компиляции или исполнении (например, Properties-файлы)

- *src/test*: исходные файлы, необходимые для организации автоматического тестирования
  - *src/test/java*: тест-задания для автоматического тестирования
- *target*: все создаваемые в процессе работы Мавена файлы
  - *target/classes*: компилированные Java-классы

3.2.1.3. Жизненный цикл

Жизненный цикл Maven проекта – это список поименованных фаз, определяющий порядок действий при его построении.

Жизненный цикл Maven содержит три независимых порядка выполнения:

1. clean – жизненный цикл для очистки проекта. Содержит следующие фазы:
   - pre-clean;
   - clean;
   - post-clean.
2. default – основной жизненный цикл, содержащий следующие фазы:
   - validate – выполняется проверка, является ли структура проекта полной и правильной;
   - generate-sources;
   - process-sources;
   - generate-resources;
   - process-resources;
   - compile – компилируются исходные тексты;
   - process-test-sources;
   - process-test-resources;
   - test-compile;
   - test – собранный код тестируется заранее подготовленным набором тестов;
   - package – упаковка откомпилированных классов и прочих ресурсов (например, в JAR-файл);
   - integration-test – программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта;
   - install – установка программного обеспечения в локальный Maven-репозиторий, чтобы сделать его доступным для других проектов текущего пользователя;
   - deploy – стабильная версия программного обеспечения распространяется на удаленный Maven-репозиторий, чтобы сделать его доступным для других пользователей.

3. site – жизненный цикл генерации проектной документации. Состоит из фаз:
    - pre-site;
    - site;
    - post-site;
    - site-deploy.

Стандартные жизненные циклы могут быть дополнены функционалом с помощью Maven-плагинов. Плагины позволяют вставлять в стандартный цикл новые шаги (например, распределение на сервер приложений) или расширять существующие шаги.

3.2.1.4. Зависимости

В файле pom.xml задаются зависимости, которые имеет управляемый с помощью Maven проект. Менеджер зависимостей основан на нескольких основных принципах.

1. **Репозитории**. Maven ищет необходимые файлы в локальных каталогах или в локальном Maven-репозитории. Если зависимость не может быть локально разрешена, Maven подключается к указанному Maven-репозиторию в сети и копирует в локальный репозиторий. По умолчанию Maven использует Maven Central Repository, но разработчик может конфигурировать и другие публичные Maven-репозитории (например, Apache, Ibiblio, Codehaus или Java.Net).
2. **Транзитивные зависимости**. Необходимые библиотеки подгружаются в проект автоматически. При разрешении конфликта версий используется принцип "ближайшей" зависимости, т.е. выбирается зависимость, путь к которой через список зависимых проектов является наиболее коротким.
3. **Исключение зависимостей**. Файл описания проекта предусматривает возможность исключить зависимость в случае обнаружения цикличности или отсутствия необходимости в определённой библиотеке.
4. **Поиск зависимостей**. Поиск зависимостей (open-source-библиотек и модулей) ведётся по их координатам (groupId, artifactId и version). Эти координаты могут быть определены с помощью специальных поисковых машин, например, Maven search engine. Например, по поисковому признаку "pop3" поисковая машина предоставляет результат с groupId="com.sun.mail" и artifactId="pop3".
5. **Менеджеры репозиториев**. Репозитории реализуются с помощью таких менеджеров репозиториев Maven (Maven Repository Manager), как Apache Archiva, Nexus (ранее Proximity), Artifactory, Codehaus Maven Proxy или Dead Simple Maven Proxy.

Область распространения зависимости позволяет включать зависимости только на определённую стадию построения проекта. Существует 6 возможных областей:

1. **compile**. Область по умолчанию. Зависимость доступна во всех путях поиска классов в проекте. Распространяется на зависимые проекты.
2. **provided**. Область аналогична compile, за исключением того, что JDK или контейнер сам предоставит зависимость во время выполнения программы.
3. **runtime**. Зависимость не нужна для компиляции, но нужна для выполнения.
4. **test**. Зависимость не нужна для нормальной работы приложения, а нужна только для компиляции и запуска тестов.
5. **system**. Область аналогична provided за исключением того, что содержащий зависимость JAR указывается явно. Артефакт не ищется в репозитории.
6. **Import**. Начиная с версии Maven 2.0.9, используется только с зависимостью типа pom в секции <dependencyManagement>. Зависимости текущего POM заменяются на зависимости из указанного POM.

3.2.2. Установка Apache Maven

Для установки Apache Maven необходимо выполнить описанные ниже действия.

1. Зайдите на официальный сайт Apache Maven.
2. В разделе "Загрузка" скачайте последнюю стабильную версию.
3. Распакуйте архив в инсталляционную директорию. Например, в C:\Program Files\maven\ (в Windows).
4. Установите переменную окружения M2_HOME, значение – путь до распакованной папки.
5. В переменной PATH добавьте к списку директорий строку %M2_HOME%\bin".
6. Проверьте корректность установки, набрав в командной строке mvn –version.

3.3. IntelliJ IDEA

IntelliJ IDEA – это среда разработки для Java, включающая поддержку всех последних технологий и фреймворков. IntelliJ IDEA предоставляет инструменты для продуктивной работы и подходит для создания коммерческих, мобильных и веб-приложений.

3.3.1. Установка IntelliJ IDEA

Существует две версии IntelliJ IDEA:

- **Community** – бесплатная, но ограниченная. В ней не поддерживаются фреймворки Spring, Vaadin, GWT, языки JavaScript и TypeScript, SQL и многое другое;
- **Unlimited** – включает в себя всевозможные фичи для коллективной разработки, но бесплатна только в первые 30 дней пробного периода. Потом придётся заплатить минимум $533.

В данном случае мы используем сторонние фрейморки и нам подходит бесплатная версия.

Для установки в ОС Windows нам потребуется скачать с официального сайта исходник с расширением .exe. Далее запускаем его и следуем инструкциям установщика.

Глава 4. Selenide

Selenide – это библиотека для написания лаконичных и стабильных UI тестов с открытым исходным кодом.

4.1. Основной API

Основные методы

Основные методы – это open, $ и $$:

- **$(String cssSelector)** – возвращает объект типа SelenideElement, который представляет первый найденный по CSS-селектору элемент на странице;
- **$(By)** – возвращает "первый SelenideElement" по локатору типа By;
- **$$(String cssSelector)** – возвращает объект типа ElementsCollection, который представляет коллекцию всех элементов, найденных по CSS-селектру;
- **$$(By)** – возвращает "коллекцию элементов" по локатору типа By.

Методы $ возвращают объект класса SelenideElement, с ним можно совершать различные действия:

- $(byText("Sign in")).click();

и даже несколько действий сразу:

- $(byName("password")).setValue("qwerty").pressEnter();

либо проверить какое-то условие:

- $(".welcome-message").shouldHave(text("Welcome, user!")).

$$ можно удобно использовать, когда нужный элемент является одним из группы однотипных элементов. Например, вместо:

$(byXpath("//*[@id='search-results']//a[contains(text(),'selenide.org')]")).click();

можно использовать более читабельный и лаконичный вариант:

$$("#search-results a").findBy(text("selenide.org")).click();

Такой составной локатор удобен тем, что в случае ошибки нахождения элемента позволяет по сообщению об ошибке сразу же определить, какая из "частей" не сработала. В первом же случае мы сможем лишь получить информацию о том, что "целый локатор" не сработал, и потратим больше времени на поиск той "части", которая привела к ошибке.

Также, если нужно работать с разными элементами одной и той же коллекции, есть возможность вынести коллекцию элементов в переменную:

```
ElementsCollection resultLinks = $$("#search-results a");
resultLinks.first().shouldHave(text("selenide.org"));
resultLinks.get(1).shouldHave(text("ru.selenide.org"));
resultLinks.shouldHave(size(10));
resultLinks.findBy(text("github.com/codeborne/selenide")).click()
```

Таким образом мы не повторяем локатор "#search-results a" в разных местах, и, следовательно, если он поменяется, нам нужно будет внести изменения только в одном месте.

Также удобно в IDE использовать autocomplete при наборе имени уже созданной переменной resultLinks.

Класс SelenideElement – обёртка вокруг Selenium WebElement, которая добавляет несколько весьма полезных методов.

Сам объект SelenideElement, который возвращается методом $, является прокси-элементом. В момент создания с помощью $ реальный элемент на странице не ищется. Зато потом при любой попытке совершить с прокси-элементом какое-то действие или проверку, прокси-элемент получает последнюю актуальную "версию" реального элемента со страницы (типа WebElement) и проксирует ей указанное действие или проверку.

Методы поиска внутренних элементов

- find(String cssSelector) | $(String cssSelector)
- find(By) | $(By)
- findAll(String cssSelector) | $$(String cssSelector)
- findAll(By) | $$(By)

Здесь $ и $$ просто лаконичные синонимы для соответствующих команд.

Все эти методы возвращают прокси-элементы, т.е. не ищут реальные элементы до тех пор, пока не будет вызвано какое то действие над элементом (например, .click()).

Таким образом, можно пошагово уточнять, какой внутренний элемент необходимо получить внутри внешнего элемента, строя цепочку последовательных вызовов,. Например:

```
$("#header").find("#menu").findAll(".item")
```

Можно сохранить цепочку поиска любой длины в переменную независимо от того, загружена ли страница на момент инициализации переменной, или даже открыт ли браузер.

Методы-проверки состояния элемента (assertions)

Методы-проверки assertions – инициируют для прокси-элемента поиск актуальной версии элемента на странице, совершают проверку по предикату ("условию" – объекту класса Condition) и возвращают объекты SelenideElement, позволяя использовать цепочки вызовов.

- should(Condition) | shouldBe(Condition) | shouldHave(Condition)
- shouldNot(Condition) | shouldNotBe(Condition) | shouldNotHave(Condition)

Проверки играют роль явных ожиданий. Они ждут до удовлетворения условия (visible, enabled, text("some text")), пока не истечет таймаут (значение Configuration.timeout, которое установлено по умолчанию в 4000 миллисекунд).

Можно использовать проверки явно – с целью ожиданий нужного состояния у элементов перед действием, например, $("#submit").shouldBe(enabled).click();

Есть версии явных ожиданий с указанием таймаута:

- waitUntil(Condition condition,long timeout) – аналогично should и его синонимам, но с возможностью установки timeout;
- waitWhile(Condition condition,long timeout) – аналогично shouldNot и его синонимам, но с возможностью установки timeout.

Методы-действия над элементом

- click() – левый клик;
- doubleClick() – левый даблклик;
- contextClick() – правый клик;
- hover() – навести курсор на элемент;
- setValue(String) | val(String) – установить значение элемента;
- pressEnter() – навести фокус на элемент и нажать клавишу **Enter**;
- pressEscape() – имитация нажатия клавиши **Esc**;
- pressTab() – имитация нажатия клавиши **Tab**;
- selectRadio(String value) – выбрать одно из значений radio button;
- selectOption(String) – выбрать одно из значение dropdown list;
- append(String) – добавить текст к имеющемуся в строке;
- dragAndDropTo(String) – имитация drag&drop.

При вызове этих методов, инициализируется поиск актуальной версии элемента на странице, и совершают соответствующее действие.

Действия имеют встроенные неявные ожидания до видимости элемента, пока не истечет таймаут (значение Configuration.timeout, которое установлено по умолчанию в 4000 миллисекунд).

Следовательно, код $("#submit").shouldBe(visible).click() избыточен и будет достаточно $("#submit").click().

Большинство действий также возвращают объект SelenideElement, позволяя использовать цепочки вызовов, например – $("#edit").setValue("text").pressEnter();.

Методы получения статусов элементов и значений их атрибутов

Инициируют для прокси-элемента поиск актуальной версии элемента на странице и имеют разную логику неявных ожиданий в зависимости от контекста:

Выполняется ожидание до появления элемента в DOM:

- getValue() | val() – получить значения атрибута value;
- data() – получить значения атрибута data;
- attr(String attr) – получить значения атрибута attr;
- text() – получить видимый текст элемента;
- innerText() – получить текст элемента в DOM;
- getSelectedOption() – получить текущую опцию выбранного элемента dropdown list;
- getSelectedText() – получить текущий текст выбранного dropdown list;
- getSelectedValue() – получить текущее значение выбранного элемента dropdown list.

Нет ожидания :

- isDisplayed() – возвращает false, если элемент либо невидимый, либо его нет в DOM;
- exists() – возвращает true, если элемент есть в DOM, иначе – false.

Другие полезные команды

- uploadFromClasspath(String... fileName) – загрузить файл или файлы, filename – путь до файла;
- download() – скачать файл;
- uploadFile(File... file) – загрузить файл или файлы.

4.1.1. Condition

Условия используются в конструкциях should / shouldNot / waitUntil / waitWhile. Рекомендуется статически импортировать используемые условия, чтобы получить все преимущества читаемого кода.

Example

```
public class ExampleTest {
  SelenideElement input = $("#input").should(Condition.visible);
}
```

Со статическим импортом:

ExampleWithStaticImport

```
import static com.codeborne.selenide.Condition.*;

public class ExampleTest {
   SelenideElement input = $("#input").should(visible);
}
```

Перечень возможных условий:

- visible | appear – проверка видимости элемента;
- present | exist – проверка присутствия элемента на странице;
- hidden | disappear | not(visible) – проверка отсутствия элемента на странице;
- readonly – проверка, что статус элемента readonly;
- name – проверка значения тега name;
- value – проверка значения тега value;
- type – проверка значения тега name type;
- empty – проверка, что текст элемента пуст;
- attribute(name) – проверка наличие атрибута name;
- attribute(String name, String value) – проверка, что атрибут name имеет значение value;
- cssClass(String value) – проверка, что атрибут class имеет значение value;
- focused – проверка, что у элемента есть псевдокласс focus;
- enabled – проверка, что у элемента есть псевдокласс enabled;
- disabled – проверка, что у элемента есть псевдокласс disabled;
- selected – проверка, что у элемента есть псевдокласс selected;
- matchText(String regex) – проверка текста элемента по regex;
- text(String substring) – проверка вхождения текста;

- exactText(String wholeText) – проверка точного совпадения текста;
- textCaseSensitive(String substring) – проверка текста с учетом регистра;
- exactTextCaseSensitive(String wholeText) – проверка точного совпадения текста с учетом регистра;
- and;
- or;
- not.

4.1.2. CollectionCondition

- empty – аналогичен size(0);
- size(int i) – проверка количества элементов в массиве;
- sizeGreaterThan(int i) – проверка, что элементов строго больше i;
- sizeGreaterThanOrEqual(int i) – проверка, что элементов больше или равно i;
- sizeLessThan(int i) – проверка, что элементов строго меньше i;
- sizeLessThanOrEqual(int i) – проверка, что элементов меньше или равно i;
- sizeNotEqual(int i) – проверка, что элементов больше или меньше i;
- texts(String... substrings) – проверка вхождения текста с сопоставлением;
- exactTexts(String... wholeTexts) – проверка точного текста с сопоставлением.

4.1.3. Selectors

Класс содержит некоторые By селекторы для поиска элементов по тексту или атрибуту (которых не хватает в стандартном Selenium WebDriver API):

- byText – поиск элемента по точному тексту;
- withText – поиск элемента по тексту (подстроке);
- by – поиск элемента по атрибуту;
- byTitle – поиск по атрибуту "title";
- byValue – поиск по атрибуту "value";
- byCssSelector;
- byClassName;
- byId;
- byPartialLinkText;
- byLinkText;
- byXpath;
- byName.

4.1.4. WebDriverRunner

Этот класс содержит некоторые функции, относящиеся к браузеру:

- isChrome() – вернёт true, если сейчас запущен Google Chrome, иначе – false;
- isFirefox() – вернёт true, если сейчас запущен Mozilla Firefox, иначе – false;
- isHeadless() – вернёт true, если браузер запущен без графического интерфейса пользователя, иначе – false;
- url() – возвращает текущий url;
- source() – возвращает исходный HTML текущего окна;
- getWebDriver() – возвращает объект WebDriver (созданный Selenide автоматически или установленный пользователем);
- setWebDriver(WebDriver) – указывает Selenide использовать драйвер, созданный пользователем. С этого момента пользователь сам ответственен за закрытие драйвера.

4.1.5. Configuration

Этот класс содержит конфигурации для запуска тестов:

- timeout – время ожидания в миллисекундах, которое используется в явных (should/shouldNot/waitUntil/waitWhile) и неявных ожиданиях для SelenideElement. По умолчанию установлено в 4000 мс. Может быть изменено во время исполнения, Configuration.timeout = 6000;
- collectionsTimeout – время ожидания в миллисекундах, которое используется в явных (should/shouldNot) ожиданиях для ElementsCollection. По умолчанию установлено в 6000 мс. Может быть изменено во время исполнения, e.g. Configuration.collectionsTimeout = 8000;
- browser (e.g. "chrome", "ie", "firefox");
- baseUrl;
- reportsFolder.

Также можно передать конфигурационные параметры как system properties.

4.1.6. Быстрый старт

Для первого запуска понадобится настроенное окружение. IDE, maven, java 1.8 и выше.

Нужно создать новый пустой maven проект в IDE:

- groupId – название компании в обратном порядке (например, com.example);
- artifactId – наименование проекта (например, ui-test).

После создания IDE автоматически создала файл pom.xml.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>ui-tests</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Нужно подключить к нашему проекту библиотеку Selenide, для этого нужно добавить зависимость com.codeborne:selenide.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>ui-tests</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>com.codeborne</groupId>
            <artifactId>selenide</artifactId>
            <version>4.14.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Теперь после того, как IDE скачает и подключит нужные файлы, можно написать первый тест. Создадим класс в директории .\src\test\java (например, ExampleTest).

ExampleTest.java

```
public class ExampleTest {
}
```

В нём создадим тестовый метод (например, TestOne). Все тестовые методы должны быть публичными и не статическими.

ExampleTest.java

```
public class ExampleTest {
   public void TestOne (){
   }
}
```

Библиотека selenide автоматически скачивает нужный драйвер для работы с браузером, нужно лишь указать, какой именно браузер использовать до первого его открытия. Откроем страницу http://google.com в браузере Google Chrome и проверим наличие кнопки поиска на странице.

ExampleTest.java

```
import com.codeborne.selenide.Condition;
import com.codeborne.selenide.Configuration;
import org.junit.jupiter.api.Test;

import static com.codeborne.selenide.Selenide.$;
import static com.codeborne.selenide.Selenide.open;

public class ExampleTest {
   @Test
   public void TestOne(){
      Configuration.browser = "chrome";
      open("http://google.com");
      $("input[value=\"Поиск в Google\"]").should(Condition.exist.because("Кнопка поиск должна быть на стартовой странице."));
   }
}
```

Метод помечен специальной аннотацией @Test, selenide и включает в себя тестовую библиотеку junit, которая позволяет логгировать и запускать тесты. Первая строка метода настраивает, какой браузер будет использоваться, в данном случае – Google Chrome. Остальные настройки конфигурации можно

посмотреть в главе **Configuration** главы **Selenide**. Второй строкой мы открываем в данном браузере страницу http://google.com. Третьей – ищем элемент по css-селектору. В данном случае – input со значение (value) "Поиск в Google", и проверяем, что он существует, также указываем, почему мы делаем эту проверку.

Запустить тест можно, нажав зеленую стрелку слева от сигнатуры метода в IDE или введя в терминале mvn clean test.

4.2. Примеры использования основных команд

4.2.1. Навигация и браузер

Для открытия страницы в браузере Вы можете использовать такой код:

Example

```
open("https://goo.gl/aX4cQp");
```

Но лучше указать базовый url и использовать относительные адреса.

Example

```
Configuration.baseUrl = https://goo.gl;
open("/aX4cQp");
```

Для получения текущего url страницы используйте **getWebDriver().getCurrentUrl();**

Для получения текущего исходного кода страницы используйте **getWebDriver().getPageSource();**

Для получения, текущего title страницы используйте **getWebDriver.getTitle();**

Для настройки размера окна браузера используйте **getWebDriver().manage().window().setSize(new Dimension(x,y));**. Можно использовать готовые настройки максимального возможного разрешения – **Configuration.startMaximized = true;**

Для перехода по вкладкам браузера используйте **getWebDriver().switchTo().window(tabs.get(i));**

Для soft-update можно использовать **getWebDriver.navigate().refresh();**

Для перехода по истории можно использовать **getWebDriver.navigate().forward();** и **getWebDriver.navigate().back();**

Для скачивания файла можно использовать **File downloadedFile = $("input.link").download();**. Файл будет сохранен в папку, настроенную по умолчанию. Изменить её можно через **Configuration.reportsFolder**.

Для загрузки файла можно использовать **$("input.link").uploadFile(new File("/exampl$("div").txt"));.**

Для выполнения JavaScript-кода на стороне клиента необходимо вызвать метод **executeJavaScript("console.log('Hello')");**

4.2.2. Поиск элементов на странице

Для поиска элемента, который имеет **уникальный идентификатор**, можно использовать методы:

ById

```
$("#someId");
$(By.id("someId"));
$x(".//*[@id='someId']")
```

Стоит заметить, что последний вариант – наименее читабельный и его следует избегать.

Для поиска элемента по **css-селектору**:

ByCss

```
$(".someCssSelector");
$(By.cssSelector(".someCssSelector"));
```

Для поиска элемента по **xpath-селектору**:

ByXpath

```
$x(".//*[@class='price']");
$(By.xpath(".//*[@class='price']"));
```

Далее будет использоваться короткая запись xpath селектора – $x.

Для поиска элемента, который имеет **уникальное значение атрибута class**:

ByClass

```
$(".nav");
$(By.className("nav"));
$x(".//*[@class='nav']");
```

Для поиска элемента, который имеет **уникальное значение атрибута name**:

ByName

```
$(By.name("someName"));
$x(".//*[@name='someName']");
```

Для поиска элемента, который имеет **уникальный атрибут**:

ByTag

```
$(By.tagName("unical"));
```

```
$("unical");
$x(".//unical");
```

Для поиска элемента, который имеет **уникальный текст ссылки**:

ByLinkText

```
$(By.linkText ("link"));
$x("//a[text(),'link']");
```

Для поиска элемента, который имеет **уникальное вхождение текста ссылки**:

ByPartialLinkText

```
$(By.partialLinkText("lin"));
$x("//a[contains(text(),'lin')]");
```

Для поиска элемента, который имеет **уникальный текст**:

ByText

```
$(byText("Logout"));
$x("//*[text(),'Logout']");
```

Для поиска элемента, который имеет **уникальное вхождение текста**:

ByWithText

```
$(withText("out"));
$x("//*[contains(text(),'out')]");
```

Для поиска элемента, который имеет **уникальное значение атрибута**:

ByAttribute

```
$(byAttribute("href", "/main"));
$("[href='/main']");
$x(".//*[@href='/main']");
```

Для поиска элемента, который имеет **уникальное значение атрибута value**:

ByValue

```
$(byValue("someValue"));
$(byAttribute("value", "someValue "));
$("[value='someValue']");
$x(".//*[@value='someValue']");
```

Для поиска элемента **цепочкой**:

ByChain

```
$("#mainElement").$x(".//*[@id='subElement']");
```

4.2.3. Работа с элементами

Для имитации **клика** по элементу необходимо вызвать метод click() экземпляра класса SelenideElement:

Click

```
$("#element").click();
```

Для имитации **двойного клика** по элементу необходимо вызвать метод doubleClick() экземпляра класса SelenideElement:

DoubleClick

```
$("#element").doubleClick();
```

Для имитации **правого клика** по элементу необходимо вызвать метод contextClick() экземпляра класса SelenideElement:

ContextClick

```
$(".g").contextClick();
```

Для имитации **ввода текста** в элемент необходимо вызвать метод sendKeys(String ...) экземпляра класса SelenideElement:

SendKeys

```
$("#lst-ib").sendKeys("selenide");
```

Для имитации **добавления текста** в элемент необходимо вызвать метод sendKeys(String ...) экземпляра класса SelenideElement:

Append

```
$("#lst-ib").append("last");
```

Для имитации **очистки введенного текста** в элемент необходимо вызвать метод clear() экземпляра класса SelenideElement:

Clear

```
$("#lst-ib").clear();
```

Для имитации **очистки введенного текста, а после этого – ввода нового**, необходимо вызвать метод val(String …) экземпляра класса SelenideElement:

Clear

```
$("#lst-ib").val(“selenide”);
```

Для имитации **нажатия Enter** в элементе необходимо вызвать метод pressEnter() экземпляра класса SelenideElement:

DoubleClick

```
$("#lst-ib").val("selenide").pressEnter();
```

Для имитации **навигации по элементам через клавишу Tab** необходимо вызвать метод pressTab() экземпляра класса SelenideElement:

PressTab

```
$("#q").pressTab();
```

Для имитации **нажатия комбинаций клавиш** необходимо использовать метод sendKeys(Keys …) экземпляра класса SelenideElement:

PressKeys

```
$("#lst-ib").sendKeys(Keys.chord(Keys.CONTROL, "a"));
```

Например, **для выполнения копирования** комбинациями клавиш **Ctrl+A** и **Ctrl+C** необходимо вызвать методы с такими параметрами:

PressCopyExample

```
$("#lst-ib").sendKeys(Keys.chord(Keys.CONTROL, "a"));
$("#lst-ib").sendKeys(Keys.chord(Keys.CONTROL, "c"));
```

Для имитации работы с **элементами типа radio button** или **check box** существует специальный метод setSelected(Boolean …) экземпляра класса SelenideElement:

SetSelected

```
$(byText("Female")).setSelected(true);
 $(byText("Male")).setSelected(false);
```

Для **проверки текущего значения элемента типа radio button или check box** существует специальный метод isSelected() экземпляра класса SelenideElement, который возвращает логическое значение текущего положения:

IsSelected

```
Boolean checkBoxIsSelected = $(byText("Female")).isSelected();
```

Для имитации работы с **элементами типа dropdown list** существует специальный метод selectOption(String …) и selectOptionByValue(String …) экземпляра класса SelenideElement:

SelectOption

```
$("#menu").selectOption("text");
$("#menu").selectOptionByValue("value");
```

Для перехода на другой **frame** текущего html документа необходимо использовать метод switchTo().frame(SelenideElemnt …) экземпляра класса SelenideElement, который переключит текущий контекст на контекст выбранного фрейма:

IFrame

```
switchTo().frame($("#framelive"));
```

Для **получения последнего ребенка** первой вложенности существует специальный метод lastChild() экземпляра класса SelenideElement:

LastChild

```
SelenideElement child = $("tr").lastChild(); //return last td element
```

Для **получения соседнего элемента** существует специальный метод closest() экземпляра класса SelenideElement:

Closest

```
SelenideElement table = $("td").closest("table");
```

Для **получения родителя элемента** существует специальный метод parent() экземпляра класса SelenideElement:

Parent

```
SelenideElement child = $("td").lastChild(); //return parent tr element
```

Для **проверки существования элемента** на странице необходимо использовать специальный метод exist() экземпляра класса SelenideElement. Метод выполняется сразу после загрузки страницы без ожидания элемента:

Exist

```
boolean exist = $("a.link").exists();
```

Для **проверки видимости элемента** на странице необходимо использовать специальный метод isDisplayed() экземпляра класса SelenideElement. Метод выполняется сразу после загрузки страницы без ожидания элемента:

Visible

```
boolean visible = $("a.link").isDisplayed();
```

Для имитации **прокрутки к элементу** необходимо использовать метод scrollTo () экземпляра класса SelenideElement:

ScrollTo

```
$("a.link").scrollTo();
```

Для имитации **прокрутки элемента к верхней части окна браузера** необходимо использовать метод scrollIntoView(Boolean …) с параметром true, экземпляра класса SelenideElement:

ScrollIntoViewTop

```
$("a.link").scrollIntoView(true);
```

Для имитации **прокрутки элемента к нижней части окна браузера** необходимо использовать метод scrollIntoView(Boolean …) с параметром false, экземпляра класса SelenideElement:

ScrollIntoViewEnd

```
$("a.link").scrollIntoView(false);
```

Для имитации **наведение курсора элемент** необходимо использовать метод hover() экземпляра класса SelenideElement:

Hover

```
$("a.link").hover();
```

Для **получения исходного кода элемента** существует специальный метод innerHtml() экземпляра класса SelenideElement:

Text

```
String elementHTMLSource = $("div").innerHtml();
```

Для **получения исходного текста элемента** существует специальный метод innerHtml() экземпляра класса SelenideElement:

Text

```
String elementHTMLSource = $("div").innerText();
```

Для **получения видимого текста** существует специальный метод text() экземпляра класса SelenideElement:

Text

```
String text = $("a").text();
String elementInnerText = $("div").innerText();
```

4.2.4. Работа с элементами типа коллекция

Для получения **конкретного экземпляра** объекта массива ElementsCollection можно использовать метод get(int ...):

Get

```
SelenideElement firstElement = $$("a").get(3);
```

Для **получения первого экземпляра** объекта массива ElementsCollection можно использовать метод get(int ...) или first():

First

```
SelenideElement firstElement = $$("a").get();
SelenideElement firstElement = $$("a").first();
```

Для **получения первых N экземпляров** объекта массива ElementsCollection можно использовать метод get(int ...) или first():

nFirst

```
ElementsCollection firstTenElements = $$("a").first(10);
```

Для **получения последнего экземпляра объекта** массива ElementsCollection можно использовать метод get(int ...) или first():

Last

```
SelenideElement lastElement = $$("a").get($$("a").size()-1);
SelenideElement lastElement = $$("a").last();
```

Для **получения первых N** экземпляров объекта массива ElementsCollection можно использовать метод get(int ...) или first():

nLast

```
ElementsCollection lastTenElements = $$("a").last(10);
```

Для **получения видимого текста** каждого элемента массива ElementsCollection можно использовать метод texts():

Texts

```
List<String> texts = $$("a").texts();
```

Для **получения размера массива** ElementsCollection можно использовать метод size():

Size

```
int sizeElements = $$("a").size();
```

Для проверки, что **массив ElementsCollection пуст** можно использовать метод isEmpty:

IsEmpty

```
boolean isEmpty = $$("a").isEmpty();
```

Для **проверки сортировки** можно воспользоваться нижеуказанным алгоритмом. Для примера возьмём сайт http://live.guru99.com, где параметрами запроса можно выполнять сортировку на странице.

Example

```
import com.codeborne.selenide.Configuration;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

import java.util.Collections;
import java.util.List;

import static com.codeborne.selenide.Selenide.$$;
import static com.codeborne.selenide.Selenide.open;
import static org.testng.Assert.assertEquals;

public class SortingTest {
    //installation of configuration settings
    @BeforeClass
```

```
public static void configuration() {
    Configuration.baseUrl = "http://live.guru99.com/index.php";
    Configuration.browser = "chrome";
}

@Test
public void userCanSortProductsByNameinList() {
    // перейти на страницу показа продуктов в виде списка
    open("/mobile.html?mode=list");

    // получить наименование продуктов из списка
    List expectedNames = $$("h2.product-name a").texts();

    // отсортировать полученный список
    Collections.sort(expectedNames);

    // перейти на страницу показа продуктов в виде сортированного списка
    open("/mobile.html?dir=asc&mode=list&order=name");

    // получить наименование продуктов из списка
    List actualNames = $$("h2.product-name a").texts();

    // сравнить два списка
    assertEquals(actualNames, expectedNames, "Not sorted by name.");

}

@Test
public void userCanSortProductsByNameDescInList() {
    // перейти на страницу показа продуктов в виде списка

    open("/mobile.html?mode=list");

    // получить наименование продуктов из списка
    List expectedNames = $$("h2.product-name a").texts();

    // отсортировать полученный список
    Collections.sort(expectedNames);
```

```
        // отсортировать полученный список в обратном порядке
        Collections.reverse(expectedNames);

        // перейти на страницу показа продуктов в виде сортированного в обратном порядке списка
        open("/mobile.html?dir=desc&mode=list&order=name");

        // получить наименование продуктов из списка
        List actualNames = $$("h2.product-name a").texts();

        // сравнить два списка
        assertEquals(actualNames, expectedNames, "Not sorted by name in DESC order.");

    }
}
```

4.2.5. Условия проверки SelenideElement

В библиотеке Selenide существуют готовые условия проверки элементов, их можно использовать для проверки элемента. При проверке они вызывают явные ожидания. Для проверки используются методы класса SelenideElement:

- shouldBe(Condition... ...); || shouldHave(Condition... ...); || should(Condition... ...)
- shouldNotBe(Condition... ...); || shouldNotHave(Condition... ...); || shouldNot (Condition... ...)

При вызове метода should и его синонима фреймворк ожидает выполнения условия в течение установленного таймаута. В случае выполнение условия, тест продолжает выполняться, иначе – ему устанавливается статус Fail и выводится стандартизированная ошибка, если не указана определенная. Для её указания нужно дополнительно указать её, используя метод because(String ...) экземпляра класса Condition. Например:

Example

```
$(".link").shouldNot(Condition.text("Home").because("Текст ссылки не содержит 'Home'"));
```

В данном случае идет поиск элемента со значением атрибута class равным link, получается его видимый текст и проверяется на равенство со строкой "Home", в случае ошибки будет выведен текст:

Output

```
Element should have text 'Home' (because Текст ссылки не содержит 'Home') {.link}
Element: '<a class="link" href="/home">My Page</a>'
Timeout: 4,000 s.
```

Заметим, что проверка выполнялась в течение 4 секунд (время можно изменить через Configuration.timeout), т.к на странице была ошибка. В случае её отсутствия, тест продолжит выполнение сразу после проверки условия.

При вызове метода shouldNot (Condition… …) и его синонимов ожидания будут работать с той же логикой. При невыполнении условия тест продолжится сразу, иначе – элемент будет проверяться в течение установленных рамок.

Условия можно передавать в методы ожидание через запятую, в этом случае они будут проверяться вместе через логический оператор И. Например:

Example

```
$(".link").shouldNot(Condition.text("Home").because("Текст ссылки не содержит 'Home'"),
        Condition.text("Page").because("Текст ссылки не содержит 'Page'"));
```

Псевдокод данной проверки:

Example

```
$(".link").getText().contains("Home") && $(".link").getText().contains("Page")
```

Существуют множества встроенных условий , основные из них указаны в главе **4.1.1**. Рассмотрим некоторые из них на примере.

Для проверки **видимости элемента на странице** можно использовать условие visible:

AssertVisible

```
$("input").shouldBe(visible.because(“Поле ввода не видимо для пользователя”));
```

Для проверки **видимого текста элемента** можно использовать условие text:

AssertText

```
$("button").shouldBe(text(“Подтвердить”).because(“Текст кнопки отличен от ‘Подтвердить’”));
```

Для проверки **значения атрибута элемента** можно использовать условие hasAttribute:

AssertAttribute

```
$("a").shouldBe(hasAttribute(“href”,”/home”).because(“Ссылка ведет на страницу отличную от ‘/home’”));
```

К условиям можно применять условный оператор НЕ. Например:

ConditionWithNot

```
$("a").shouldBe(not(visible.because(“Ссылка должна быть скрыта”)));
```

Аналогом будет:

ConditionWithNot

```
$("a").shouldNotBe(visible.because(“Ссылка должна быть скрыта”));
//или
$("a").shouldBe(hidden.because(“Ссылка должна быть скрыта”));
```

В данном случае нужно выбирать вариант, который будет наиболее понятным и логически простым для понимания человеком.

4.2.6. Условия проверки ElementsCollection

Для экземпляров класса ElementsCollection существуют свои условия, они описаны в классе CollectionCondition. Полный список содержится в Главе **4.1.2**. Данные проверки работают аналогично проверкам SelenideElement (но проверяют весь массив объектов, а не один конкретный элемент), с тем отличием, что к ним не применим логический оператор НЕ. Например:

AssertTexts

```
$$("button").shouldBe(CollectionCondition.texts("Отмена","Ок").
    because("Кнопки имеют текст или порядок отличный от 'Отмена' / 'Ок'"));
```

Стоит заметить, что порядок элементов в массиве так же важен, как и их содержание.

Глава 5. Принципы разработки

5.1. DRY

DRY (Don’t Repeat Youself, "не повторяйся") или **DIE** (Duplication Is Evil, "дублирование – это зло") – данный принцип заключается в том, что нужно избегать повторений одного и того же кода. Лучше использовать универсальные свойства и функции.

Например, есть 2 метода авторизации: авторизация с паролем и авторизация без пароля.

Example

```
public static void auth(String username, String password) {
    open();
    $(By.id("login")).setValue(username);
    $(By.id("password")).setValue(password);
    $(By.id("LogIn")).click();
}

public static void auth(String username) {
    open();
    $(By.id("login")).setValue(username);
    $(By.id("LogIn")).click();
}
```

Вместо дублирования кода можно просто вызвать первый метод с пустым параметром.

Example

```
public static void auth(String username) {
    auth(username,"");
}
```

5.2. KISS

KISS (Keep It Simple, Stupid, "не усложняй!") – смысл данного принципа программирования заключается в том, что стоит делать максимально простую и понятную архитектуру, применять шаблоны проектирования и не изобретать "велосипед".

Например, необходимо отсортировать массив чисел.

Example

```
public static void customSort(int[] arr) {
    for (int i = arr.length – 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}

public static void standartSort (int[] arr) {
    Arrays.sort(arr);
}
```

В первом случае – изобретаем "велосипед", во втором – используем стандартные методы.

5.3. YAGNI

YAGNI (You Ain't Gonna Need It, "вам это не понадобится!") – суть заключается в том, чтобы реализовать только поставленные задачи и отказаться от избыточного функционала.

Например, необходимо проверить, что в процессе верно заполнена одна из контекстных переменных.

Варианты решения:

1. Проверить в тесте, что переменная заполнена верно, остальные проверки будут в других тестах (если это необходимо).

Example

```
public static void checkContextValue (String value) {
   assertTrue($(TEXTFIELD_CONTEXT).getText().contains("value"),
         "Контекстная переменная заполнена не верно");
}
```

2. Проверить в тесте, что переменная заполнена верно, отображение корректно, используется нужный шрифт и т.п.

Example

```
public static void checkAllParams(String value, String attr) {
   assertTrue($(TEXTFIELD_CONTEXT).isDisplayed(),
         "Элемент не отображен на форме");
   assertTrue($(TEXTFIELD_CONTEXT).getAttribute().equals(attr),
         "Элемент не содержит атрибута"+attr);
   assertTrue($(TEXTFIELD_CONTEXT).getText().contains(value),
         "Контекстная переменная заполнена не верно");
   assertTrue($(TEXTFIELD_CONTEXT).lastChild().exists(),
         "Дочерний элемент отсутствует");
}
```

Можно проверить всё и не успеть выполнить другие задачи, либо проверить **только нужное**.

5.4. Частые ошибки

Программирование без планирования

Хороший код должен проходить через следующие стадии:

- Замысел;
- Исследование;
- Планирование;
- Написание;
- Проверка;
- Изменение.

Каждому из этих пунктов надо уделить достаточно усилий.

Новички склонны писать код без предварительных раздумий. Это может сработать на маленьких автономных приложениях, но на крупных может обернуться трагедией.

Прежде, чем что-то сказать, вы обдумываете слова, чтобы потом не было стыдно. Точно также избегайте непродуманного кода, за который однажды будет стыдно. И слова, и код – это отображение ваших мыслей.

Программирование – это на 90% изучение существующего кода и его изменение посредством маленьких, легко тестируемых порций, вписывающихся в общую систему. Само написание кода – это всего лишь 10% работы программиста.

Программирование – это не просто написание строк кода, а творчество, основанное на логике, которое надо развивать в себе.

Чрезмерное планирование

Планировать, прежде чем нырять в написание кода – хорошая вещь. Но даже хорошие вещи могут навредить вам, если переборщить с ними. Даже водой можно отравиться, если слишком много её выпить.

Не ищите идеальный план, такого не существует в мире программирования. Ищите достаточно хороший план для старта. Любой план изменится, зато он заставит вас придерживаться структуры в коде, которая облегчит вашу дальнейшую работу.

Линейное планирование всей программы "от А до Я" (водопадным методом) – не годится для большинства программных продуктов. Разработка подразумевает обратную связь, и вы постоянно будете удалять и добавлять функционал, что

никак нельзя учесть в "водопадном планировании". Планировать следует несколько следующих элементов и каждый новый надо включать в план лишь после гибкой адаптации к реальности (Agile).

К планированию надо подойти очень ответственно, потому что и его недостаток, и избыток могут навредить качеству кода. А качеством кода ни в коем случае нельзя рисковать.

Хвататься за первое решение

Новички, столкнувшись с проблемой, склонны хвататься за первое попавшееся решение, не подумав о побочных эффектах в перспективе. Хорошие решения, в отличие от первых, появляются тогда, когда вы находите разные решения и подбираете из них самое оптимальное для себя. Если у вас не укладывается в голове, что у проблемы может быть несколько решений, значит вы плохо понимаете саму проблему.

Ваша задача как профессионального программиста – найти не первое попавшееся решение, а самое простое. То есть то, которое наиболее просто реализуется, эффективно работает и легко поддерживается.

Не отступать

Другая частая ошибка новичков – не отступать, даже когда они поняли, что выбранное решение не самое лучшее. Подход "не сдаваться" хорош во многих сферах, но не в программировании. Программистам полезно признавать ошибки раньше и чаще. Как только вы засомневались в решении, отбросьте его и переосмыслите проблему. Не важно, сколько вы уже вложили в этот путь. Системы контроля версий типа Git позволяют создавать ветки и экспериментировать с разными решениями, активно пользуйтесь этим.

Не цепляйтесь за код только потому, что вложили в него много времени и сил. Плохой код должен быть отброшен.

Не гуглить

Если вы не пионер на острие новейшей технологии, велика вероятность, что вашу задачу кто-то уже решил. Гуглите, чтобы сэкономить драгоценное время.

Поиск может показать вам проблему с неожиданной стороны. И то, что вы считали решением и хотели реализовать, может оказаться усугублением

проблемы. Даже когда кажется, что вы всё знаете для решения, гугл наверняка удивит вас.

Обратная сторона медали – новички любят копировать из гугла всё подряд без особых раздумий. Но, даже если код решает вашу проблему, используйте его только тогда, когда ясно понимаете каждую его строчку.

Ухудшать код

Представьте, что вам досталась такая бардачная комната. И теперь перед вами стоит задача поставить туда какую-то свою вещь. Многие склонны сразу же сделать это. Задача будет решена за секунды. Но так вы станете соучастниками этого бардака. Правильный подход – это упорядочить ту часть комнаты, куда вы добавляете что-то. Например, если это одежда, то приберитесь в шкафу для одежды, сложите туда всю одежду из комнаты, а потом уже положите свою вещь на отведенное ей место. Всегда улучшайте код хоть немного, но никогда не ухудшайте.

Ниже приведено несколько распространенных ошибок, приводящих к беспорядку в коде.

Дублирование

Если вы копипастите код, только чтобы изменить там одну строчку, вы вносите большой беспорядок в него. В аналогии с комнатой – это как будто вы покупаете стулья разной высоты на все случаи жизни вместо того, чтобы купить один с регулируемым сидением. Держите в голове абстракции и корректируйте их для разных нужд, это упростит код.

Не использование файла конфигурации

В файл конфигурации обязательно нужно помещать те величины, которые могут меняться в зависимости от окружения и встречаются в разных местах кода.

Каждый раз, когда вы вводите в код новую величину, спрашивайте себя: "Может быть стоит поместить её в файл конфигурации?" И почти всегда ответом будет "Да".

Лишние условные операторы и временные переменные

Любой if разветвляет логику вашей программы, поэтому их наличие должно быть сведено к минимуму насколько это возможно без вреда для читабельности. Самое сложное – определить правильный уровень для изменений: будете вы расширять текущий код или вынесете его во внешнюю функцию и вызовете её?

Вот яркий пример ненужного if:

Example

```
public static boolean isOdd(number) {
        if (number % 2 == 1) {
                return true;
        } else {
                return false;
        }
}
```

Его можно переписать без единого if:

Example

```
public static boolean isOdd(number) {
        return (number % 2 === 1);
}
```

Комментирование очевидных вещей

Большинство комментариев можно заменить названиями переменных и функций. Прежде, чем написать комментарий, вспомните об этом.

Например, такой код:

Example

```
//  на вход принимается 2 числа
public static int sum (int a, int b) {
 int c; //сумма чисел
 c=a+b; //операция суммирования
 return c;
}
```

Можно заменить таким:

Example

```
public static int sumTwoNumber(int numberOne, int numberTwo) {
        return (numberOne+numberTwo);
}
```

Но иногда прояснить код можно только комментарием. В таком случае сосредоточьтесь на вопросе: "ЗАЧЕМ нужен этот код", а не "ЧТО делает этот код". Вот пример кода, где комментарии только засоряют его:

Example

```
// create a variable and initialize it to 0
int sum = 0;
// Loop over array
array.forEach(
  // For each number in the array
 (number) => {
   // Add the current number to the sum variable
   sum += number;
 }
```

Не делайте так.

Не подвергать сомнению существующий код

Если вы не senior программист, который всегда работает в одиночку, то рано или поздно в своей жизни вы столкнетесь с тупым кодом. Новички склонны не замечать этого, тем более, если он исправно работает и давно написан.

Самое страшное в этой ситуации, что новичок может подумать, что это хороший код, и будет повторять существующие в нём неправильные подходы.

Иногда код может выглядеть плохо, потому что разработчик был вынужден написать его таким в силу объективных причин. В таком случае уместно оставить комментарии с описанием этих причин.

Новичкам можно посоветовать такое правило: любой недокументированный код, который вы не понимаете, возможно, плохой. Изучайте его. Спрашивайте о нём. Пользуйтесь командой git blame, выдающей автора каждой строки кода.

Если автор далеко или не может вспомнить свой код, изучайте его до тех пор, пока полностью не поймете. Только так вы сможете ответить на вопрос, плохой

код перед вами или хороший. Никогда не решайте этого наугад без твёрдых знаний.

Одержимость лучшими практиками

Термин "лучшие практики" – вредный, он ограничивает вас в исследовании, "ведь уже есть лучшая практика". "Лучших практик" не бывает. Бывают хорошие практики на сегодняшний день и для этого языка программирования.

Довольно часто то, что вчера считалось "лучшей практикой", сегодня считается плохой практикой. Вы всегда можете найти более лучшую практику, если потратите достаточно времени для этого. Поэтому не беспокойтесь о "лучших практиках", а сосредоточьтесь на том, что вы можете сделать хорошо.

Не делайте что-то, потому что где-то прочитали цитату об этом, или увидели, как кто-то делает это, или кто-то сказал про это "лучшая практика". Ставьте всё под сомнение, бросайте вызов всем теориям, знайте все возможные варианты и принимайте только обоснованные решения.

Изобретение колеса

В программировании зачастую полезно повторно изобретать колёса. Это довольно гибкая и стремительно меняющаяся сфера знаний. Никакая команда не может угнаться за всеми обновлениями и новыми требованиями.

Например, если нам нужно колесо, меняющее скорость вращения в зависимости от времени суток, возможно, вместо переделки обычного колеса, есть смысл переосмыслить его и сделать заново. Но если вам нужно колесо для его обычных целей, то, пожалуйста, не изобретайте его заново. Просто возьмите обычное колесо и используйте его.

Иногда бывает сложно выбрать нужное колесо из-за многообразия. Проводите исследование. Пробуйте перед покупкой. Большинство "программных колёс" бесплатны и с открытым кодом. По возможности используйте заготовки с открытым исходным кодом (open source), их легко отлаживать, улучшать, заменять и поддерживать.

В то же время, если вам нужно только колесо, не надо покупать целую машину и прикручивать эту машину к другой машине на место колеса. Не подключайте целую библиотеку ради одной-двух функций.

Глава 6. TestNG

TestNG (где NG – "next generation", т.е. "новое поколение") – это фреймворк для автоматизации тестирования, основанный на JUnit(Java) и NUnit (C#). Он может быть использован для:

- Unit-тестирования;
- Функционального тестирования;
- Интеграционного тестирования;
- end-to-end тестирования.

TestNg набрал высокую популярность среди Java-разработчиков за короткое время, и на данный момент – это один из наиболее распространенных тестовых фреймворков среди Java разработчиков.

Он использует Java аннотации для конфигурирования и написания тестовых методов.

Вот ряд функций, которые дополняют TestNg, по сравнению с JUnit 4:

- Дополнительные аннотации Before и After, такие как Before/After Suite и Before/After Group;
- Зависимые тесты;
- Группировка тестовых методов;
- Многопоточное выполнение;
- Встроенный фреймворк отчетности.

TestNG написан на Java и может быть использован как с языком Java, так и с другими основанными на Java языками (например, Groovy). В рассматриваемом нами тестовом фреймворке тесты и тестовые наборы настраиваются или описываются в основном через файлы XML.

По умолчанию файл называется testng.xml, но мы можем дать ему любое другое имя по нашему усмотрению.

TestNG позволяет пользователям создавать конфигурации тестов через XML-файлы и разрешает включать или исключать из них различные пакеты, классы и методы в тестовых наборах.

Он также дает возможность группировать тестовые методы в именованные группы и включать/исключать их из списка прогоняемых тестов.

6.1. Преимущества TestNG

Ниже приведены основные преимущества TestNG:

1. Множество опций у аннотаций Before и After.
2. Основанные на XML тестовые конфигурации и определения тестовых наборов.
3. Зависимые методы.
4. Группы/группы групп.
5. Зависимые группы.
6. Параметризация тестовых методов.
7. Data-Driven тестирование.
8. Многопоточное выполнение.
9. Улучшенная система отчетов.

6.2. Аннотации TestNG

@BeforeSuite

Метод с этой аннотацией будет выполнен до любых тестов, описанных в тестовом наборе testing.

@AfterSuite

Метод с такой аннотацией будет запущен после всех тестов из тестового набора TestNG.

@BeforeTest

Такие методы будут выполняться до каждой секции Test в наборе тестов.

@AterTest

Методы будут запускаться после каждой секции Test в тестовом наборе.

@BeforeGroups

Метод с аннотацией @BeforeGroup будет запускаться 1 раз до выполнения любого тестового метода указанной группы.

@AfterGroups

Такой метод будет запускаться 1 раз после запуска всех тестовых методов из указанной группы.

@BeforeClass

@BeforeClass помеченный метод выполняется однажды до запуска любого тестового метода в определенном тестовом классе.

@AfterClass

Такой метод запускается после выполнения всех тестовых методов тестового класса.

@BeforeMethod

Эти методы выполняются до запуска каждого тестового метода.

@AfterMethod

Такие методы запускаются после выполнения каждого тестового метода.

@DataProvider

Помечает метод, как предоставляющий данные для тестового метода. Этот метод возвращает в качестве данных двумерный массив Object [ ] [ ].

@Factory

Указывает, что метод является фабрикой, возвращающей массив объектов Object[ ]. Эти объекты будут в последствии использованы фреймворком TestNG как тестовые классы . Такой механизм нужен для запуска набора тест-кейсов с различными значениями.

@Listeners

Применимо к тестовым классам. Определяет массив тестовых классов прослушивателей, реализующих org.testng.ITestNGListener. Предназначена для отслеживания статуса выполнения и логирования.

@Parameters

Аннотация используется для передачи параметров тестовому методу. Эти значения параметров подставляются из файла конфигурации testing.xml во время выполнения.

@Test

Отмечает класс или метод как тестовый. Если используется на уровне класса, все public-методы этого класса помечаются как тестовые методы.

6.3. TestNG аннотации Before и After

Аннотации Before и After в основном используются для выполнения определенного кода до и после выполнения тестовых методов. Обычно это требуется для инициализации некоторых переменных или настройке перед запуском выполнения теста, последующей очистки вышеописанного после того, как выполнение теста закончится.

TestNG предоставляет 5 разных видов аннотаций Before и After, каждый из которых может быть использован в зависимости от требований тестирования.

Далее приведены различные виды Before и After поддерживаемые TestNG:

- @BeforeSuite и @AfterSuite
- @BeforeTest и @AfterTest
- @BeforeGroups и @AfterGroups
- @BeforeClass и @AfterClass
- @BeforeMethod и @AfterMethod

6.4. TestNG – как исключить метод из выполнения

При работе с TestNG бывают сценарии, когда может потребоваться исключить из выполнения часть теста или набор тестов. Например, какой-то серьезный баг в функции может мешать выполнению сценариям тестов, которые связаны с этой функцией. Так как мы в курсе этой проблемы, нам нужно отключить эти сценарии? чтобы они не выполнялись.

Отключения теста в TestNG можно добиться установкой атрибута enable аннотации @Test в значение false. Помеченный таким образом тестовый метод не будет запущен на проверку в тестовом наборе. Если этот атрибут установлен для аннотации Test на уровне класса, все public методы внутри класса будут отключены.

```
@Test( enabled=false )
```

6.5. TestNG – работа с таймаутами

Во время тестирования бывают случаи, когда отдельные тесты зависают или выполняются больше допустимого времени. В таких случаях имеет смысл отметить эти тесты проваленными, и продолжить тестирование. В этом разделе мы научимся устанавливать для тестов время выполнения, при превышении которого они будут проваливаться по таймауту.

TestNG позволяет задавать период времени для ожидания выполнения теста. Таймаут может быть задан двумя способами:

1. На уровне тестового набора: применяется ко всем тестам тестового набора TestNG.
2. На уровне отдельного тестового метода: будет применен к конкретному методу. Эта настройка имеет более высокий приоритет, чем общий таймаут на уровне тестового набора.

Для задания длительности таймаута необходимо использовать атрибут timeOutаннотации @Test:

```
@Test ( timeOut = 500 )
```

6.6. TestNG – примеры зависимых тестов

Зависимости – это возможность в TestNG, позволяющая тестовому методу зависеть от одного или группы тестовых методов. Благодаря этому, можно запустить на выполнение набор тестов до определенного другого метода. Зависимость метода работает только в том случае, если методы, от которых он зависит, являются частью того же класса или находятся в одном из базовых классов.

Тест с одним зависимым тестовым методом

Создадим пример тестового метода, который зависит от другого тестового метода в этом же классе.

Example

```
public class DependentTestExamples{
@Test(dependsOnMethods = { "testTwo" });
 public void testOne() {
  System.out.println("Test method one");
 }
 @Test
 public void testTwo() {
  System.out.println("Test method two");
 }
}
```

Вышеописанный класс содержит два тестовых метода, которые выводят в консоль свое имя во время выполнения. testOne метод зависит от тестового метода testTwo. Данная зависимость задана через настройки атрибута dependsOnMethods аннотации @Test.

Запустим тесты на выполнение:

Output

```
Test method two
Test method one
PASSED: testTwo
PASSED: testOne
```

Как видно из вывода на консоль, тестовый метод testTwo выполняется раньше, чем testOne. Это происходит из-за настроенной зависимости.

Тест с зависимостью от множества тестовых методов

Иногда может потребоваться установить зависимость тестового метода от нескольких других тестовых методов. Такая возможность существует в TestNG.

Example

```
public class DependentTestExamples{
@Test(dependsOnMethods = { "testTwo", "testThree" })
 public void testOne() {
  System.out.println("Test method one");
 }
 @Test
 public void testTwo() {
  System.out.println("Test method two");
 }
 @Test
 public void testThree() {
  System.out.println("Test method three");
 }
}
```

В примере выше в классе есть три тестовых метода, причем метод testOne зависит от testTwo и testThree.

Запустим этот пример на выполнение:

Output

```
Test method three
Test method two
Test method one
PASSED: testThree
PASSED: testTwo
PASSED: testOne
```

Как можно видеть из вывода на консоль, testTwo и testThree выполнились до testOne.

Зависимости и наследование

До этого момента мы рассматривали примеры, в которых зависимые тесты были частью одного класса. Зависимость от тестовых методов может быть

объявлена, только если эти тестовые методы принадлежат тому же классу, что и зависящий метод или любому из базовых классов его класса.

Рассмотрим, как TestNG выполняет тестовые методы, когда методы, от которых они зависят, являются частями базовых классов их классов:

Example

```
public class ParentClassTest{
@Test(dependsOnMethods = { "testTwo" })
 public void testOne() {
  System.out.println("Test method one");
 }
 @Test
 public void testTwo() {
  System.out.println("Test method two");
 }
}
public class DependentTestExamples extends ParentClassTest{
 @Test(dependsOnMethods = { "testOne" })
 public void testThree() {
  System.out.println("Test three method in Inherited test");
 }
 @Test
 public void testFour() {
  System.out.println("Test four method in Inherited test");
 }
}
```

Тестовый класс содержит четыре тестовых метода, выводящих на консоль свои имена во время выполнения.

Запустим тест на выполнение:

Output

```
Test four method in Inherited test
Test method two
Test method one
Test three method in Inherited test
PASSED: testFour
PASSED: testTwo
```

```
PASSED: testOne
PASSED: testThree
```

Судя по выводу в консоль, последовательность выполнения следующая: testFour – testTwo – testOne – testThree. testThree зависит от testOne и testTwo, TestNG выполняет все тестовые методы, от которых зависит метод, и в конце сам этот метод.

Тест, зависящий от группы

Зависимость от тестовых методов очень похожа на зависимость от групп. Но, в данном случае, до выполнения зависимого тестового метода выполняется группа тестовых методов.

Example

```
public class DependentTestExamples{

@Test(dependsOnGroups = { "test-group" })
 public void groupTestOne() {
   System.out.println("Group Test method one");
 }
@Test(groups = { "test-group" })
 public void groupTestTwo() {
   System.out.println("Group test method two");
 }
@Test(groups = { "test-group" })
 public void groupTestThree() {
  System.out.println("Group Test method three");
 }
}
```

Класс содержит три тестовых метода, методы groupTestTwo и groupTestThree объединены в группу test-group. Метод groupTestOne зависит от этой группы.

Запустим тесты.

Output

```
Group Test method three
Group test method two
Group Test method one
```

```
PASSED: groupTestThree
PASSED: groupTestTwo
PASSED: groupTestOne
```

Метод может зависеть от другого метода, только если тот находится в том же классе или в базовом классе для его класса, но ни в каких других классах. Если вам нужно, чтобы метод зависел от другого тестового метода, находящегося в отдельном классе, можно добавить тот тестовый метод-зависимость в группу, и для зависимого метода поставить зависимость от этой группы.

6.7. Assert в TestNG

Assert помогает проверять условия теста и принимать решения, когда тест провален или выполнен. Тест считается выполненным, только если завершается без вызова какого-либо исключения.

В приведенном далее коде мы проверяем соответствие title страницы значению "Google". Если соответствие не выполняется, тест проваливается.

Example

```
@Test
public void testCaseVerifyHomePage() {
  open("http://google.com&#187;);
 Assert.assertEquals("Google", getWebDriver().getTitle());
}
```

Здесь можно сделать также другую запись: Assert.assertEquals("Goooogle", getWebDriver().getTitle(), "Title not matching");

TestNG поддерживает механизм проверок с помощью класса Assert, эти проверки играют важную роль при тестировании приложения.

Теперь изменим заголовок страницы в коде и посмотрим на результат его выполнения:

Example

```
@Test
public void testCaseVerifyHomePage() {
  open("http://google.com&#187;);
  Assert.assertEquals("Goooooogle", getWebDriver().getTitle());
}
```

Этот код выдаст следующую ошибку проверки:

Output

```
java.lang.AssertionError: expected [Google] but found [Goooooogle]
```

Вариантов проверок в TestNG существует много. Далее будут перечислены самые популярные:

- assertEqual(String actual, String excepted): принимает два аргумента типа String и проверяет их на эквивалентность. В случае различий в аргументах, тест будет провален;

- assertEqual(String actual, String excepted, String message): принимает три аргумента и проверяет первые два на равенство. Если они не равны, тест будет провален, в консоль будет выдано сообщение из третьего аргумента;
- assertEquals(boolean actual, boolean excepted): принимает два аргумента типа boolean и проверяет их на равенство. Тест провален, если аргументы не равны;
- assertEquals(java.util.Collection actual, java.util.Collection excepted, java.lang.String message): принимает два объекта коллекций и проверяет, что они содержат одни и те же элементы в одном и том же порядке. В случае отличий, тест будет провален, и в консоль будет выдано сообщение из третьего аргумента;
- Assert.assertTrue(condition): принимает один аргумент boolean и проверяет что он имеет значение true. Если нет, генерируется AssertionError;
- Assert.assertTrue(condition, message): принимает один аргумент boolean и один String. Проверяет на истинность condition и, если оно не равно true, генерируется AssertionError, и в консоль выдается сообщение message;
- Assert.assertFalse(condition): принимает один аргумент boolean и проверяет что он имеет значение false. Если нет, генерируется AssertionError;
- Assert.assertFalse(condition, message): принимает один аргумент boolean и один String. Проверяет на истинность condition и, если оно не равно false, генерируется AssertionError, и в консоль выдается сообщение message.

Ниже расположен пример кода с проверками:

Example

```
@Test
public void testCaseVerifyHomePage() {
 open("http://google.com&#187;);
 Assert.assertEquals("Gooogle", getWebDriver().getTitle(), "Strings are not matching");
 //Напишите код для авторизации и методы isUserLoggedSuccessfully и isUserLoggedOut, возвращающие значения типа boolean.
 Assert.assertTrue(isUserLoggedSuccessfully(), "User failed to login");
 Assert.assertFalse(isUserLoggedOut()))
 }
}
```

Глава 7. Пример создания автоматизированного теста

Для написания автоматизированного теста необходимо прежде всего создать тест-кейс, написанный на естественном языке.

Для этого нужно выделить функцию и один конкретный кейс проверки с единственно верным ожидаемым результатом проверки.

Например, проверим функцию согласования документа.

Предусловия:

- В системе создан документ.
- Пользователь авторизован в системе.

Тест-кейс:

- Открыть карточку документа.
- Отправить текущий документ на согласование текущему пользователю.
- Согласовать документ.

Ожидаемый результат: документ имеет статус согласован

Постусловия:

- Удалить документ.

Каждый пункт проверки должен представлять собой один метод.

Каждый из пунктов тест-кейса будет являться шагом автоматизированного теста.

Перед созданием собственных методов стоит проверять наличие таковых в существующем проекте. В данном случае:

Предусловия и постусловия

Для создания предусловий используется WebAPI, в частности, метод WebAPI.create.DocForTest(String documentName);. Все предусловия и постусловия должны выполняться наиболее быстрым и валидным способом. В нашем случае за аксиому взято, что WebAPI работает корректно, т.к. исключает бизнес-логику и работу с отображением и обработкой на стороне клиента. Таким же способом мы удалим созданный нами в рамках теста документ, вызвав метод WebAPI.delete.DocForTest(random_string);

Шаги тест-кейса

Для выполнения шага "Открыть карточку документа" нужно перейти в карточку документа наиболее стабильным и быстрым способом, т.к этот шаг является промежуточным. Если внимательно посмотреть на сигнатуру метода WebAPI.create.DocForTest(String documentName);, можно заметить, что она возвращает прямую ссылку на созданный в этом методе документ. Нам осталось открыть её в браузере.

Для выполнения шага "Отправить текущий документ на согласование текущему пользователю" нужно кликнуть в тулбаре текущего документа на функцию отправки документа на согласование, выбрать текущего пользователя и согласиться на отправку. Сигнатура такого метода будет приблизительно такая – sendDocToApprove(String taskTheme, String executorUserName). Она явно указывает на то, что делает метод, инкапсулируя реализацию. Дословно – отправить документ на согласование с темой taskTheme пользователю executorUserName. Т.к этот метод принадлежит модулю Задачи, следует создать его именно в классе TaskHelp. Но, прежде, чем создать подобный метод, нужно проверить, что его не создал кто-то до Вас. В данном конкретном случае, такой метод уже имеется в проекте.

Для выполнения шага "Согласовать документ" нужно зайти в задачу согласования и выполнить её. Наименование задачи нам известно, следовательно, мы можем выполнить поиск в текущих задачах по имени, зайти в эту задачу и выполнить согласование. В данном случае, следуя алгоритму предыдущего шага, перед созданием нужно убедиться, что подобный функционал еще не реализован. Работа с документами должна находиться в модуле DocHelp, если выполнить поиск по методам данного класса, можно найти метод с такой сигнатурой: docApprovalConfirm(String themeTask), по наименованию видно, что данный метод выполняет согласование документ, а входные параметры подсказывают, что для его выполнения нужно наименование задачи. Из чего можно сделать вывод, что метод согласует документ с наименованием, которое мы в него передадим, что подтверждается его Javadoc (описание).

docApprovalConfirm

```
/**
    * Выполнить задачу согласования со статусом Согласован
    * @param themeTask Название задачи
*/
```

Для выполнения проверки важно понять, по каким критериям пользователи понимают, что функционал работает ожидаемо. В данном случае можно воспользоваться статусом документа, т.к. после согласования он изменится на "Согласован". Это работа с документами, значит, искать стоит в DocHelp, примерная сигнатура – checkDocumentStatus(String statusName), подобный метод уже есть и его можно использовать.

Таким образом тест будет выглядеть так:

Example

```
@Test(description = "Карточка документа: Согласование")
public void checkApproveDocument() {
   String documentName = createRandomString("doc_approve", 10); // создаём случайное название для документа
   String linkOnDoc = webAPI.create.DocForTest(random_string); // создаём документ и получаем ссылку на него


   MainHelp.auth(); //авторизуемся
   MainHelp.goTo(linkOnDoc); //переходим по прямой ссылке на документ
   DocHelp.sendDocToApprove(documentName, “admin”); //отправляем документ на согласование пользователю с логином “admin”
   DocHelp.docApprovalConfirm(documentName);//выполняем задачу согласование с наименование documentName
   MainHelp.goTo(linkOnDoc); //переходим по прямой ссылке на документ
   DocHelp. checkDocumentStatus(“Согласован”);//Проверяем, что статус документа “Согласован”


   webAPI.delete.DocForTest(documentName);//Удаляем тестовые данные
}
```

В данном случае методы, эмулирующие работу пользователей, уже были созданы. Рассмотрим их подробнее для понимания, что делать, если нужных в конкретном случае не будет. Стоит отметить, чт.е. дополнительный класс помощник – MainHelp. В нём хранятся методы, которые используются повсеместно и не привязаны к конкретному модулю системы. Например, авторизация или переход по ссылке с проверкой на стандартные ошибки.

MainHelp.auth()

```
   /**
    * Зайти в систему под админом
    */
   @Step("Авторизация под админом без пароля")
```

```
public static void auth() {
    auth("admin", "");
}
```

Этот метод является перегрузкой полного метода авторизации, т.к. очень часто нужна авторизация под пользователем Администратор.

Далее следует метод перехода по ссылке с проверкой на стандартные ошибки:

MainHelp.goTo(String url)

```
/**
 * Переход на новую страницу с проверкой на стандартные ошибки
 *
 * @param url relative url new page
 */
@Step("Переход на страницу ")
public static void goTo(String url) {
    inspection(); //Проверка на стандартные ошибки на странице
    if (url.contains("http://") || url.contains("https://")) { //Если url содержит протокол – он обсолютные
        open(url);
    } else { //иначе – относительный
        open(Configuration.baseUrl + url);
    }
    inspection(); //после перехода по ссылке и прогрузки странице, проверка на стандартные ошибки на странице
}
```

Остальные методы хранятся в классе DocHelp и построены аналогичным образом. Если функционал, специфичный для данного шага, он выполняется в текущем методе, иначе – выделяется отдельный общий метод, который потом вызывается в нужных местах. Таким образом, вызывая нужные методы, можно эмулировать любые пользовательские действия.