**Лабораторная работа №5**

**“Тестирование web-приложений с помощью Selenium”**

**Цель работы:** Научиться разрабатывать тесты для web-приложений с помощью Selenium

**Продолжительность работы** - 4 часа.

**Содержание**

## 1. Общие сведения о Selenium………….…..............................................................................1 2. Взаимодействие с UI………………………….. ………………………...............................3 3. Поиск элементов на странице..….........................................................................................3

## 4. Автоматизация тестирования web-приложений.................................................................5 5. Порядок выполнения лабораторной работы......................................................................13

## Общие сведения о Selenium

Selenium - это каркас для Web-тестирования, реализующий новый подход к валидации Web-приложений. В отличие от большинства инструментов для Web-тестирования, которые пытаются симулировать HTTP запросы, Selenium подходит к Web-тестированию так, как если бы он сам был браузером. При запуске автоматического теста Selenium, каркас запускает браузер и действительно проводит браузер через все шаги, намеченные в тесте, точно так же, как бы это делал пользователь, взаимодействуя с приложением.

Архитектура Selenium в действительности состоит из двух логических сущностей: проверочный код, написанный вами, и сервер Selenium, который помогает взаимодействовать с тестируемым приложением. Для успешного выполнения тестов необходимо наличие работающих экземпляров сервера Selenium и тестируемого приложения. Результат тестов, конечно, будет зависеть от того, насколько хорошо написано приложение.

## Selenium IDE 92

Selenium состоит из различных программных инструментов. Каждый из них имеет свою роль.

### Selenium 2 ( Selenium Webdriver)

Selenium 2 - будущее направление проекта и новейшее дополнение к инструментарию Selenium. Этот новый инструмент автоматизации предоставляет все виды удивительных особенностей, в том числе более связанный и объектно-ориентированный набор API, а также ответ на ограничения старой версии.

Selenium 2 является продуктом слияния Selenium и Webdriver. Он поддерживает Webdriver API и базовые технологии, а также Selenium 1 технологии под Webdriver API для максимальной гибкости в переносе тестов. Кроме того, Selenium 2работает с интерфейсом Selenium RC для обратной совместимости.

### Selenium 1 (Selenium RC or Remote Control)

Selenium RC был основным проектом в течение длительного времени, пока не произошло слияние с Webdriver, что привело к созданию нового и более мощного инструмента.

Selenium 1 по-прежнему активно используется и предоставляет некоторые особенности, которые недоступны в Selenium 2, включая поддержку нескольких языков программирования (Java, Javascript, Ruby, Python, Perl и C#) и совместимость почти со всеми браузерами.

### Selenium IDE

Selenium IDE (Integrated Development Environment) является инструментом для создания тестовых сценариев. Представляет собой плагин для браузера Firefox и обеспечивает легкий в использовании интерфейс для разработки автоматизированных тестов. Selenium IDE имеет функцию записи, которая позволяет записывать действия пользователя, как они выполняются, а затем экспортирует их в многоразовый код одного из языков программирования, который в последующем может быть выполнен.

### Selenium Grid

Selenium Grid позволяет масштабировать Selenium RC для больших наборов тестов и тестов, которые должны быть запущены в нескольких средах. Selenium Grid позволяет запускать тесты параллельно, то есть, различные тесты могут быть запущены одновременно на различных удалённых машинах. Это имеет два преимущества. Во-первых, если у вас есть большой набор тестов, или медленно выполняющиеся тесты, то вы можете существенно увеличить производительность с помощью Selenium Grid, разделив этот набор тестов на отдельные тесты, которые будут одновременно запускаться на разных удалённых машинах. Во-вторых, если вам надо запустить набор тестов в нескольких средах, то вы можете использовать отдельную удалённую машину для каждой среды. В обоих случаях Selenium Grid значительно уменьшает время, необходимое для запуска наборов тестов за счёт использования параллельных вычислений.

#### Взаимодействие с UI

Для имитации действий пользователя и воздействия на UI можно использовать Selenium. Начать писать интеграционные тесты с использованием Selenium очень просто. Нужно просто запустить SeleniumServer, который принимает входящие сообщения и транслирует их браузеру.

При конфигурировании Selenium Server нужно указать всего 4 параметра: адрес и порт selenium сервера, используемый браузер и адрес, по которому расположено тестируемое веб-приложение.

|  |
| --- |
| public class SeleniumExampleTests  {  [Fact]  public void SeleniumExample1()  {  ISelenium selenium = new DefaultSelenium("localhost",5561,"\*firefox","localhost:8099");  selenium.Start();  selenium.Open("/Account/LogOn");  selenium.WaitForPageToLoad("10000");  selenium.Type("Name","username");  selenium.Type("Password","12345");  selenium.Click("//input[@value='Войти']");  selenium.WaitForPageToLoad("10000");  Assert.True(selenium.IsTextPresent("Вход успешен!"));  selenium.Stop();  }  } |

Главные преимущества Selenium — это кроссбраузерность и поддержка многих языков. Можно изменить версию используемого браузера без перезапуска Selenium Server.

# Поиск элементов на странице

Для поиска элементов в Selenium используются локаторы. Локатор – это строка, уникально идентифицирующая элемент страницы. WebDriver предоставляет несколько способов использования локаторов для поиска элементов:

* **By.id** – в качестве локатора используется атрибут id элемента страницы;
* **By.name** – в качестве локатора используется атрибут name элемента страницы;
* **By.xpath** – используется для поиска элемента по XPath выражению;
* **By.tagName** – поиск по имени HTML тега;
* **By.className** – поиск по классу(атрибут class) элемента;
* **By.cssSelector** – данный тип локаторов основан на описаниях таблиц стилей (CSS);
* **By.linkText** – поиск ссылки с указанным текстом. Текст ссылки должен быть точным совпадением;
* **By.partionalLinkText** – поиск ссылки по части с указанным текстом.

В WebDriver реализованы два метода — findElement и findElements. В качестве параметра они принимают объект типа By, который задает критерий поиска.

* java.util.List **findElements(By by)** — находит все элементы в текущем контексте, идентифицированные классом By. На время его выполнения влияет [implicit wait](http://internetka.in.ua/selenium-driver-wait-until/) (неявные ожидания). При выполнении неявного ожидания, метод вернет список сразу же, как только будет найден хоть один элемент, или вернет пустой список, если время ожидания истекло.
* WebElement **findElement(By by)** — находит первый, удовлетворяющий условиям поиска, элемент на странице. На время выполнения этого метода также влияют неявные ожидания. Если элемент не найден сразу, то метод будет вызываться повторно пока не истечет время ожидания. Если же ничего найти не удалось, будет выдано исключение NoSuchElementException. Поэтому не используйте findElement для проверки отсутствия элементов, вместо этого можно использовать метод findElements и проверку на нулевое количество полученных элементов.

Примеры использования:

<div id="first">

<a class="red\_link">

<div class="label">Первая ссылка</div>

</a>

<a>Вторая ссылка</a>

</div>

WebElement element = driver.findElement(By.id("first"));

driver.findElement(By.tagName("a"));

driver.findElement(By.cssSelector("#first"));

driver.findElement(By.linkText("Вторая ссылка"));

Обратите внимание, что для первой ссылки использовать такой локатор нельзя. Метод findElement(By.linkText("Первая ссылка")) не найдет ни одного элемента. Потому, что искомый текст должен содержаться непосредственно в теге <a>. Первую ссылку можно получить используя, например, className или xpath:

driver.findElement(By.className("red\_link"));

driver.findElement(By.xpath(".//\*[text()='Первая ссылка']/.."));

Поисковые запросы можно выстраивать в цепочку:

<div id="first">

<a>Первая ссылка</a>

</div>

<div id="second">

**<a>Вторая ссылка</a>**

</div>

driver.findElement(By.id("second")).findElement(By.tagName("a"));

В тоже время найти такой элемент можно с помощью, например, XPath запроса. Что лучше использовать для такого поиска — сложный XPath/CSS запрос или цепочку простых запросов — решать только вам.

Сложный запрос получается более коротким и емким, и его можно предварительно отладить с помощью дополнительных [инструментальных средств](http://internetka.in.ua/xpath-start-part1/). Цепочки получаются достаточно длинными (чем сложнее запрос, тем длиннее будет цепочка) и их нельзя отладить перед использованием. Но есть и свои плюсы — если во время выполнения поиска не найден какой-либо элемент, цепочка более точно сообщит о том, что именно не нашлось.

# Автоматизация тестирования Web-приложений

Автоматизация тестирования – место встречи двух дисциплин: разработки и тестирования. Очевидно, что автоматизация тестирования имеет множество плюсов. Автоматическое решение:

1. Экономит время
2. Исключает человеческий фактор при тестировании
3. Снимает бремя рутинного регрессионного тестирования

Все, кто хоть раз занимался автоматизированным тестированиям, знают об оборотной стороне медали. Автоматические тесты могут быть:

1. Хрупкими и «ломаться» из-за изменения UI
2. Непонятными, содержать код «с душком»
3. Недостоверными: тестировать неверное поведение или зависеть от особенностей окружения

Для примера, рассмотрим следующий код. По названию видно, что мы тестируем то, что по запросу «gangam style» гугл выдаст YouTube-канал корейского популярного исполнителя PSY первым результатом.

[Test]

public void Google\_SearchGangnamStyle\_PsyYouTubeChanelIsOnTop()

{

var wd = new OpenQA.Selenium.Firefox.FirefoxDriver {Url = "http://google.com"};

try

{

wd.Navigate();

wd.FindElement(By.Id("gbqfq")).SendKeys("gangnam style");

wd.FindElement(By.Id("gbqfb")).Click();

var firstResult = new WebDriverWait(wd, TimeSpan.FromSeconds(10)).Until(

w => w.FindElement(By.CssSelector("h3.r>a")));

Assert.AreEqual("PSY - YouTube", firstResult.Text);

Assert.AreEqual("http://www.youtube.com/user/officialpsy", firstResult.GetAttribute("href"));

}

finally

{

wd.Quit();

}

}

В этом тесте очень много проблем:

1. Перемешаны слои приложения (драйвер, локаторы, результаты)
2. Строки зашиты в тесте
3. Для того, чтобы изменить веб-драйвер, например на IE придется менять все тесты
4. Локаторы зашиты в тесте и будут дублироваться в каждом тесте заново
5. Дублирование кода создания веб-драйвера
6. Assert не сопровожден сообщением об ошибке
7. Если первый Assert «упадет», то второе условие вовсе не будет проверено
8. При первом взгляде на тест не ясно, что в нем происходит, придется вчитываться и тратить время на понимание кода

**Если подходить к автоматизации «в лоб», вместо того, чтобы избавиться от рутинного повторения одинаковых действий, мы получим дополнительную головную боль с поддержкой тестов, ложными срабатываниями и спагетти-кодом в придачу.**

**Слои приложения в автоматизированном тестировании**

Ваши тесты – тоже код. Относитесь к ним также трепетно, как и коду в приложении. Тема слоев бизнес-приложений уже достаточно хорошо освещена. Какие слои можно выделить в тестах?

1. Технический драйвер (WebDriver, Selenium RC, etc)
2. Контекст тестирования (целевое окружение, пользователи, данные)
3. Абстракция UI – страницы, виджеты, компоненты страниц (PageObject pattern)
4. Тесты (тестовый фреймворк: NUnit, xUnit, MSTest)
5. DSL

Проведем эволюционный рефакторинг и исправим наш тест.

## Технический драйвер

## В нашем случае, это Selenium.WebDriver. Сам по себе WebDriver – не инструмент автоматизации тестирования, а лишь средство управления браузером. Мы могли бы автоматизировать тестирование на уровне HTTP-запросов и сэкономить кучу времени. Для тестирования веб-сервисов нам вообще не потребуется веб-драйвер: прокси вполне достаточно.

## Использование веб-драйвера хорошая идея потому что:

1. Современные приложения – больше, чем просто запрос-ответ. Сессии, куки, java-script, веб-сокеты. Все это может быть чертовски сложно повторить программным образом
2. Такое тестирование максимально приближено к поведению пользователя
3. Сложность написания кода гораздо ниже

К слою технического драйвера относятся:

1. Все настройки веб-драйвера
2. Логика создания и уничтожения веб-драйвера
3. Контроль ошибок

Для начала вынесем настройки в конфиг. У нас это выглядит так:

<driverConfiguration targetDriver="Firefox" width="1366" height="768" isRemote="false" screenshotDir="C:Screenshots" takeScreenshots="true"

remoteUrl="…"/>

Создадим отдельный класс, который возьмет на себя логику чтения конфига, создания и уничтожения веб-драйвера.

[Test]

public void WebDriverContextGoogle\_SearchGangnamStyle\_PsyYouTubeChanelIsOnTop()

{

var wdc = WebDriverContext.GetInstance();

try

{

var wd = wdc.WebDriver;

wd.Url = "http://google.com";

wd.Navigate();

wd.FindElement(By.Id("gbqfq")).SendKeys("gangnam style");

wd.FindElement(By.Id("gbqfb")).Click();

var firstResult = new WebDriverWait(wd, TimeSpan.FromSeconds(10)).Until(

w => w.FindElement(By.CssSelector("h3.r>a")));

var expected = new KeyValuePair<string, string>(

"PSY - YouTube",

"http://www.youtube.com/user/officialpsy");

var actual = new KeyValuePair<string, string>(

firstResult.Text,

firstResult.GetAttribute("href"));

Assert.AreEqual(expected, actual);

}

finally

{

wdc.Dispose();

}

}

Стало немного лучше. Теперь мы уверены, что всегда-будет использоваться только один веб-драйвер. Все настройки в конфиге, поэтому мы можем менять драйвер и другие настройки без перекомпиляции.

## Контекст тестирования

Для black-box тестирования приложения нам потребуется некоторое количество входных данных:

1. Целевое окружение – url, порты тестируемых приложений
2. Пользователи, с разным ролевым набором

Эта информация не относится к логике тестирования, поэтому мы вынесем это в конфигурационную секцию. Все окружения опишем в конфиге.

<environmentsConfiguration targetEnvironment="Google">

<environments>

<environment name="Google" app="GoogleWebSite">

<apps>

<app name="GoogleWebSite" url="http://google.com/" />

</apps>

<users>

<user name="Default" login="user" password="user" />

</users>

</environment>

</environmentsConfiguration>

Вместо wd.Url = «[google.com](http://google.com)»; стало wd.Url = EnvironmentsConfiguration.CurrentEnvironmentBaseUrl;

1. Мы не должны больше дублировать URL во всех тестах
2. Чтобы протестировать другое окружение достаточно собрать проект с другой конфигурацией и добавить трансформацию

<environmentsConfiguration targetEnvironment="Google-Test" xdt:Transform="SetAttributes">

## Page Objects

Паттерн [Page Objects](https://code.google.com/p/selenium/wiki/PageObjects) хорошо зарекомендовал себя в автоматизации тестирования.  
Основная идея – инкапсулировать логику поведения страницы в классе страницы. Таким образом, тесты будут работать не с низкоуровневым кодом технического драйвера, а с высокоуровневой абстракцией.

Основные преимущества Page Objects:

1. Разделение полномочий: вся «бизнес-логика» страницы должна помещаться в Page Objects, классы тестов лишь вызывают публичные методы и проверяют результат
2. DRY – все локаторы помещаются в одном месте. ~~Если~~ когда UI изменится, то мы изменим локатор лишь в одном месте
3. Скрытие слоя технического драйвера. Ваши тесты будут работать с высокоуровневой абстракцией. В будущем, возможно, вы захотите сменить драйвер: например, использовать PhantomJS, или вообще для каких-то участков отказаться от использования WebDriver, для улучшения производительности. В этом случае, вам придется заменить только код Page Objects. Тесты останутся неизменными
4. Page Objects позволяет записать локаторы в декларативном стиле

## Чего не хватает в Page Objects

Канонический паттерн предполагает создание одного класса на страницу вашего приложения. Это может быть неудобно в ряде случаев:

1. Кастомизируемый и/или динамически-изменяемый слой
2. Виджеты или иные элементы, присутствующие на многих страницах

Частично эти проблемы можно решить с помощью наследования, но агрегация видится предпочтительнее, как с технической точки зрения, так и c точки зрения понимания кода.

Поэтому лучше воспользоваться расширенной версией паттерна – Page Elements. Page Elements – позволяет дробить страницу на более мелкие составляющие – блоки, виджеты и т.д. После чего эти блоки можно переиспользовать в нескольких страницах.  
Создадим страницу:

[FindsBy(How = How.Id, Using = "gbqfq")]

public IWebElement SearchTextBox { get; set; }

[FindsBy(How = How.Id, Using = "gbqfb")]

public IWebElement SubmitButton { get; set; }

public GoogleSearchResults ResultsBlock { get; set; }

public void EnterSearchQuery(string query)

{

SearchTextBox.SendKeys(query);

}

public void Search()

{

SubmitButton.Click();

}

И «виджет» с результатами

public class GoogleSearchResults : PageElement

{

[FindsBy(How = How.CssSelector, Using = "h3.r>a")]

public IWebElement FirstLink { get; set; }

public KeyValuePair<string, string> FirstResult

{

get

{

var firstLink = PageHelper.WaitFor<GoogleSearchResults>(w => w.FirstLink);

return new KeyValuePair<string, string>(firstLink.Text, firstLink.GetAttribute("href"));

}

}

}

В NuGet есть пакет WebDriver.Support с прекрасным методом *PageFactory.InitElements.* Метод хорош, но имеет побочные эффекты. *PageFactory* из пакета WebDriver.Support возвращает прокси и не дожидается загрузки элемента. При этом, если все методы синхронизации работают с классом *By*, который пока не умеет работать с атрибутом *FindsBy*.

Эта проблема решается созданием базового класса Page.

/// <summary>

/// Get Page element instance by type

/// </summary>

/// <typeparam name="T">Page element type</typeparam>

/// <param name="waitUntilLoaded">Wait for element to be loaded or not. Default value is true</param>

/// <param name="timeout">Timeout in seconds. Default value=PageHelper.Timeout</param>

/// <returns>Page element instance</returns>

public T GetElement<T>(bool waitUntilLoaded = true, int timeout = PageHelper.Timeout)

where T : PageElement

/// <summary>

/// Wait for all IWebElement properies of page instance to be loaded.

/// </summary>

/// <param name="withElements">Wait all page elements to be loaded or just load page IWebElement properties</param>

/// <returns>this</returns>

public Page WaitUntilLoaded(bool withElements = true)

Для того, чтобы реализовать метод *WaitUntilLoaded* достаточно собрать все публичные свойства с атрибутами FindBy и воспользоваться классом WebDriverWait. Я опущу техническую реализацию этих методов. Важно, что на выходе мы получим простой и изящный код:

var positionsWidget = Page.GetElement<GoogleSearchResults>();

Остался последний неудобный случай. Существуют некоторые виджеты, которые скрывают/показывают часть элементов в зависимости от состояния. Разбивать такой виджет на несколько с одним свойством каждый – нецелесообразно.

Решение тоже нашлось:

public static IWebElement WaitFor<TPage>(

Expression<Func<TPage, IWebElement>> expression,

int timeout = Timeout)

var firstLink = PageHelper.WaitFor<GoogleSearchResults>(w => w.FirstLink);

Давайте посмотрим, как будет выглядеть код после рефакторинга:

[Test]

public void Google\_SearchGangnamStyle\_PsyYouTubeChanelIsOnTop()

{

try

{

var page = WebDriverContext.CreatePage<GooglePage>(EnvironmentsConfiguration.CurrentEnvironmentBaseUrl);

page.EnterSearchQuery("gangnam style");

page.Search();

var expected = new KeyValuePair<string, string>(

"PSY - YouTube",

"http://www.youtube.com/user/officialpsy");

var actual = page.GetElement<GoogleSearchResults>().FirstResult;

Assert.AreEqual(expected, actual);

}

finally

{

WebDriverContext.GetInstance().Dispose();

}

}

На этом этапе стало гораздо лучше:

1. Класс тестирования снял с себя управление драйвером и делегировал эти обязанности в класс страницы
2. Мы избавились от дублирования локаторов
3. Читаемость тестов улучшилась

## Тесты

После того, как мы вынесли локаторы и логику в Page Objects, код тестов стал лаконичнее и чище. Однако несколько вещей до сих пор не очень хороши:

1. Логика создания веб-драйвера дублируется из теста в тест
2. Логика создания страницы в каждом методе тоже избыточна
3. Магические строчки, «gangnam style», «PSY — YouTube», ”http://www.youtube.com/user/officialpsy” мозолят глаза
4. Сам сценарий теста достаточно хрупок: результаты индексации могут измениться и нам придется менять код

Создадим базовый класс тестов

public class WebDriverTestsBase<T> : TestsBase

where T:Page, new()

{

/// <summary>

/// Page object instance

/// </summary>

protected T Page { get; set; }

/// <summary>

/// Relative Url to target Page Object

/// </summary>

protected abstract string Url { get; }

[SetUp]

public virtual void SetUp()

{

WebDriverContext = WebDriverContext.GetInstance();

Page = Framework.Page.Create<T>(

WebDriverContext.WebDriver,

EnvironmentsConfiguration.CurrentEnvironmentBaseUrl,

Url,

PageElements);

}

[TearDown]

public virtual void TearDown()

{

if (WebDriverContext.HasInstance)

{

var instance = WebDriverContext.GetInstance();

instance.Dispose();

}

}

}

Перепишем тест еще раз

public class GoogleExampleTest : WebDriverTestsBase<GooglePage>

{

[Test]

public void Google\_SearchGangnamStyle\_PsyYouTubeChanelIsOnTop()

{

Page.EnterSearchQuery("gangnam style");

Page.Search();

var expected = new KeyValuePair<string, string>(

"PSY - YouTube",

"http://www.youtube.com/user/officialpsy");

var actual = Page.GetElement<GoogleSearchResults>().FirstResult;

Assert.AreEqual(expected, actual);

}

}

Уже почти идеально. Вынесем магические строки в атрибут TestCase и добавим комментарий к Assert’у

[TestCase("gangnam style", "PSY - YouTube", "http://www.youtube.com/user/officialpsy")]

public void Google\_SearchGoogle\_FirstResult(string query, string firstTitle, string firstLink)

{

Page.EnterSearchQuery(query);

Page.Search();

var expected = new KeyValuePair<string, string>(firstTitle, firstLink);

var actual = Page.ResultsBlock.FirstResult;

Assert.AreEqual(expected, actual, string.Format(

"{1} ({2}) is not top result for query "{0}"",

firstTitle, firstLink, query));

}

1. Код теста стал понятным
2. Повторяющиеся операции перенесены в базовый класс
3. Мы предоставили достаточно информации, в случае падения теста все будет понятно из логов тест-ранера
4. Можно добавить сколько угодно входных и выходных параметров без изменения кода теста с помощью атрибута TestCase

## DSL

В этом коде осталась две проблемы:

1. Код стал понятным и чистым, но чтобы его поддерживать в таком состоянии квалификация специалистов, поддерживающих тесты должна быть соответствующей
2. У отдела QA, скорее всего есть свой тест-план, а наши авто-тесты пока с ним никак не коррелируют
3. Часто одни и те же шаги повторяются сразу в нескольких сценариях. Избежать дублирования кода можно с помощью наследования и агрегации, но это уже кажется сложной задачей, особенно учитывая то, что порядок шагов может быть разным
4. *Google\_SearchGangnamStyle\_PsyYouTubeChanelIsOnTop()*: CamelCase трудночитаем

С помощью плагина [SpecFlow](http://www.specflow.org/specflownew/) , мы можем решить эти проблемы. SpecFlow позволяет записать тестовые сценарии в Given When Then стиле, а затем, автоматизировать их.

Feature: Google Search

As a user

I want to search in google

So that I can find relevent information

Scenario Outline: Search

Given I have opened Google main page

And I have entered <searchQuery>

When I press search button

Then the result is <title>, <url>

Examples:

|searchQuery |title |url

|gangnam style |PSY - YouTube |http://www.youtube.com/user/officialpsy

[Binding]

public class GoogleSearchSteps : WebDriverTestsBase<GooglePage>

{

[Given("I have opened Google main page")]

public void OpenGooglePage()

{

// Page is already created on SetUp, so that's ok

}

[Given(@"I have entered (.\*)")]

public void EnterQuery(string searchQuery)

{

Page.EnterSearchQuery(searchQuery);

}

[When("I press search button")]

public void PressSearchButton()

{

Page.Search();

}

[Then("the result is (.\*), (.\*)")]

public void CheckResults(string title, string href)

{

var actual = Page.GetElement<GoogleSearchResults>().FirstResult;

var expected = new KeyValuePair<string, string>(title, href);

Assert.AreEqual(expected, actual);

}

}

Таким образом:

1. Каждый шаг можно реализовывать лишь однажды
2. Атрибуты Given When Then поддерживают регулярные вырадения – можно создавать повторно использующиеся «функциональные» шаги
3. QA-отдел может записывать сценарии в проектах авто-тестов
4. Тестировщики могут писать DSL, а автоматизацию можно поручить программистам
5. В любой момент времени отчет о пройденных тестах, а значит и количестве разработанного функционала, доступен на CI-сервере

**Порядок выполнения лабораторной работы**

### Выбрать web-ресурс, включающий в себя форму авторизации, форум и страницу поиска. Разработать и протестировать для него следующие сценарии с помощью Selenium:

### Тестирование регистрации на форуме

### Тестирование телефонного справочника

* Тестирование системы голосования
* Тестирование поиска

Необходимо использовать разные браузеры. Сценарии могут быть выбраны другие, но они должны быть согласованы с преподавателем.

### Пример тестирования поиска:

1. Зайти на сайт https://miet.ru/search/.
2. Проверить наличие поля поиска
3. Проверить наличие текста на странице “Поиск по сайту”.
4. Ввести в строке поиска заведомо бессмысленный набор букв.
5. Нажать кнопку “Поиск”.
6. Проверить наличие на странице текста “К сожалению, на ваш поисковый запрос ничего не найдено”.
7. Ввести в строке поиска слово “Кожухов”.
8. Нажать кнопку “Поиск”.
9. Проверить наличие таблицы результатов
10. Проверить наличие текста во всех ячейках таблицы.
11. Проверить наличие на странице текста “Кожухов Игорь Борисович” в категории “Люди”.
12. Закрыть окно браузера.

**Ссылки:**

1. <http://www.interface.ru/home.asp?artId=8834>
2. <http://korzh.net/2010-12-seleniumin-tegracionnoe-testirovanie-veb-prilozhenij.html>
3. http://internetka.in.ua/selenium-webdriver-findelement-by/
4. http://www.pvsm.ru/razrabotka/33334/print/