Tallinna Ülikool
Informaatika Instituut

Automaattestimine Rahvusarhiivi veebirakenduse näitel

Automated Testing of Web Applications. The Case of the National Archives of Estonia

Bakalaurusetöö

Autor: Antonina Jarmonova
Juhendaja: Inga Petuhhov

Autor: ……"……" …… 2010.a.
Juhendaja: ……"……" …… 2010.a.
Instituudi direktor: ……"……" …… 2010.a.

Tallinn 2010

Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

....
(kuupäev) (autor)

Содержание

Введение

Автоматическое тестирование, как средство экономии времени и денег, оказывает сильное влияние на успешность реализации программного продукта (ПО) в целом, а потому широко используется для достижения высоких потребительских качеств ПО. Поскольку сам автор работает тестировщиком, то ему часто приходится заниматься автоматизацией тестирования. Основная трудность, с которой он сталкивается перед началом каждого нового проекта по разработке ПО - это выбор средства для автоматизации. Кажется, что такого рода сложности должны изжить себя по мере накопления автором опыта работы с тем или иным инструментом тестирования. Отчасти это так, но не до конца. Проблему представляют быстро развивающиеся инфотехнологии, в том числе средства для разработки ПО. Потребности людей в части использования ИТ[1]-услуг все возрастают. Растут и требования, предъявляемые к свойствам, которым должно отвечать разрабатываемое ПО. В погоне за обеспечением лучшего качества ПО разработчики применяют все более новые подходы. Конечно, ни для кого не секрет, что ИТ-сфера развивается очень стремительно. Автор заостряет внимание на этом факте единственно для того, чтобы продемонстрировать истоки проблемы: каждый новый проект - это комбинированная смесь новейших технологий, поддержка которых инструментами автоматизации не гарантирована. Выяснить, как на самом деле поведет себя тестовый фреймворк[2] при тестировании ПО можно единственным способом - проверить на практике. То есть пока не попробуешь, в результатах нельзя быть уверенным.

Поэтому в данной работе на примере веб-приложения для Национального Архива (*Rahvusarhiiv*) автор будет проводить автоматическое тестирование отдельных участков системы, на которых используются различные технологии отображения данных. Цель – установить степень поддержки этих технологий специально выбранным фреймворком для автоматизации.

Первая часть (главы с первой по четвертую) содержит теоретическую основу данной работы, в которой приводится информация общего характера о тестировании ПО, его значении и классификации, о принципах работы веб-приложений и -технологий, а также о

[1] ИТ - инфотехнологии
[2] Фреймворк - каркас или структура программной системы

существующих подходах к автоматизации тестирования, в часности тестирования веб-приложний.

После того, как будет сделан выбор в пользу использования конкретного средства для автоматизации тестирования (пятая глава), автор ознакомится со способами применения и заявленными возможностями этого инструмента (шестая глава).

Детальное описание тестируемого объекта в седьмой главе имеет своей целью, основываясь на анализе используемых технологий и особенностей их применения, установить необходимую глубину проводимого автоматического тестирования. Это даст возможность правильно составить сценарии тестирования таким образом, чтобы они охватывали все имеющиеся критерии оценки возможностей тестового фреймворка относительно данного приложения.

По результатам проведенного в восьмой главе тестирования будет сделано заключение о том, в какой степени можно автоматизировать тестирование веб-приложения для Национального Архива с помощью выбранного фреймворка. Можно ли обойтись одним лишь выбранным фреймворком для того, чтобы в достаточном мере покрыть функциональность приложения автоматическими тестами? Или же трудности с тестированием каких-либо технологий вынудят продолжить поиск более подходящего фреймворка? А если так, то следует ли заменить лишь некоторые тесты, проводимые на вызывающих сложности местах, или заменить все тесты полностью, таким образом отказавшись от использования выбранного инструмента для тестирования данного веб-приложения? Результаты тестирования должны помочь ответить на все поставленные здесь вопросы.

1 Тестирования ПО и его значение

Для того, чтобы любой типичный программный продукт мог считаться завершенным, он должен быть на должном уровне протестирован. Тестирование - это неотъемлемая часть разработки ПО. С тестированием ПО (*Software testing*) часто используются понятия обеспечение качества (*Quality Assurance*) и контроль качества (*Quality Control*). Иногда эти понятия используются как синонимы одного и того же. На самом деле они имеют разные значения. Автор считает важным правильное понимание их значений, так как это позволяет найти верный подход к процессу тестирования.

Ниже приводятся определения этих значений больше не для того, чтобы дать их исчерпывающее значение, а для того, чтобы обозначить границы исследуемой темы в этой работе.

Обеспечение качества (QA) – совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации ПО информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения качества выпускаемого продукта. (Булат, 2008a)

Контроль качества (QC) – совокупность действий проводимых над объектом тестирования в процессе разработки для получения информации об актуальном состоянии объекта тестирования в разрезах: "готовность Продукта к выпуску", "Соответствие зафиксированным требованиям", "Соответствие заявленному уровню качества продукта". (Булат, 2008a)

И, наконец, **тестированием** называют одну из техник контроля качества, включающую в себя активности по планированию работ, проектированию тестов, выполнению тестирования и анализу полученных результатов на соответствие поставленным требованиям. (Булат, 2008a) Целью тестирования является выявление как можно большего числа дефектов и их исправление.

Тестирование позволяет повысить качество и надежность ПО, таким образом, обеспечивая его высокие потребительские свойства. Такой продукт имеет большую конкурентноспособность на рынке ПО.

В общем процессе обеспечения качества место тестирования структурно изображено на рисунке ниже (см. Рисунок 1). (Булат, 2008b)

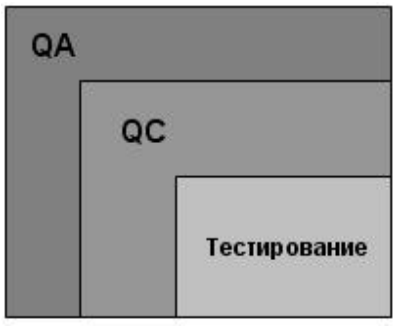


Рисунок 1. Место тестирования в процессе обеспечения качества

Процесс тестирования достаточно сложен и требует уделения значительных средств и внимания. Часто, однако, уровень проводимого тестирования определяется сроками реализации продукта, что отодвигает вопросы качества на второй план. Также при недостаточном финансировании или его урезании разработчики порой склонны рассматривать процесс тестирования скорее как нужное, но всё же дополнение к разработке ПО. Результатом такого пренебрежения может стать выявление множества непредвидимых проблем на разных этапах развития ПО, что не может не повлиять как на затраты по исправлению, так и на сроки его реализации.

По данным, опубликованным американским национальным институтом стандартов NIST (*National Istitute of Standards and Technology*), около 60% дефектов обнаруживается в процессе тестирования и около 21% уже непосредственно при эксплуатации. При этом стоимость исправления дефектов быстро растет по мере продвижения ПО к стадии эксплуатации. Так стоимость исправления дефекта после ввода системы в эксплуатацию вдвое превышает аналогичную стоимость на стадии тестирования продукта, и более чем в тысячу раз – в период выработки требований к продукту. (NIST, 2002) Поэтому правильно организованный процесс тестирования – залог успешного завершения разработки ПО.

1.1 Классификация тестирования

Классификация тестирования имеет достаточно широкую интерпретацию и может производится по разным признакам, таким как: виды тестирования, уровни, методы, техники и т.д. Приведенная ниже классификация преставлена в объеме, необходимом для дальнейшего исследавания темы данной работы.

1.1.1 Виды тестирования

В зависимости от преследуемых целей, по видам тестирование можно условно разделить на три группы (Булат, 2007а):

1. **Функциональное тестирование** – рассматривает внешнее поведение системы на соответствие требованиям (техническим заданиям, спецификациям[3], различным другим проектным документам). По сути, это тестирование действий и реакции на них, например, нажатие на кнопки, активизация гиперссылок и так далее.
2. **Нефункциональное тестирование** – проверяет не то "что" система должна делать, а "как" она работает. Конкретно под нефункциональными свойствами системы подразумевается производительность, расширяемость, надежность, удобство пользования и т.д.
3. **Связанное с изменениями** – проводится после внесения каких-либо изменений, будь то исправление ошибки или доработка каких-то новых возможностей. Для подтверждения работоспособности каждой новой версии ПО или правильности осуществленного исправления дефекта обычно проводятся следующие виды тестирования:
    - Дымовое тестирование (*Smoke Testing*) направлено на поверхностную проверку всех модулей приложения на предмет работоспособности и наличие быстро находимых критических и блокирующих дефектов.
    - Регрессионное тестирование (*Regression Testing*) направлено на проверку изменений, сделанных в приложении или окружающей среде. Дает гарантию того, что изменения в новой версии приложения не повредили уже существующую функциональность.

1.1.2 Уровни тестирования

Тестирование можно также производить по различных уровнях. Уровень тестирования определяет то, над чем производятся тесты: над отдельным модулем, группой модулей или системой, в целом. По уровням тестирование делится на:

- компонентное или модульное (*Component/Unit testing*)

[3] Спецификация - документ с набором требований и параметров, которым должно удовлетворять ПО.

- интеграционное (*Integration testing*)
- системное (*System testing*)
- приемочное (*Acceptance testing*) это тестирование на соответствие системы требованиям пользователя. По результатам приемочного тестирования выносится решение о принятии или непринятии приложения заказчиком. Приемочное тестирование проводится в случае достижения продукта необходимого уровня качества. В случае если заказчик не удовлетворен результатами тестирования, приложение отправляется на доработку. (Morgan, Samaroo, Thompson & Williams, 2007)

1.1.3 Методы тестирования

По методам тестирование можно разделить на тестирование "белого" и "черного" ящиков.

1. При методе "черного ящика" (*black-box testing*) тестируемое приложение рассматривается как "черный ящик", внутреннее наполнение которого неизвестно. Тестировщик имеет доступ только к входным и выходным данным. Проследить же за процессом обработки этих данных он не имеет возможности. Поэтому главным оценочным критерием является спецификация, которая позволяет делать выводы, соответствуют ли полученные данные ожидаемому результату или нет.
2. При тестировании методом "белого ящика" (*white-box testing*) тестировщик работает непосредственно с исходным кодом, опираясь на знание внутренней структуры системы при разрабатке тестовых сценариев. Данные для тестирования определяются путем изучения логики системы, без учета системных требований. При тестировании "белым ящиком" основное внимание уделяется коду системы. Чем большее покрытие кода[4] было достигнуто, тем более полно выполнено тестирование.

1.1.4 Техники тестирования

Техники тестирования сильно различаются по своей специфике, которую могут определять, например направленность на тестирование конкретного объекта или на

[4] Покрытие кода - процентный показатель насколько исходный код программы был протестирован.

проверку определенного результата и т.д. По SWEBOK[5] (*Software Engineering Body of Knowledge*) техники делятся на следующие группы и подгруппы (SWEBOK, 2004):

- Техники, базирующиеся на интуиции и опыте тестировщика. В случае использования данной техники тестирование проводиться либо без определённого плана и документации - специализированное тестирование (*ad hoc testing*), либо без заранее определенного плана методом свободного поиска - исследовательское тестирование (*exploratory testing*).
- Техники, базирующиеся на спецификации. Данные техники объединяет наличие определенного плана действий или анализа данных. Сюда относятся анализ граничных значений (*boundary-value analysis*), эквивалентное разделение (*equivalence partitioning*) и другие.
- Техники, ориентированные на код - основаны на анализе исходного кода. Тестами предусматриваются возможные сценарии развития событий, исходя из покрытия всех условий и решений блок-схемы кода (*control-flow based*). Либо тесты основываются на проигрывании потока данных (*data-flow based*) - прогоняются всевозможные тестовые данные (комбинации различных исходных значений) для определенного случая или сценария. Эти тесты называются управляющие данными (*data-driven tests*).
- Тестирование, ориентированное на дефекты. Подобные техники ориентированы на специфические категории ошибок.
- Техники, базирующиеся на условиях использования.
- Техники, базирующиеся на природе приложения. Определенная технологическая или архитектурная природа приложения может требовать специфичной техники тестирования, которая важна именно для заданного типа приложения. Сюда же входит веб-ориентированное тестирование.
- Выбор и комбинация различных техник.

[5] SWEBOK - документ, подготавливаемый комитетом Software Engineering Coordinating Committee, в который вовлечено сообщество IEEE Computer Society.

1.1.5 Сценарии тестирования

Все вышеописанные техники тестирования (за исключением *Ad hoc testing*) предполагают наличие определенного плана действий. Пошагово задокументированный план действий называется тестовым сценарием (*Test Case*), который разрабатывается на основе имеющихся системных требований, спецификации и прочей документации. В тестовых сценариях записываются проделываемые шаги, конкретные условия и параметры, необходимые для проверки реализации тестируемой функции или её части. Структура сценария тестирования может варьироваться, но обычно имеет следующий вид:

1. Исходные условия
2. Описание производимых действий/событий
3. Ожидаемый результат
4. Полученный результат

В процессе реализации ПО, предъявляемые требования могут по-разным причинам меняться, как это часто и бывает. Поэтому сценарии тестирования также приходится часто обновлять.

2 Веб-приложения и используемые для их отображения технологии

Качество тестирования веб-приложений напрямую зависит от степени осведомленности тестировщика о принципе их работы и технологиях, применяемых для их отображения.

Веб-приложение - это клиент-серверное приложение, главной особенностью которого является то, что взаимодействие с пользователем происходит посредством веб-интерфейса, в роли которого выступает веб-браузер, тем самым освобождая приложение от этой обязанности. Стандартизированная архитектура интерфайса веб-браузера позволяет осуществлять все коммуникации с приложением по стандартному протоколу HTTP. Совершаемые пользователем действия обрабатываются браузером, и тот, в случае необходимости, обращается с запросом к приложению. Приложение, в свою очередь, создает и передает в виде ответа браузеру новую или обновленную HTML-страницу, работу с которой продолжает пользователь.

Но если раньше сайты были не более чем парой связанных между собой HTML-страниц, то теперь это полноценные приложения, способные выполнять сложные задачи по обработке данных и получению информации. Это способствовало появлению целого ряда инструментов для разработки веб-приложений, которые помогают решать часто возникающие проблемы и задачи.

2.1 AJAX

При обмене данными между приложением и веб-браузером часто бывает, что обновлению подлежит лишь какая-то часть HTML-страницы. Поэтому для быстроты и удобства страница не перезагружается полностью, а обновляются лишь ее отдельные данные, то есть обновление происходит динамически. Для обеспечения подобного подхода к построению веб-страниц используются специальные технологии, которые позволяют осуществлять так называемый "фоновый" обмен данными.

Одной из таких технологий является AJAX (от англ. *Asynchronous Javascript* and XML — "асинхронный *JavaScript* и XML"), в основе работы которого лежит использование *JavaScript* и асинхронной отправки запросов на сервер без перезагрузки веб-станицы.

Благодаря использованию AJAX трафик при работе с веб-приложением значительно сокращается, так как вместо загрузки всей страницы достаточно загрузить только

изменившуюся часть, часто довольно небольшую. Это позволяет снизить нагрузку на сервер и ускорить реакцию интерфейса. (Маклафлин, 2005)

2.2 Tapestry

Tapestry - это популярный *Apache*[6] фреймворк с открытым исходным кодом для разработки динамичных, надежных и масштабируемых веб-приложений на *Java*. *Tapestry* позволяет разработчику абстрагироватся от написания HTML-кода и всего что с ним связано и оперировать компонентами. Генерацией HTML-страниц занимается сам *Tapestry*. (Apache Tapestry, n.d.)

[6] Apache Software Foundation — организация-фонд, состоитоящий из разработчиков-энтузиастов.

3 Автоматическое тестирование

Помимо того, что осуществлять тестирование вручную скучно и выматывающе, как и вся рутинная работа, такой способ еще и крайне ненадежен. Ведь вероятность допущения тестировщиком ошибки возрастает с каждым последующим повторением тестового сценария. А если разрабатываемое приложение требует проведения систематических регрессионных и дымовых тестов, то вряд ли тестировщик может гарантировать, что все учел и ничего не упустил из вида. Поэтому для повышения эффективности тестирования имеет смысл до какой-то степени автоматизировать работу тестировщика. Автоматическое тестирование подразумевает использование специальных инструментов для выполнения тестов и проверки результатов выполнения в автоматическом режиме для экономии времени и увеличения надежности.

3.1 Целесообразность автоматизации

При принятии решения о целесообразности автоматизации тестирования приложения следует ответить на следующие вопросы:

1. Возможна ли автоматизация в контексте данного приложения? Используемые приложением технологии могут иметь специфичные свойства, которые могут препятствовать автоматизации тестирования в той или иной степени. Например, использование протокола HTTPS, который ограничивает в целях безопасности доступ к приложению, и многие инструменты для тестирования не могут это обойти. (Булат, 2007с)
2. Нужна ли автоматизация? Не всегда затраты как в денежном, так и временном эквиваленте на автоматизацию тестирования окупаются. Поэтому следует учитывать, что:
    - Затраты на поддержку автоматизации могут быть очень высоки. Например, находящееся на начальной стадии развития приложение подвержено частым и сильным изменениям. В этом случае, на поддержку составленных автоматических тестов обновленными может уходить много рессурсов.
    - Процесс разработки автоматизированных тестов может быть довольно сложным. Чем специфичнее и сложнее тестируемое приложение, в смысле

используемых технологий и замысловатости функциональности, тем сложнее организовать автоматическое тестирование.

- Стоимость инструмента для автоматизации может быть достаточно высока. Свободно распространяемые инструменты, как правило, отличаются более скромным функционалом и меньшим удобством работы. Поэтому в случае с небольшим приложением и простыми тестовыми сценариями, ручное тестирование может выглядеть более привлекательно. (Булат, 2007b)

3. В какой степени нужно автоматизировать тестирование? Невозможно полностью автоматизировать тестирование, как бы тестировщик не старался. Есть вещи, которые можно сделать только вручную. Поэтому автоматизировать нужно, прежде всего, те тесты, которые хорошо этому поддаются. Это, прежде всего:

   - регрессионные и дымовые тесты
   - нагрузочные тесты[7]
   - приемочные тесты
   - тесты, управляющие данными.

   Если приложение небольшое и систематическое проведение регрессионных тестов не нужно, то и для автоматизации тестирования меньше причин.

3.2 Минусы и плюсы автоматизации

Преимущества автоматизации тестирования:

- Экономия времени на выполнение сценариев (проигрывание тестов).
- Надежность – при каждом запуске тесты будут выполнены в полном объеме согласно заданному сценарию, то есть, исключен "человеческий фактор". Тестировщик ничего не пропустит по неосторожности и ничего не напутает в результатах.
- Частота проведения – позволяет выполнять тесты много чаще.

[7] Нагрузочное тесты - тесты на производительность

- Присутствие тестировщика не требуется – во время выполнения тестов тестировщик может заниматься другими полезными делами, или тесты могут выполняться в нерабочее время.
- Избавляет тестировщика от рутинной работы.

Недостатки автоматизации тестирования:

- Ниже уровень интеллектуалности проводимых тестов – все написанные тесты всегда будут выполняться однообразно. Это одновременно является и недостатком, так как тестировщик, выполняя тест вручную, может обратить внимание на некоторые детали и, проведя несколько дополнительных операций, найти дефект. Программа действует по заданному скрипту и соответственно сделать этого не может.
- Мелкие ошибки и неточности останутся пропущенными – автоматически прогоняемый скрипт может быть недостаточно тонко настроен и пропускать мелкие ошибки. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки в тех элементах, с которыми не осуществляется взаимодействие во время выполнения скрипта.
- Требует более высокого уровня квалификации – некоторые средства для автоматизации тестирования очень сложны в использовании. Создание тестовых сценариев требует от тестировщика высокого уровня компетентности или каких-то специфичных навыков.
- Автоматические тесты следует постоянно обновлять. (Булат, 2007b)

4 Существующие подходы к автоматическому тестированию веб-приложений

Обычно функциональое тестирование веб-приложения с клиентской стороны проводится методом "черного ящика", то есть сводится к вводу данных и анализу полученного результата. Четкая формализация интерфейса с помощью HTML-разметки посредством браузера является основой для автоматизации тестирования.

На данный момент имеется широкий выбор средств, позволяющих автоматизировать тестирование веб-приложений. Большинство из них используют так называемый "*Capture & Playback*" подход. Его суть заключается в том, что тестовые скрипты записываются автоматически на основе работы пользователя с тестируемым приложением. Инструмент перехватывает действия пользователя, а также реакцию браузера на эти действия и записывает их в тесты пошагово – как действие и ожидаемый результат. При последующем проигрывании этих тестов инструмент автоматически воспроизводит ранее записанные действия и сравнивает получемые результаты с ожидаемыми. Точность сравнения может настраиваться. Можно также добавлять дополнительные проверки – задавать условия на свойства объектов (цвет, расположение, размер и т.д.) или на функциональность приложения (содержимое сообщения и т.д.). Многие инструменты тестирования, основанные на этом подходе, хранят записанные действия и ожидаемый результат в некотором внутреннем представлении, используя или распространенный язык программирования, или собственный язык. Кроме элементов интерфейса, инструменты могут оперировать HTTP-запросами, последовательность которых также может записываться при работе пользователя, а затем модифицироваться и воспроизводиться. (Сортов & Хорошилов, 2004)

5 Выбор средства для автоматизации тестирования

Выбор инструмента напрямую зависит от специфики тестируемого веб-приложения и предъявляемых требований по отношению к тестовым сценариям, т.к. инструменты тестирования не могут поддерживать абсолютно все технологии, используемые при разработке приложений. Зачастую многие из средств поддерживают тестирование определенных приложений не в полном объеме, а лишь до какой-то степени. Это значит, что выбор инструмента сводится к банальному методу проб и ошибок, при котором задачей тестировщика является выяснение также степени поддержки заявленной технологии. То есть, до какого уровня программа способна адекватно воспринимать тестируемую технолгию.

Для автоматизации тестирования имеется широкий выбор средств. Ниже приведены средства для автоматизации функционального тестирования веб-приложений, с которыми автор так или иначе имел дело ранее.

1. **Canoo WebTest** – это бесплатный *Apache 2.0* лицензированный продукт. Особенность его работы с тестируемым веб-приложением заключается в эмуляции поведения браузера через HTTP запросы. Это позволяет существенно повысить скорость выполнения тестов. Но в тоже время, именно потому, что работа с браузером лишь симулируется, проигрывание находящегося на странице *Javascript* и плохо структурированного HTML-кода является известной проблемой данного инструмента. (Canoo WebTest, n.d.)
2. **TestComplete** разработан компанией *AutomatedQA* и представляет собой законченную систему для автоматизации тестирования веб-приложений, веб-сервисов, *.Net*, *Visual C++*, *Visual Basic*, *Delphi*, *C++Builder* и *Java*-приложений. Позволяет работать с *ActiveX* [8] компонентами. (HP QuickTest Professional software, n.d.)
3. **Selenium** – это бесплатный продукт для проведения приемочного тестирования. *Selenium* работает непосредственно с браузером, т.е. нет эмуляции поведения браузера, что должно гарантировать полную адекватность проводимого тестирования. Он включает в себя несколько инструментов, каждый из которых имеет свои особенности и область применения. (Selenim HQ, n.d.)

[8] Active X - технология управления мултимедийными данными на странице.

4. **Rational Functional Tester** – платный продукт, разработанный компанией IBM и ориентированный на тестирование различных *Java-*, *.Net-* и веб-приложений. Имеет широкие возможности для интеграции со средами разработки. (IBM, n.d.)
5. **QuickTest Professional (QTP)** разработан корпорацией *Hewlett Packard (*HP*)* и относится к обьединенной в общий центр *Mercury Quality Center* линейке продуктов по обеспечению управления процессами QC для широкого диапазона приложений. QTP использует концепцию управления ключевыми словами (*keyword-driven testing*). Это позволяет упростить создание и поддержку автоматизированных тестов. При подобном подходе тестировщик имеет полный доступ к свойствам нижележащих тестов и объектов через интегрированную среду разработки и отладки, которая полностью синхронизирована с *Keyword View*. (Automated QA, n.d.)
6. **Borland SilkTest** выпускается компанией *Borland* в составе линейки коммерческих продуктов *Application Lifecycle Management*. *SilkTest* предназначен для автоматизации тестирования веб-, *Java-*, *.Net-* и клиент-серверных приложений через графический интерфейс пользователя, благодаря чему достигается тестирование методов работы конечного пользователя в полном объеме. (Borland, n.d.)

В данной работе автор будет проводить автоматическое тестирование с помощью тестового фреймворка *Selenium*. Выбор в пользу данного фреймворка обусловлен, прежде всего, его использоваением по месту работы автора. Процесс автоматизации тестирования в фирме, которой работает автор, начинается именно с него, поскольку *Selenium* предоставляет достаточно широкие возможности по автоматизации, и в тоже время распространяется бесплатно в рамках лицензии *Apache*.

6 Детальное описание Selenium

Разработка *Selenium* началась в *ThoughtWorks* в 2004 году с создания "*JavaScriptTestRunner*" как ядра системы для проигрывания тестов, которые могли оперировать с содержащимся на странице *JavaScript*. Далее к имеющемуся прототипу добавили поддержку нескольких языков и серверную составляющую. Целью было создание средства автоматизации приемочного тестирования веб-приложений, которое могло бы работать с *JavaScript* так, как это обычно делает реальный браузер. (Selenim HQ, n.d.)

6.1 Selenium Core

Selenium Core - это ядро для запуска тестовых скриптов, на котором основывается вся линейка инструментов *Selenium*. *Selenium Core* написан на чистом DHTML / *JavaScript* (по-сути, это набор динамический страниц и *JavaScript* сценариев), что дает гибкость использования во всех браузерах с поддержкой *JavaScript*. *Core* поддерживает множество операционных систем и браузеров, подстраиваясь под специфику работы каждого конкретного браузера. Это позволяет выявлять проблемы совместимости между браузерами. (Поляруш, 2009)

Selenium работает непосредственно через браузер из соседнего модального окна[9] (*iframe*), обходя возможные ограничения по доступу с одного домена на другой (*cross-domain security restrictions*). Для этого *Selenium* должен быть инсталлирован на один сервер с тестируемым приложением. Это можно отнести к минусам *Selenium Core*, поскольку инсталлировать *Selenium* на веб-сервер не всегда возможно (Selenim HQ, n.d.). Наглядно принцип работы *Selenium Core* можно видеть ниже (см. Рисунок 2):

[9] Модальное окно - контейнер внутри HTML-документа, куда загружаются любые другие независимые документы.

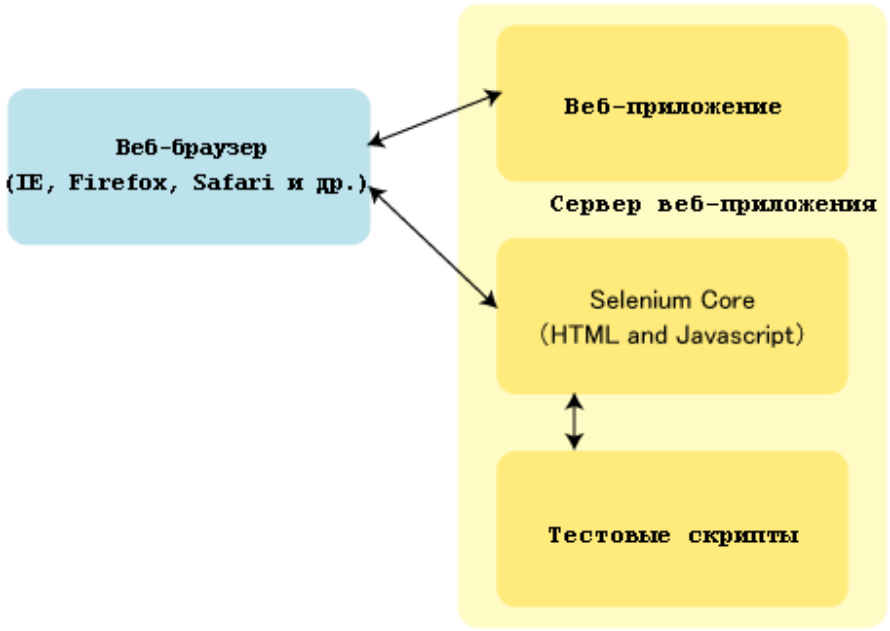


Рисунок 2. Принцип работы *Selenium Core* (Поляруш, 2009)

Тесты могут быть реализованы на одном из поддерживаемых языков программирования или на собственном языке *Selenese*, который пишется в фомрате xHTML[10] и имеют пошаговую структуру (см. Таблица 1):

Таблица 1. Структура шагов записанного теста

Первая команда	Цель	Значение
Вторая команда	Цель	Значение

Исходный код на *Selenese* выглядит примерно так (см. Пример кода 1):

```
<table border="1">
 <tr>
   <td>Первая команда</td>
   <td> Цель</td>
   <td>Значение</td>
 </tr>
 <tr>
   <td>Вторая команда</td>
   <td>Цель</td>
   <td>Значение</td>
```

[10] xHTML — Расширяемый язык разметки веб-страниц, созданный на базе XML.

```
  </tr>
</table>
```

Пример кода 1. Таблица шагов представленная на *Selenese*

Готовые тесты запускаются либо с консоли (в режиме "*driven*"), либо с помощью специального проигрывателя *TestRunner*. *TestRunner* имеет графический интерфейс с настройками для запуска тестов, поэтому более удобен. К таким настройкам относится регулировка скорости выполнения: от быстрой, почти незаметной глазу, до скорости работы обычного пользователя.

При проигрывании конкретного теста *TestRunner* отображает его пошаговое описание в следующем виде (см. Таблица 2):

Таблица 2. Проигрываемый тест в виде таблицы шагов

open selenium page			// Название проигрываемого теста
open	/myPage		// Открыть страницу */myPage*
type	nameField	selenium	//Заполнить поле *nameField* значением "*selenium*"
click	submitButton		//Нажать на кнопку *submitButton*
verifyTitle	Selenium web application testing system		//Проверить соответствие заголовка значению "*Selenium web application testing system*"

По завершению проигрывания тестов выводится отчет, сколько тестов и команд прошли или не прошли тестирование. Успешно выполненные шаги окрашены в зеленый цвет, не прошедшие тестирование подсвечиваются красным. В поле невыполненного шага выводится пояснение (см. Рисунок 3).

open selenium page		
open	/myPage	
type	nameField	selenium
click	submitButton	
verifyTitle	Selenium web application testing system	Actual value 'selenium - Google otsing' did not match 'Selenium web application testing system'

Рисунок 3. Пошаговый отчет проигранного теста

6.2 Selenium IDE

Главной функцией IDE является автоматическая запись тестов. IDE реализован в виде плагина[11] к *Firefox*. Для его использования наличие *Selenium Core* не требуется. Также с помощью IDE сохраненные тесты можно проигрывать, но только на *Firefox*.

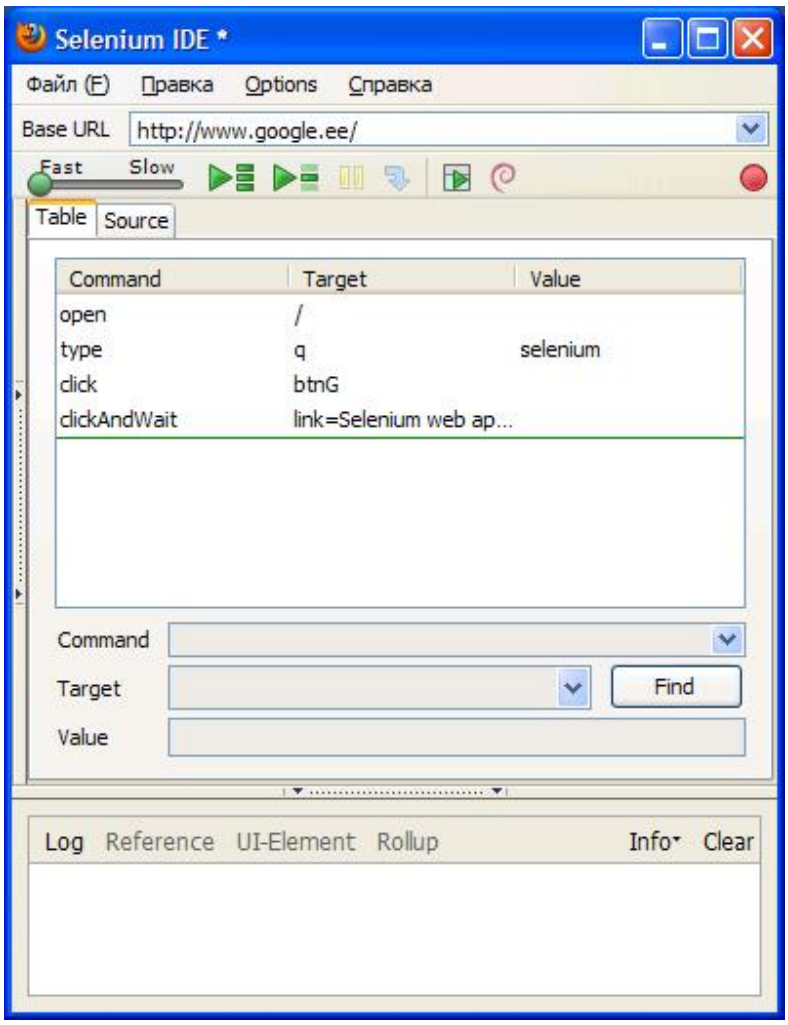


Рисунок 4. *Selenium IDE* интерфейс

При нажатии кнопки записи, IDE отслеживает действия пользователя, связанные с навигацией и вводом данных в браузере и записывает в виде скрипта *Selenese*. Пользователь может видеть произведенные действия сразу же в виде таблицы, состоящей из команд, как и в случае с HTML-таблицами. Наглядность записываемых команд

[11] Плагин — независимо компилируемый программный модуль, динамически подключаемый к основной программе, предназначенный для расширения и/или использования её возможностей.

помогает пользователю лучше контролировать запись, а также производить модификации в процессе записи теста (см. Рисунок 4).

Каждая команда записывается в отдельную строку и имеет три колонки. Первая из них является действием или проверкой (*action* или *check*). Вторая – элементом (*target*), к которому применяется команда, или по-другому цель. И третья – ожидаемым значением (*value*).

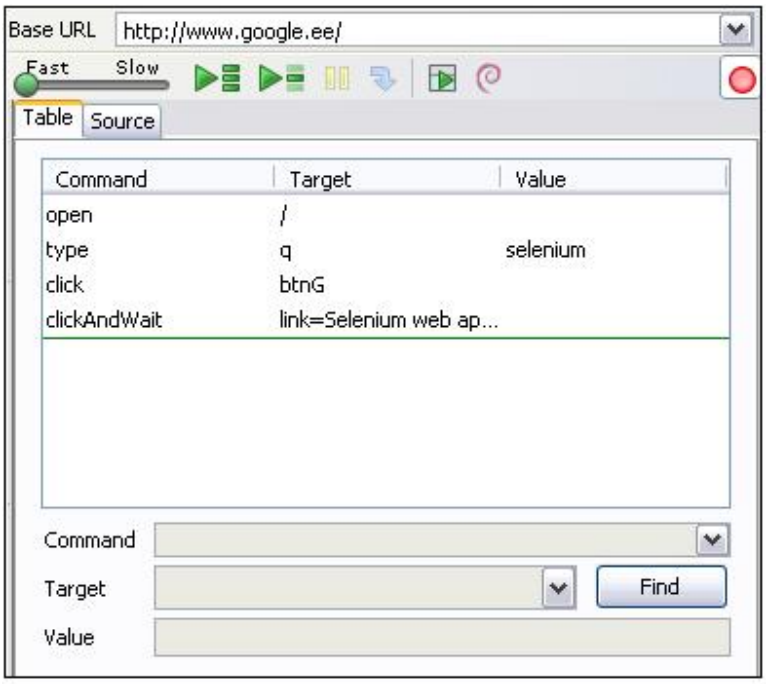


Рисунок 5. Записанные шаги в представлении *Selenium IDE*

Изображенный выше тест (см. Рисунок 5) записан в автоматическом режиме и его команды интерпритируются следующим образом (см. Таблица 3):

Таблица 3. Пример записанного в *Selenium IDE* теста

Command	Target	Value	Пояснения
open	/		//открыть станицу (т. к. базовый URL: *http://www.google.ee/* указан выше, то значение *Target* здесь просто "/")
type	q	selenium	//ввести в поле для поиска "*q*" (имя поля) значение "*selenium*"

click	btnG		//нажать кнопку "*btnG*" (имя кнопки). Значение кнопки - "*Google otsing*".
clickAndWait	Link=Selenium web ...		//нажать на ссылку со значением *"Selenium web ap..."* и ждать загрузки страницы.

6.2.1 Команды

Команды можно условно поделить на две группы - действия (*actions*) и проверки (*checks*):

- **Действия** (*actions*) имитируют взаимодействие пользователя с веб-приложением. С их помощью *Selenium* управляет браузером. Это, к примеру, нажатие на кнопку или ввод текста в поле. Имеется достаточно широкий набор действий, вроде *open*, *click*, *type*, *select* и т.д., а также более уточняющего характера: *OpenWindow*, *OpenWindowAndWait* и т.д.
- **Проверки** (*checks*) устанавливают соответствие ожидаемых и реальных результатов выполнения команд. Это может быть проверка на наличие определенного элемента, его расположения или состояния. Сам IDE их не записывает. Поэтому такого рода команды добавляются позже вручную. Проверка наиболее полно представлена двумя типами – *assert*(подтвердить) и *verify*(удостовериться). Разница состоит в том, что при не прохождении *assertSomething* команды выполнение теста прерывается, а при провале *verifySomething* команды выдается сообщение об ошибке, но тест продолжает работу (если есть такая возможность). Примеры проверок выглядят следующим образом: *verifyLocation* / *assertLocation*, *verifyTitle* / *assertTitle*, *verifyValue* / *assertValue* и т.д.

При выборе той или иной команды внизу окна появляется информация об использовании данной команды: ее сфера применения, синтаксис, особенности и т.д. Также полный список и примеры использования команд можно увидеть в документации на официальном сайте.

6.2.2 Цели

Указание цели как обьекта, над которым совершается действие, может быть произведено несколькими способами. Цель может идентифицироватся либо через названия всякого

рода свойств HTML-элемента (*id*, *name*, *identifier*, *class*, *dom*, *value* и другие), либо через локализацию элемента посредством *xPath*[12], *link* и другие.

Такой выбор обусловлен, прежде всего, отсутствием четкой структуризации HTML-кода. У элемента, который требуется идентифицировать, может отсутствовать как *id*, так и все остальные отличительные признаки, к которым можно было бы привязать идентификацию. Или же, идентификаторы могут присутствовать, но подвергаться частым изменениям. Ярким примером служит использование при разработке *JSF*-фреймворка[13], который сам отвечает за создание HTML-кода и, соответственно, за присвоение сгенерированным HTML-элементам идентификационных номеров (*id*). Эти *id*-номера присваиваются автоматически и обновляются каждый раз с внесением изменений в структуру кода. В данном случае, с каждой новой версией пришлось бы переписывать тесты заново, что сводит на ноль все преимущества автоматизированного теста. Также нельзя положиться на изпользование только *xPath*, в силу тех же самых аргументов: HTML-код подвержен частым изменениям при постоянной интеграции. Поэтому структура HTML-страницы может также измениться, и соответственно путь нахождения элемента.

Именно поэтому описание местонахождения объекта на странице и его свойств часто комбинируются для создания наиболее стабильной цели. При автоматической записи теста *Selenium* конечно сам идентифицирует объект более удобным для него способом. Например, если имеется *id* или *name*, то прописав в поле "*Target*" лишь значение этого *id*. Если пользователя это не устраивает, то *Selenium* также оставляет возможность выбрать другой более подходящий для данного случая идентификатор.

[12] XPath — язык запросов к элементам XML-документа.
[13] JavaServer Faces — фреймворк для разработки веб-приложений на Java.

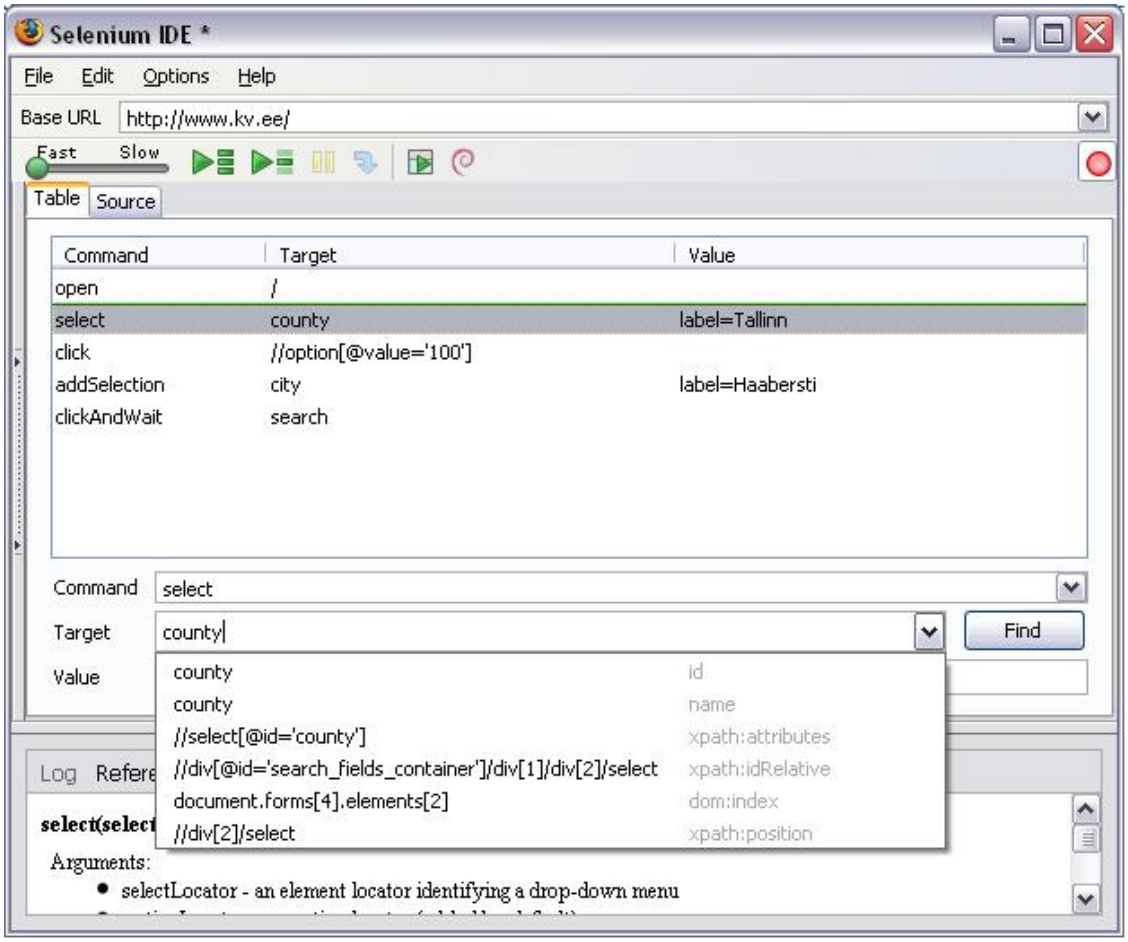


Рисунок 6. Варианты идентификаторов, предоставляемые *Selenium IDE*

На примере выше (см. Рисунок 6) вторая команда означает: выбрать из *select*-списка, *id* которого равно "*county*", значение "*Tallinn*". В тоже время для выбора предоставляется еще целый список возможных значений, причем далеко не всех. Так певыми вариантами идут *id* и *name*, в данном случае они идентичны. Далее следуют:

– *xpath:attributes*, значение которого *//select[@id='county']* расшифровывается следующим образом: найти select-элемент, id которого соответствует *'country'*.

– *xpath:idRelative*, значение которого *//div[@id='search_fields_container']/div[1]/div[2]/select* расшифровывается следующим образом: найти *select*-элемент, который находится в *//div[1]/div[1]/div[2]/* по *xPath*. Плюс *id*-номер корневого *div*-а у взятого *xPath*, должен соответствовать значению *'search_fields_container'*.

– *dom:index*, значение которого *document.forms[4].elements[2]* расшифровывается следующим образом: найти 3-ий элемент DOM[14]-модели документа в 5-ой форме.

– *xpath:position*, значение которого *//div[2]/select* расшифровывается следующим образом: найти по-позиции *select* во втором *div*-элементе.

6.2.3 Конвертация теста в язык программирования

При переходе на "*Source*" закладку можно видеть составленный тест на уже описанном выше *Selenese* языке с таблицей действий (см. Дополнение 1).

Можно конвертировать также имеющийся HTML в один из языков программирования, поддерживаемых *Selenium* (см. Рисунок 7).

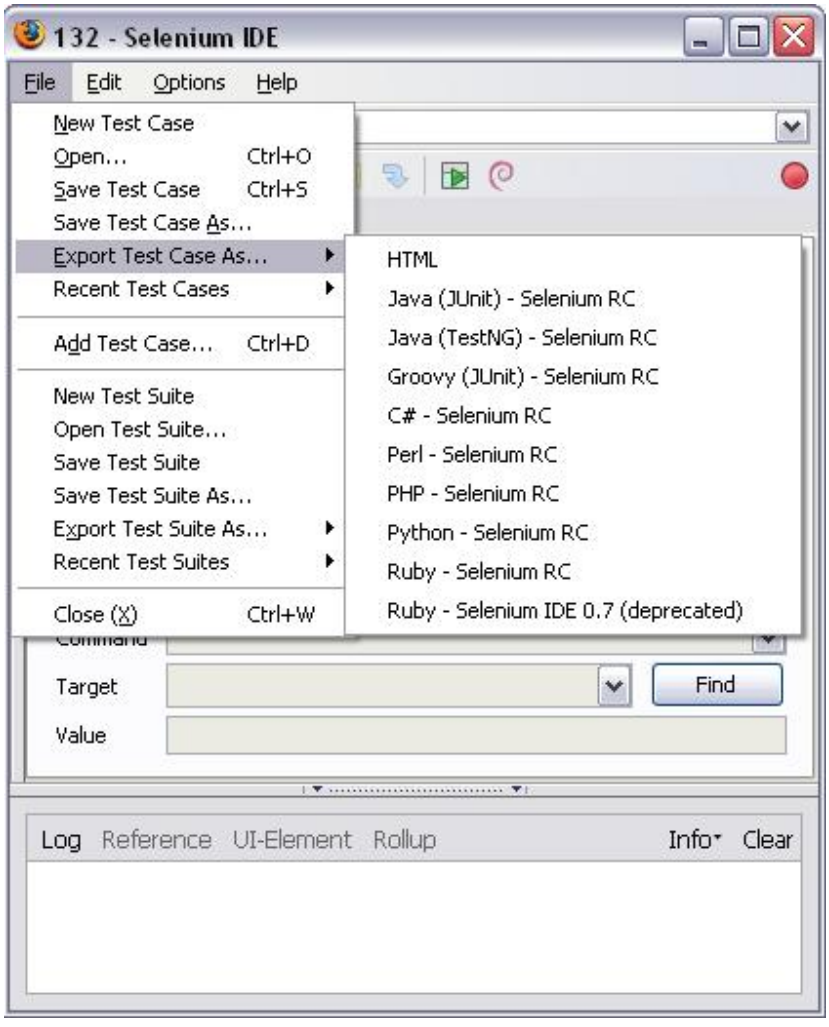


Рисунок 7. Конвертация заисанного теста в язык программирования

[14] Document Object Model — программный интерфейс для доступа к документам.

Итак, с помощью IDE можно составить тест на одном из выбранных языков программирования, не владея навыками программирования. Эта функция является очень важной для использования IDE в совокупности с другими продуктами из линейки Selenium.

Существенным минусом является невозможность создания сложных сценариев тестирования: не поддерживается в полной мере управление потоками данных (*flow control*). То есть, согласно информации с форума поддержки, для IDE было создано расширение для управления потоками данных (*flowControl extension*), но как им пользоваться осталось для автора неясным. (OpenQA, 2009).

6.3 Selenium Remote Control

Selenium Remote Control (RC) является написанным на *Java* HTTP-сервером, который проигрывает записанные тесты на различных браузерах. Архитектура *Selenium Server* в действительности состоит из двух логических частей: *Selenium Core* и прокси-сервер[15], который помогает взаимодействовать с тестируемым приложением. Управление RC можно производить посредством ввода команд с серверной консоли либо через удаленный интерфейс. Принцип работы RC изображен ниже (см. Рисунок 8):

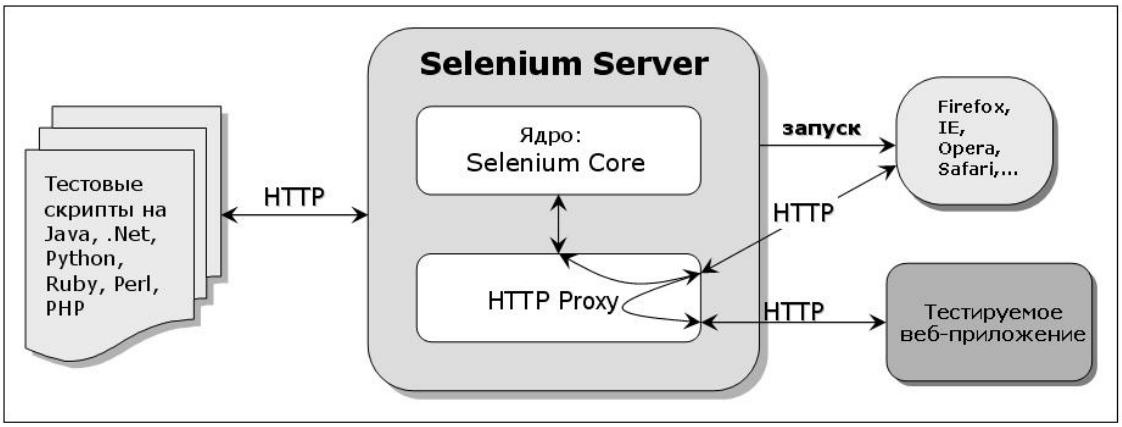


Рисунок 8. Принцип работы *Selenium RC (*Инструменты для автоматизации функционального тестирования, 2006)

[15] Прокси -сервер — служба сети, позволяющая клиентам выполнять косвенные запросы к другим сетевым службам.

Selenium RC работает исключительно на уровне *Javascript*. При запуске *Selenium RC* прописывает себя в настойках как прокси-сервер. Эта настройка, собственно, и есть все отличие в поведении браузера, запущенного через *Selenium*-сервер. При начале работы *Selenium*-сервер открывает новую сессию, в которой указывается тип браузера (**iexplore*, **firefox*, **opera* и т.п.) и URL, с которого этот браузер начнет работу. Открытая сессия будет использоваться далее при последующих запросах. Работая как прокси, *Selenium*-сервер перехватывает все URL, начинающиеся с */selenium-server/*, и отсылает свои HTML-старницы. Таким образом, *Selenium Remote Control* решает проблему инсталляции на веб-сервер, но проблема *cross-domain security restriction* по-прежнему актуальна. Переход браузера с одного домена на другой во время одной сессии обычно ломает тест. (Кантор, 2008)

Также RC может интегрироватся с различными системами тестирования из семейства *xUnit*[16]: *JUnit* и *TestNG*, и имеет поддержку следующих языков программирования: *Java*, *.Net*, *Ruby*, *Perl*, *Python*, *PHP*. В совокупности это дает большие возможности для создания более гибких тестов, с поддержкой *flow control* в полной мере. Конечно, использование данных возможностей *Seleniuma* имеют один существенный минус: тестировщик должен обладать достаточно высокой квалификацией, чтобы работать с кодом. Поэтому создание теста на IDE и его последующая конвертация в один из поддерживаемых языков программирования, здесь будет очень в помощь тестировщику.

Самый очевидный сценарий использования совместных возможностей IDE - RC выглядит следующим образом:

1. Автоматическая запись теста на IDE.
2. Редактирование записанного теста вручную под свои нужды с помощью предоставляемых IDE возможностей.
3. Экспорт записанного теста в нужный язык программирования. В случае с *Java*, можно даже выбрать тестовый фреймворк: *JUnit* или *TestNG* - под который экспортируемый тест будет подогнан. То есть содержать соответсвуюшие классы, методы и т.д.
4. Импорт полученного *.java* файла (в случае *Java* языка) в тестовый фреймворк.

[16] xUnit - тестовые фреймворки, которые ориентированы на тестирвание кода.

5. Работа с тестом уже непосредтсвенно в тестовом фреймворке.

6.3.1 Создание управляющих данными тестов

Создание управляющих данными тестов для автоматического тестирования требует от тестировщика достаточно высокой квалификации.

Принцип тестирования следующий: создается определенный сценарий тестирования, который в дальнейшем прогоняется с множеством всевозможных значений исходных данных. Например, требуется протестировать правильность валидации полей, что может в определенных случаях представлять проблему. Конечно, если поле одно, то быстрее и дешевле это будет сделать вручную. А если их несколько, и требуется протестировать поведение системы со всевозможными комбинациями наборов значений, и не единожды, а при каждой новой версии? То есть, это как раз отличное место для автоматизации.

Технически процесс автоматического проведения управляющих данными тестов не очень отличается от всего выше описанного. Главным отличием является наличие внешнего источника данных, откуда будут браться значения для проигрывания теста. Это может быть *Excel* файл с табличкой значений, которая должна быть определенным образом структурированна, чтобы тест мог правильно их интерпретировать. Возможность проведения на *Selenium* управляющих данными тестов в силу его интеграции с тестовыми фреймворками является большим плюсом.

7 Описание проводимого тестирования

Тестирование будет проводится на реальном веб-приложении, находящемся в процессе разработки. Веб-приложение разрабатывается для Национального Архива и должно обеспечивать электронную архивацию имеющихся на сохранении матерьялов: документов, фильмов, изображений и пр.

Данное веб-приложение разрабатывается с применением новейших технологий. Построение внутренней архитектуры продиктовано правилами разработки приложений под фреймворк *Tapestry 5*, который был выбран для саздания данного приложения. Само назначение данного приложения подразумевает тесную работу с файлами, в первую очередь их загрузку и поддержку асинхронных процессов. Для загрузки содержимого страницы при навигации по приложению широко используется AJAX и модальные окна.

7.1 Охват и критерии оценки тестирования

Наиболее верный способ дать адекватную оценку тому, насколько выбранный фреймворк подходит для тестировния приложения Национального Архива – это создать минимальный набор тестов, затрагивающий основные технологические решения, применяемые в данном веб-приложнии. Поэтому охват производимого тестирования должен включать в себя проверку следующих возможностей:

1. **Навигация** в рамках веб-приложения. Это одна из самых простых функциональных возможностей приложения. Тем не менее, учитывая использование таких технологий как Tapestry при создании HTML-кода, проверка навигации по приложению является неоходимой.
2. Поддержка **AJAX**. Насколько эффективно *Selenium* справляется с тестированием элементов, выполненных с использованием AJAX.
3. Поддержка **модальных окон** (*iframe*). Модальное окно создается с помощью тега *<IFRAME>*, и по-сути является контейнером внутри обычного документа, который позволяет загружать в область заданных размеров любые другие независимые документы. Насколько эффективно *Selenium* справляется с тестированием функциональности, выполняющейся в модальных окнах.

4. **Загрузка файлов** (*file upload*). Поскольку специфика проекта напрямую связана с загрузкой файлов извне, то способность *Selenium* автоматически ее воспроизводить, также является важным критерием.

5. Поддержка **асинхронных процессов**. При асинхронном запросе браузер не ждет ответа от сервера. После отсылки запроса, приложение продолжает свою работу в интерактивном режиме. Сервер тем временем обрабатывает запрос, и посылает ответ инициатору запроса. Это позволяет повысить быстродейтсвие приложения. Но потенциально может не поддерживаться или поддерживаться не в достаточной мере тестовым фреймворком.

6. Проведение ***управляющих данными тестов***. Насколько эффективно можно проводить на *Selenium* тестирование таких тестов.

7. Способность *Seleniuma* выполнять записанные тесты относительно **браузеров *Firefox*** и ***Internet Explorer***. Целью будет проверить способность *Seleniuma* проигрывать тесты на этих браузерах без необходимости для тестировщика дополнительно модифицировать тесты. Насколько *Selenium* способен самостоятельно подгонять тесты под определенные особенности браузера.

Критериями оценки тестирования будет служить способность *Selenium* производить тестирование с перечисленными выше случаями. То есть, способен ли *Selenium* впринципе справлятся с тестирование функциональности, реализованной с помощью конкретной технологии и относительно определенного браузера, и насколько эффективно он это делает.

7.2 Описание проводимых тестов

Тесты разработаны автором на основе описания рабочих процессов обычного пользователя данной системы. Тесты должны покрыть все пункты, описанные в главе 7.1. Проводимые тесты нацелены на проверку возможностей *Selenium*. Это значит, что, например, полученные результаты поиска будут проверятся не на соответствие заданным критериям, а на наличие результата, то есть смог ли *Selenium* выполнить эту команду.

Выполнение тестов пошаговое: действие - ожидаемый результат (проверка).

7.2.1 Тест 1— Создание нового проекта

Ниже представлено пошаговое описание создания нового проекта, начиная с самого начала: от входа в систему и до подтверждения сохранения проекта. В этом тесте описывается создание проекта самым оптимальным способом с минимальным количеством шагов (т. е. заполняются только обязательные поля и т.д).

Конкретно данный тест направлен на тестирование навигации и использование модальных окон.

Начальные условия: Пользователь находится на стартовой странице приложения.

Шаги выполнения теста:

1. Залогиниться в систему, нажав на кнопку "*Logi sisse*" (см. Рисунок 9). (По-существу, это просто переход со станицы на страницу, поскольку авторизация еще не реализована).

   Проверить: должна загрузиться глвная страница.

   Рисунок 9. Загрузочная страница приложения: Зайти в систему

2. Нажать на кнопку "*Koosta uus projekt*" для создания нового проекта (переход на форму для добаления проекта) (см. Дополнение 2).

   Проверить: должна загрузиться форма для добавления проекта.

3. Нажать ссылку "*Vali...*" для заполнения первого обязательного поля. Ссылка запускает *Javascript*, который открывает *iframe*-окно, в котором производится поиск возможных значений(имен) для заполнения поля (см. Дополнение 3).

   Проверить: должно открыться модальное окно (см. Дополнение 4).

4. В модальном окне в поле "*Nimi*" ввести значение "*karu*".

5. Нажать кнопку "*Otsi*".

Проверить: должны появиться результаты поиска.

6. Выбрать один из результатов, нажав на ссылку "*Vali*" соответствующей строки (см. Дополнение 5). Ссылка "*Vali*" запускает находящийся на главной странице *Javascript*, который закрывает iframe окно и заполняет "*Arhiivimoodustaja*" поле выбранным значением.

   Проверить:

   a. "*Arhiivimoodustaja*" поле должно быть заполненно выбранным значением.

   b. "*Arhiivi viitenumber*" поле должно быть заполнено значением, которое соответствует полю "*Arhiivi viitekood*" в выбранной строке данных.

7. Заполнить поле "*Projekti nimi*" значением "*Uus Projekt*".

8. Нажать кнопку "*Salvesta*" (см. Дополнение 6).

   Проверить: Проект успешно сохранен (см. Дополнение 7).

9. Закончить выполнение теста.

7.2.2 Тест 2— Загрузка архивной схемы для сохраненного проекта

Тест представляет собой пошаговое описание загрузки архивной схемы для существующего проекта, начиная с самого начала: от входа в систему и до подтверждения загрузки схемы.

Данный тест направлен на тестирование загрузки файлов, а также на тестирование элементов, выполненных с помощью AJAX технологии.

Исходные условия: Пользователь находится на стартовой странице приложения. (Начало следующего теста со стартовой страницы позволяет повысить его надежность. Так как его выполнение не будет зависеть от успешного выполнения предыдущего теста.)

Шаги выполнения теста:

1. Залогиниться в систему, нажав на кнопку "*Logi sisse*".

   Проверить: должна загрузиться главная страница.

2. Открыть последний сохраненный проект (первый в списке).

Проверить: должен загрузиться детальный вид (форма) проекта (см. Дополнение 7).

3. Перейти на "*Arhiiviskeemid*" закладку (Реализовано с помощью AJAX) (см. Рисунок 10).

   Проверить выполнение перехода на "*Arhiiviskeemid*" закладку.

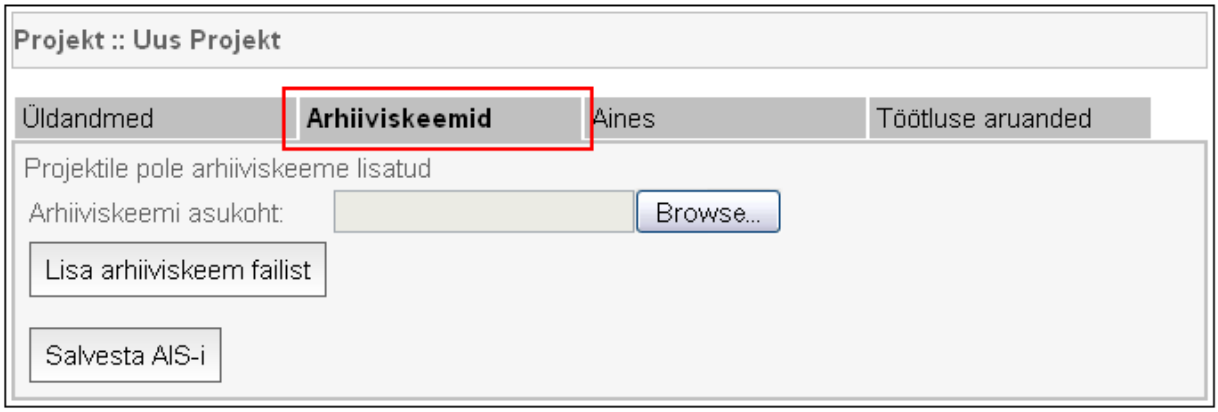


Рисунок 10. Форма открытого проекта: "*Arhiiviskeemid*" закладка

4. С помощью кнопки "*Browse...*" загрузить для поля "*Arvitiskeemi asukoht*" файл. (Файл представляет собой определенным образом структуированный *xml*-документ.)

5. Нажать на кнопку "*Lisa arhiiviskeem failist*". (Выбранный для загрузки файл проходит валидацию на соответствие расширению и внутренней структуре).

   Проверить: загруженный файл должен появиться в списке добавленных "*Arhiiviskeemid*" (происходит за счет обновления страницы) (см. Рисунок 11).

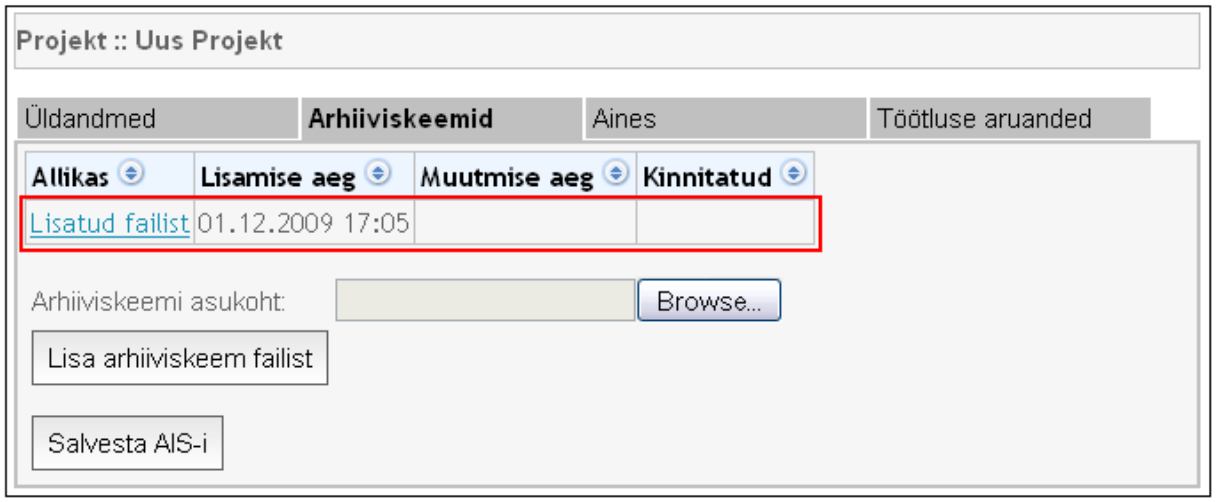


Рисунок 11. Закладка "*Arhiiviskeemid*": файл загружен

6. Закончить выполнение теста.

7.2.3 Тест 3— Администрирование загруженной архивной схемы.

Тест представляет собой пошаговое описание навигации по структуре архивной схемы.

Данный тест направлен на тестирование элементов, выполненных с помощью AJAX технологии.

Исходные условия: Пользователь находится на стартовой странице приложения.

Шаги выполнения теста:

1. Залогиниться в систему, нажав на кнопку "*Logi sisse*".

   Проверить: должна загрузиться глвная страница.

2. Открыть последний сохраненный проект (первый в списке).

   Проверить: должен загрузиться детальный вид(форма) проекта.

3. Перейти на "*Arhiiviskeemid*" закладку.

   Проверить выполнение перехода на "*Arhiiviskeemid*" закладку.

4. Открыть последнюю добавленную из файла архивную схему, нажав на ссылку "*Lisatud failist*".

   Проверить: должна загрузиться страница администрирования архивных схем (см. Дополнение 8).

5. Загруженная архивная схема представлена в виде структурного дерева в левом окне. С помощью "+" знаков открыть самый последний элемент первой ветви дерева. Реализовано с помощью AJAX-технологии.

   Проверить: при достижении последнего элемента "+" должен исчезнуть (см. Рисунок 12).

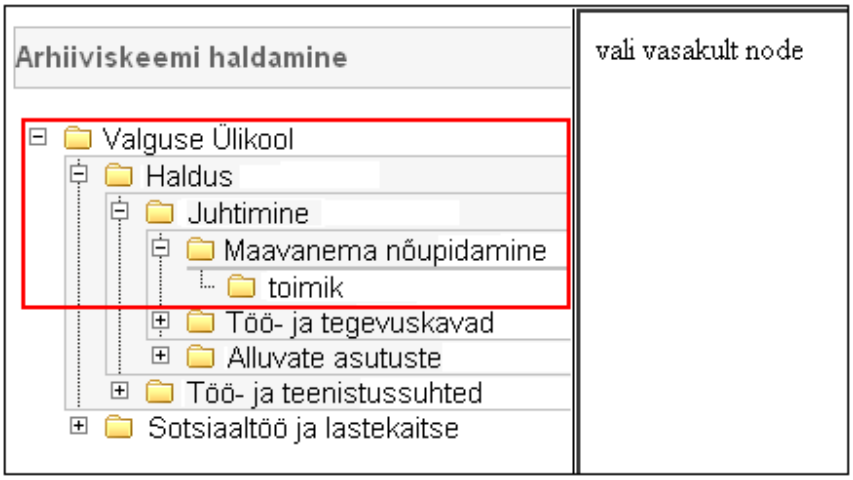


Рисунок 12. Страница управления архивными схемами: навигация по структурному дереву архивной схемы

6. Выбрать последний элемент "*toimik*" для загрузки его содержимого в правое окно.

   Проверить: В правом окне должно отобразиться содержимое документа "*toimik*". Но поскольку эта фунциональность еще не реализована в полной мере, то загружаемое содержимое имеет нечитаемый вид (см. Дополнение 9).

7. Закончить выполнение теста.

7.2.4 Тест 4— Загрузка архивного матерьяла в проект

Тест представляет собой пошаговое описание загрузки файлов в проект. Загрузка файлов отличается от загрузки архивных схем поддержкой асинхронных процессов.

Данный тест направлен на тестирование асинхронных процессов.

Исходные условия: Пользователь находится на стартовой странице приложения.

Шаги выполнения теста:

1. Залогиниться в систему, нажав на кнопку "*Logi sisse*".

   Проверить: должна загрузиться главная страница.

2. Открыть последний сохраненный проект (первый в списке).

   Проверить: должен загрузиться детальный вид (форма) проекта.

3. Перейти на "*Aines*" закладку.

   Проверить выполнение перехода на "*Aines*" закладку (см. Рисунок 13).

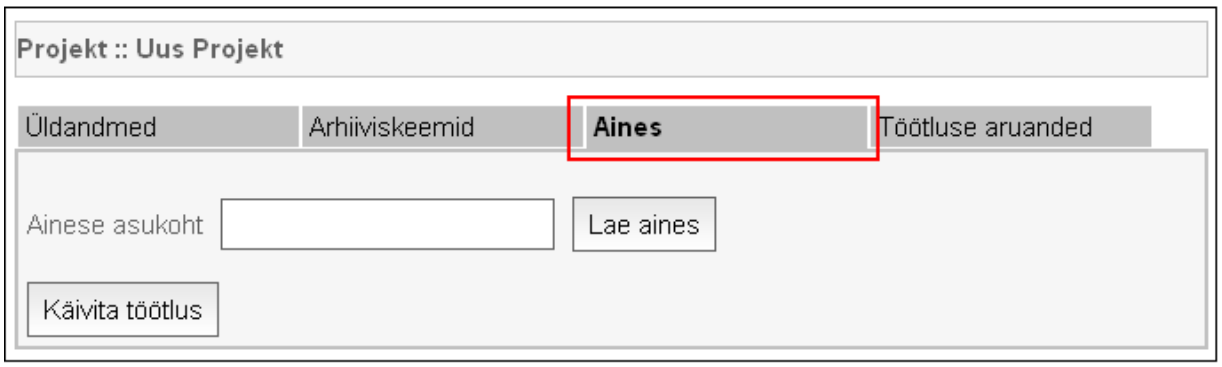


Рисунок 13. Форма открытого проекта: "*Aines*" закладка для загрузки файловых данных

4. В поле "*Ainese asukoht*" ввести путь нахождения файла, который должен быть загружен. В данном случае это "*C:\Arhiiviaines\Aines1*". Его размер около 59 MB.

5. Нажать кнопку "*Lae aines*".

   Проверить: В процессе загрузки файла должно выводиться соответственное сообщение: "*Laen andmeid kaustast: C:\Arhiiviaines\Aines1*" (см. Рисунок 14).

Рисунок 14. "*Aines*" закладка: Идет загрузка файла

6. Обновлять окно до тех пор, пока файл не загрузится. (Автоматическое обновление страницы еще не реализовано, поэтому это следует делать вручную.)

   Проверить: При завершении загрузки сообщение о загрузке должно исчезнуть.

7. Закончить выполнение теста.

7.2.5 Тест 5— Загрузка архивных данных с помощью внешнего источника

Описание теста полностью эдентично тесту номер 4. Но вместо того, чтобы вводить путь нахождения файла вручную (4. шаг), он будет браться из *excel*-таблицы (см. Рисунок 15). Таким образом, данный тест будет проигран по-очереди для каждого значения из *excel*-таблицы.

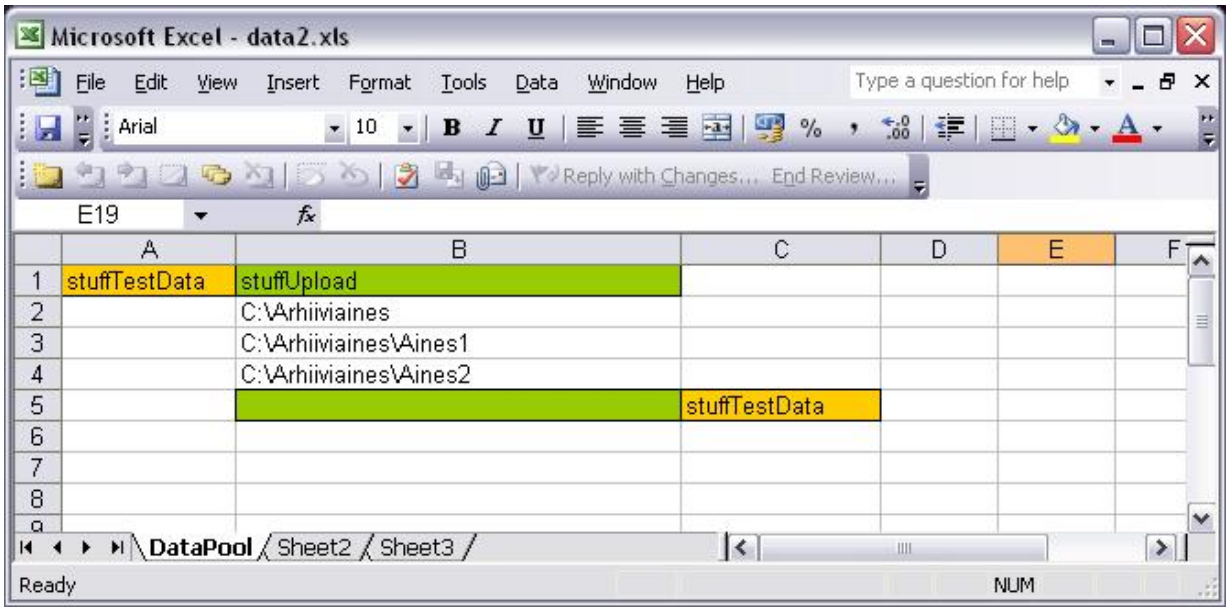


Рисунок 15. *Excel* таблица для хранения тестовых данных

Тест направлен на тестирование загрузки данных из внешнего источника, в данном случае *excel*-файла.

8 Реализация тестирования

Устанавливается следующее программное обеспечение:

- *Eclipse Java EE IDE*
- *Plugin TestNG для Eclipse*
- *Selenium Core*
- *Selenium IDE*
- *Selenium RC*

В *Eclipse* создается проект "*testng*", к которому подключаются все необходимые библиотеки. Стандартная файловая структура для тестов выглядит следующим образом (см. Рисунок 16):

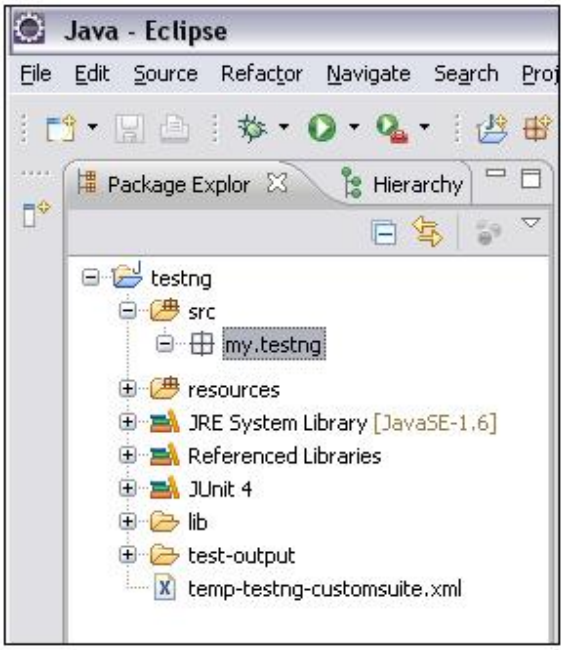


Рисунок 16. Файловая структура для проекта *testng* в *Eclipse*

В каталоге *my.testng* будут находиться проигрываемые тесты.

Теперь можно перейти к созданию непосредственно самих тестов. Пошаговое написание автоматических тестов выглядит следующим образом:

1. Запись теста в автоматическом режиме с помощью *Selenium IDE*.

2. Редактирование теста. Это в первую очередь добавление команд на проверку ожидаемых результатов. Это может быть проверка присутствует ли определенный элемент или находится ли он в правильном состоянии. Также редактированию подлежат указанные как цели значения, так как предлагаемые системой могут по определенным причинам не подходить для долговременного использования, и другие проверки.
3. После завершения изменений тест проигрывается с помощью IDE еще раз или несколько, чтобы убедиться, что тест работает правильно.
4. Конвертация созданного теста в *java*-код с помощью функции IDE "*Export Test Case As...*" –> "*Java (TestNG) - Selenium RC*". Файл сохраняется с расширением *.java*.
5. Импорт сохраненного *.java* файла в каталог тестов. В данном случае это будет *my.testng*.

По сути, имеющийся *java*-файл – это класс, содержащий последовательность вызовов *selenium*-методов, реализующих последовательно записанные действия.

Для каждого теста (класса) должны быть созданны аннотации: *@BeforeClass* и *@AfterClass*, в методы которых прописываются настройки и команды для начала и завершения работы с браузером (см. Дополнение 18).

Автор вынес эти аннотации в отделный класс, который будет расширять проигрываемые тесты. Поэтому в самих тестах не придется каждый раз описывать эти аннотации. Имя класса: *kleioBase*, который расширяет базовый класс *SeleneseTestCase*.

Если компиляция прошла успешно, запускается *Selenium-server* и запускается созданный тест в *Eclipse* под *TestNG* командой*: Run as -> TestNG Test.*

Selenium RC проигрывает тест в автоматическом режиме с установленной скоростью. При завершении проигрывания записывается отчет об успешном или нет выполнении теста.

8.1.1 Реализация теста номер 1

1. Записанный с помощью *Selenium IDE* тест состоит из девяти шагов (см. Дополнение 10). Используется небольшое количество команд:
    - **open** - открыть указанную страницу приложения.

- **clickAndWait** - нажать на указанный элемент и дождаться загрузки страницы.
- **click** - нажать на указанный элемент. Автор заметил, что IDE не всегда может уловить событие перезагрузки страницы. Поэтому вместо *clickAndWait* IDE вполне может записать просто команду *click*. Это чревато ошибкой при следующем прохождении теста, поскольку, не дождавшись загрузки страницы, *Selenium* может приступить к выполнению следующего шага, который, конечно же провалится. Поэтому при любой записанной *click*-команде тестировщик должен убедиться, что перезагрузки страницы действительно не происходит.
- **type -** ввести в указанное поле значение.

2. Отредактированные автором шаги и элементы выделены зеленым цветом (см. Дополнение 11). Были добавлены следующие команды-проверки:
   - **waitForElementPresent** - дождаться загрузки указанного элемента. Эта команда хороша как для ожидания загрузки указанного элемента, так и для проверки отображения правильного элемента на странице.
   - **verifyAttribute** - проверить наличие указанного атрибута у конкретного элемента. В контексте данного теста, это проверка на наличие у атрибута "*class*" значения "*active*" для элемента "*projectEdit*". Иными словами, проверяется активна ли закладка "*Üldandmed*" (по *id* – "*projectEdit*").
3. Создается тест в виде класса *createProject.java*. Он расширяет уже созданный класс *kleioBase* (это редактируется вручную). Поскольку вход в систему для удобства вынесен в *klieoBase* класс, данный тест начинается с третьего шага:

waitForElementPresent	//a[contains(text(),'Koosta uus projekt')]

В *java*-коде это выражется целым *for*-блоком со всякими проверками:

```
for (int second = 0;; second++) {
   if (second >= 60)
         fail("timeout");
   try {
```

```
        if (selenium.isElementPresent("//a[contains(text(),'Koosta uus
projekt')]"))
                break;
   } catch (Exception e) {
   }
        Thread.sleep(1000);
}
```

Пример кода 2. *waitForElementPresent* команда в *java*-коде

Команды нажатия на кнопку и ввода данных в поле выглядят значительно проще:

click	link=Koosta uus projekt

```
selenium.click("link=Koosta uus projekt");
```

Пример кода 3. *click* команда в *java*-коде

type	searchArchiveConstituentName	karu

```
selenium.type("searchArchiveConstituentName", "karu");
```

Пример кода 4. *type* команда в *java*-коде

Проверка атрибута элемента на соответствие выглядит следующим образом:

verifyAttribute	//span[@id='projectEdit']/@class	active

```
verifyEquals(selenium.getAttribute("//span[@id='projectEdit']/@class"),
"active");
```

Пример кода 5. *verifyAttribute* команда в *java*-коде

Команда *clickAndWait* реализуется двумя методами

clickAndWait	saveProject

```
selenium.click("saveProject");

selenium.waitForPageToLoad("30000");
```

Пример кода 6. *clickAndWait* команда в *java*-коде

Для просмотра полного текста кода см. Дополнение 19.

4. **Результат:**

   - Проигрывание теста в браузере *Internet Explorer* завершено успешно.
   - Проигрывание теста в браузере *Firefox* завершено успешно.

8.1.2 Реализация теста номер 2

1. Записанный с помощью *Selenium IDE* тест состоит из шести шагов (см. Дополнение 12). При автоматической записи были использованы те же команды, что и для теста номер 1.
2. Отредактированные автором шаги/элементы выделены зеленым цветом (см. Дополнение 13). Была добавлена неописанная в тесте 1 команда-проверка:

   - **verifyElementPresent** - проверить наличие указанного элемента на странице.

3. Создается тест с именем *failUpload.java*. Данный тест также начинается с третьего шага.

   Команда на проверку *verifyElementPresent* имеет следующий вид:

verifyElementPresent	//table[@class='t-data-grid']

```
verifyTrue(selenium.isElementPresent("//table[@class='t-data-grid']"));
```

Пример кода 7. *verifyElementPresent* команда в *java*-коде

Для просмотра полного текста кода см. Дополнение 20.

4. **Результат:**

   - Проигрывание теста в браузере *Internet Explorer* завершено ошибкой. Тест прервался на 8. шаге (см. Дополнение 27).

     Причина: Время ожидания "*//table[@class='t-data-grid']*" элемента истекло, а подтверждение нахождения элемента получено не было.

     Если следить за выполнением теста, то сразу видно, что отклонение от намеченного результата произошло на шаге номер 6. *Type*-метод должен был ввести в поле загрузки схемы значение "*C:\Documents and Settings\jarmoant\Desktop\liigitusyksus.xml*" и нажать кнопку подтверждения. При

нажатии на кнопку приложение выдает ошибку: "*Arhiiviskeemi fail on tühi*". Из этого следует, что значение не было введено.

Выяснилось, что это известная проблема работы *Selenium* с *IE* при загрузке файлов. [0] Связана она с ограничениями по соображениям безопасности браузером IE. На данный момент исправление этой ошибки средствами Selenium невозможно. Хотя имеется несколько альтернативных решений.

Решение: Для *Selenium* проблему представляет нажатие и ввод данных в поле для загрузки данных (*upload file input*). Согласно источнику [0], это может быть решено с помощью *java.awt.Robot*. Идея состоит в том, чтобы эмулировать ввод данных с клавиатуры на уровне операционной системы. По существу, это означает полную эмуляцию нажатия пользователем клавиш клавиатуры. Воспроизведение таких символов как ":" (двоеточие) будет воспроизведено передачей трех событий:

– нажать и удерживать клавишу "*Shift*"

– нажать клавишу ":"

– отпустить клавишу "*Shift*".

Таким образом нужно ввести все символы строки "*C:\Documents and Settings\jarmoant\Desktop\liigitusyksus.xml*" вплоть до эмуляции "*Enter*" по завершению ввода.

За основу был взят пример кода, приведенный на странице: http://gusiev.com/2009/04/upload-files-with-selenium-ide/. (Gusiev, 2009) Для просмотра адаптированной версии кода, использованной в данном тесте см. Дополнение 21 и Дополнение 22. К сожалению, как выяснилось, найденное решение работает только для браузера IE.

- Проигрывание теста в браузере *Firefox* завершено успешно.

8.1.3 Реализация теста номер 3

1. Записанный с помощью *Selenium IDE* тест состоит из двенадцати шагов (см. Дополнение 14). При автоматической записи были использованы те же команды, что и для теста номер 1.

2. Отредактированные автором шаги/элементы выделены зеленым цветом (см. Дополнение 15). Была добавлена ранее неиспользованнная команда-проверка:

    - **waitForAttribute** - дождаться загрузки указанного атрибута для элемента.

    В данном тесте при раскрытии каждого следующего элемента структурного дерева архивной схемы требуется произвести два вида проверки подряд: *waitForElementPresent* и *waitForAttribute*. Это значит дождаться открытия следующего элемента и дождаться пока статус этого элемента не станет желаемым. Без любой из этих проверок тест не пройдет.

3. Создается тест с именем *archiveScheme.java*. Также редактируется вручную.

    Команда на ожидание *waitForAttribute* имеет следующий вид:

waitForAttribute	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/@class

```
   for (int second = 0;; second++) {
         if (second >= 60)
               fail("timeout");
               try {
                     if ("last leaf".equals(selenium.
getAttribute("//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li" +
"@class")))
                           break;
               } catch (Exception e) {
               }
         Thread.sleep(1000);
   }
```

Пример кода 8. *waitForAttribute* команда в *java*-коде

Поскольку в тесте присутствуют повторяющиеся шаги, то это дает возможность оптимизировать их в цикл:

```
         String leafPath = "//div[@id='archiveTree']/ul[@class='ltr']/li";
// путь первого элемента архивной схемы "Valguse Ülikool" инициализируется
как переменная leafPath
```

```
        String nextLeaf = "/ul/li[1]"; // отрезок пути до следующего
элемента архивной схемы инициализируется как переменная nextLeaf
        selenium.click(leafPath);
        try {
              while (selenium.getAttribute(leafPath + nextLeaf +
"/@class").matches("^closed[\\s\\S]*$")) {
                    for (int second = 0;; second++) {
                          if (second >= 60)
                                fail("timeout");
                          try {
                                if (selenium.isElementPresent(leafPath +=
nextLeaf))
                                      break;
                          } catch (Exception e) {
                          }
                          Thread.sleep(1000);
                    } // for цикл ожидает следующий элемент архивной схемы
                    selenium.click(leafPath);
              } // while цикл будет работать до тех пор, пока каждый
следующий элемент архивной схемы будет иметь статус "closed"
        } catch (Exception e) {}
```

Пример кода 9. Повторяющиеся шаги раскрытия архивной схемы оптимизированы в цикл

Для просмотра полного текста кода см. Дополнение 23.

4. **Результат:**

   - Проигрывание теста в браузере *Internet Explorer* завершено успешно.
   - Проигрывание теста в браузере *Firefox* завершено успешно.

8.1.4 Реализация теста номер 4

1. Записанный с помощью Selenium IDE тест состоит из шести шагов (см. Дополнение 16). При автоматической записи была добавлена раннее не использованная команда:

   - **refresh** - обновить страницу.

2. Отредактированные автором шаги/элементы выделены зеленым цветом (см. Дополнение 17). Были добавлени ранне не использованные команды-проверки:

- **assertText -** подтвердить совпадение текста
- **verifyElementNotPresent** - проверить, что указанный элемент не присутствует на странице.

3. Создается тест с именем *stuffUpload.java*. Также редактируется вручную.

   Команда *assertText* легко узнаваема:

assertText	//div[@class='message']/span

```
assertTrue(selenium.getText("//div[@class='message']/span").matches("^Laen andmeid kaustast[\\s\\S]*$"));
```

Пример кода 10. *assertText* команда в *java*-коде

В IDE не было возможности создать такое условие, чтобы браузер обновлялся до тех пор, пока сообщение о загрузке файла не исчезнет, как этого требует данной тест. Поэтому команды обновления, а затем проверки на присутствие сообщения были выполнены по одному разу:

refresh	
verifyElementNotPresent	//div[@class='message']/span

После редактирования в *Eclipse* код выглядит следующим образом:

```
try{ while (selenium.getText("//div[@class='message']/span").matches(
                    "^Laen andmeid kaustast[\\s\\S]*$")) {
             selenium.refresh();
      } } catch (Exception e){}
```

Пример кода 11. Обновление браузера до тех пор, пока не исчезнет сообщение о загрузке

Для просмотра полного текста кода см. Дополнение 24.

4. **Результат:**

   Проигрывание теста в браузере *Internet Explorer* завершено успешно.

   Проигрывание теста в браузере *Firefox* завершено успешно.

8.1.5 Реализация теста номер 5

За основу взят записанный и отредактированный тест номер 4. Поскольку данный тест должен вводить в поле значения, которые он будет брать из файла, имеет смысл вынести шаги, совершаемые до ввода значений, в отдельный класс *DataDrivenBase.java*, чтобы тест не начинался снова для каждого нового значения. Чтение из файла "*data2.xls*", в котором находится таблица со значениями данных для ввода, и собственно сам ввод этих данных будет происходить в *stuffDataDriven.java*. Пример кода взят из Интернета со страницы: http://functionaltestautomation.blogspot.com/2009/10/dataprovider-data-driven-testing-with.html. (Narayanan, 2009)

Для просмотра полного текста кода см. Дополнение 25 и Дополнение 26.

Результат:

- Проигрывание теста в браузере *Internet Explorer* завершено успешно.
- Проигрывание теста в браузере *Firefox* завершено успешно.

Заключение

В данной работе было проведено автоматическое тестирование веб-приложения, находящегося на стадии разработки для Национального Архива, с помощью инструмента Selenium. Преследуемой целью было ответить на поставленный вначале работы главный вопрос: подходит ли выбранный фреймворк, в данном случае это Selenium, для тестирования веб-приложения Национального Архива. Руководствуясь одним лишь описанием поддерживаемых технологий из сопроводительной документации по Selenium, невозможно заранее ответить, как будет тестовый фреймворк работать с конкретным приложением и будет ли вообще. Единственным возможным способом выяснить это, было практическое проведение автоматического тестирования.

Для этого вначале автор составил сценарии тестирования таким образом, чтобы охватить имеющуюся функциональность приложения в объеме, достаточном для проверки работы Selenium со всеми используемыми технологиями. По имеющимся сценариям средствами Selenium были созданы сами тесты для автоматического проигрывания. Ввиду ограничения объема работы было составлено пять тестов, каждый из которых был проведен на двух браузерах: Internet Explorer 8 и Firefox 3.5.6. Таким образом, можно считать, что было проведено десять тестов. Первоначально, из десяти проведенных тестов были успешно завершены девять. Трудности возникли с тестом номер 2 при его проигрывании на IE. В данном случае, это уже известная и зарегистрированная ошибка: SEL-63. К счастью, Selenium имеет достаточно широкое распространение и активное сообщество, благодаря чему автор смог исправить свой тест указанным методом. Устранение ошибки потребовало от автора достаточно глубокого погружения в матерьялы по теме, а также навыков программирования. Поэтому сказать, что решение далось просто, тоже нельзя. Но даже учитывая, что оно не очень удобное, поскольку требует наличие двух разных тестов отдельно для Firefox и отдельно для IE, его все же можно использовать.

Полученные результаты дают возможность сделать выводы относительно использования Selenium для автоматизации тестирования приложения Национального Архива. Результат 10 из 10 может означать только положительный ответ. Selenium способен адекватно работать со всеми задействованными в разработке данного приложения технологиями. Возникшие проблемы с загрузкой файлов в тесте номер 2 были решены, а потому нет причин для поиска других средств автоматизации.

Использование Selenium для проведения автоматического тестирования на приложении Национального Архива в данном случае полностью оправдано.

Kokkuvõte

Bakalaureusetöö eesmärgiks on leida sobiv vahend Rahvusarhiivi veebirakenduse automaatseks testimiseks . Eesmärgiks oli vastata töö alguses püstitatud põhiküsimusele: kas valitud testimise raamistik Selenium sobib Rahvusarhiivi veebirakenduse testimise jaoks. Lähtudes ainult Seleniumi saatedokumentatsioonis olevast toetatud tehnoloogiate kirjeldusest ei ole võimalik vastata küsimusele kas ja kuidas testimise raamistik töötab konkreetse rakendusega. Ainsaks võimaluseks seda selgeks teha, on teostada automaattestimine praktikas.

Kõigepealt koostas autor testlood sellisel viisil, et need kataksid olemasolevat rakenduse funktsionaalsust piisavas mahus, et kontrollida Seleniumi sobivust kõigi rakenduses kasutusele võetud tehnoloogiate testimiseks. Koostatud testlugude järgi loodi Seleniumi vahendite abil testid automaatseks testimiseks. Kokku koostati viis testi, igaüks nendest prooviti läbi kahe brauseriga: Internet Explorer 8 ja Firefox 3.5.6. Sel viisil võib lugeda, et viidi läbi 10 testi. Esialgu läbiti edukalt üheksa testi kümnest. Raskused tekkisid teise testi läbi mängimisel IE veebilehitsejal. Selgus, et tegemist on tuntud ja registreeritud veaga: SEL-63. Tänu Seleniumi aktiivsele kasutajate kogukonnale õnnestus leida probleemile lahendus ning autor sai parandada oma testi kirjeldatud viisil. Antud vea kõrvaldamine ei olnud autori jaoks lihtne, kuna sobiva lahenduse leidmine eeldas sügavamat uurimistööd ning programmeerimisoskust. Leitud lahendust ei saa pidada kõige mugavamaks, kuna see töötab ainult IE jaoks, Firefoxi jaoks töötab aga alguses koostatud test. Aga igal juhul võib väita, et lahendus oli leitud.

Saadud tulemuste põhjal saab teha järeldused Seleniumi kasutamise sobivuse kohta Rahvusarhiivi veebirakenduse automaattestimisel. Kõigi kümne testi positiivne läbimine lubab väita, et Seleniumit saab kasutada veebirakenduse testimiseks ning ta sobib arendamiseks kasutatud tehnoloogiatega. Faili üleslaadimisega seotud probleem teise testi läbimängimisel sai lahendatud, see tähendab, et vajadust otsida teist automattestimise raamistikku ei ole.

Selenimi kasutamine Rahvusarhiivi rakenduse automaatseks testimiseks on antud juhul täiesti õigustatud.

Библиография

Apache Tapestry [сайт разработчика]. Retrieved 02.01.2010 from http://tapestry.apache.org/

Automated QA. TestComplete. [сайт разработчика] Retrieved 02.01.2010 from http://www.aqa.com.ru/products.asp

Borland. SillkTest 2009. [сайт разработчика] Retrieved 02.01.2010 from http://www.borland.com/us/products/silk/silktest/index.html

Canoo WebTest [сайт разработчика]. Retrieved 02.01.2010 from http://webtest.canoo.com/webtest/manual/WebTestHome.html

Gusiev, B. (2009). Upload files with Selenium IDE. Retrieved 02.01.2010 from http://gusiev.com/2009/04/upload-files-with-selenium-ide/

HP QuickTest Professional software. [сайт разработчика] Retrieved 02.01.2010 from https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__

IBM. Rational Functional Tester. [сайт разработчика] Retrieved 02.01.2010 from http://www-142.ibm.com/software/products/ru/ru/functional

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2007). *Software testing: An ISEB Foundation*. Test Levels, 39-46.

Narayanan, M. (2009). DataProvider - Data Driven Testing with Selenium and TestNG. Retrieved 02.01.2010 from http://functionaltestautomation.blogspot.com/2009/10/dataprovider-data-driven-testing-with.html

National Institute of Standards & Technology. (2002). Planing Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing. Retrieved 02.01.2010 from http://www.nist.gov/director/prog-ofc/report02-3.pdf

OpenQA. (2007). Bug Tracker. SEL-63. Retrieved 02.01.2010 from http://jira.openqa.org/browse/SEL-63

OpenQA. (2009). Selenium. Flow control extension for Selenium Core. Retrieved 02.01.2010 from http://wiki.openqa.org/display/SEL/flowControl

Selenim HQ. Web application testing system. [сайт разработчика] Retrieved 02.01.2010 from http://seleniumhq.org/

SWEBOK. (2004). *Guide to the Software Engineering Body* of *Knowledge*. Test Techniques, 77-79.

Булат, А. (2007a). Виды Тестирования Программного Обеспечения. Retrieved 02.01.2010 from http://www.protesting.ru/testing/testtypes.html

Булат, А. (2007b). Зачем нужно автоматизировать? Retrieved 02.01.2010 from http://www.protesting.ru/automation/functional/whytoauto.html

Булат, А. (2007c). Как автоматизировать? Retrieved 02.01.2010 from http://www.protesting.ru/automation/functional/howtoauto.html

Булат, А. (2008a). Обеспечение качества, Контроль качества, Тестирование. Retrieved 02.01.2010 from http://www.it4business.ru/up/919/

Булат, А. (2008b). Разница между QA, QC & Testing. [Рисунок]. Retrieved 02.01.2010 from http://alexeybulat.blogspot.com/2007/12/qa-qc-testing.html

Инструменты для автоматизации функционального тестирования. (2006). [Рисунок]. Retrieved 02.01.2010 from http://msemenkin.livejournal.com/2161.html

Кантор, И. (2008). Юнит-тесты уровня браузера на связке Selenium + PHP. Retrieved 02.01.2010 from http://javascript.ru/unsorted/selenium-rc#kak-ustroen-selenium-ochen-kratko

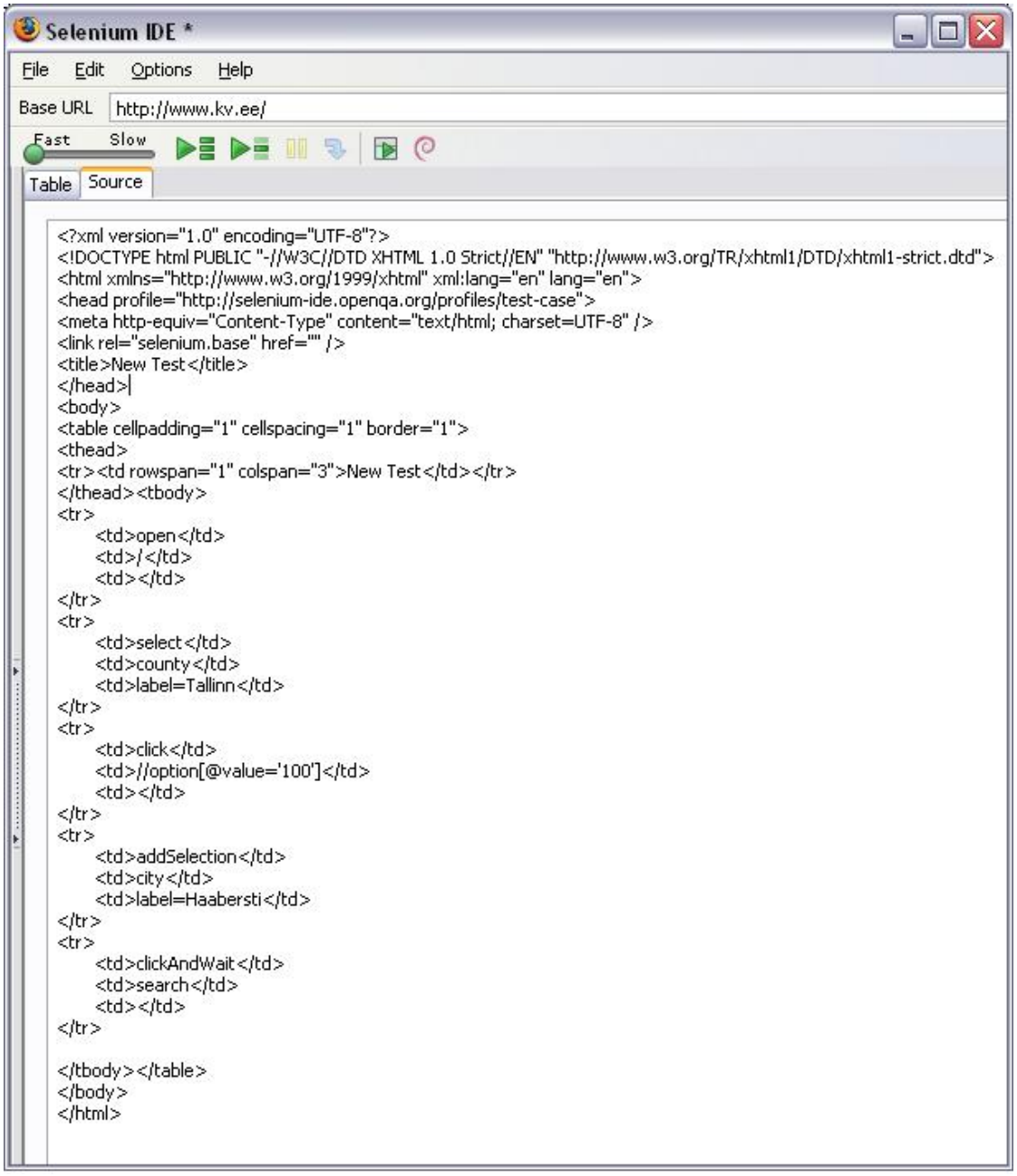
Маклафлин, Б. (2005). Освоение Ajax: Введение в Ajax. Retrieved 02.01.2010 from http://www.ibm.com/developerworks/ru/library/wa-ajaxintro1/index.html

Поляруш, М. (2009). Selenium Core. Общие понятия. Retrieved 02.01.2010 from http://www.automated-testing.info/knowledgebase/article/selenium-core-obshhie-ponjatija

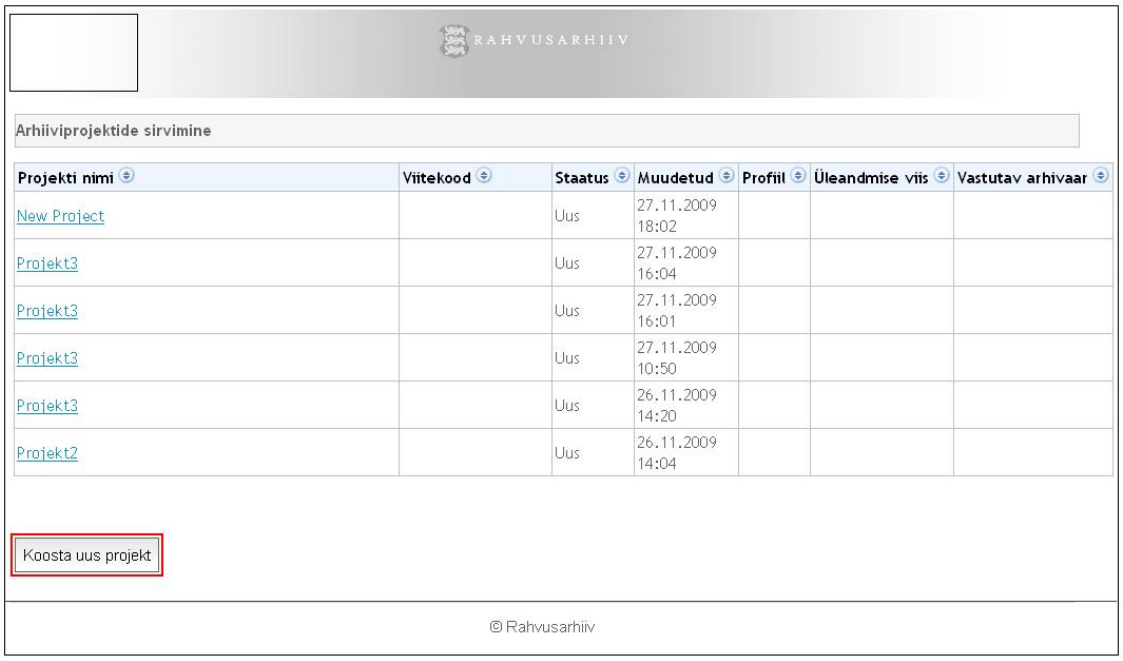
Сортов А., & Хорошилов А. (2004). Функциональное тестирование Web-приложений на основе технологии UniTesK. Retrieved 02.01.2010 from http://www.citforum.ru/SE/testing/func_testing/

Хелстен, К. (2007). Автоматические приемочные тесты с Selenium. Retrieved 02.01.2010 from http://www.ibm.com/developerworks/ru/library/wa-selenium-ajax/

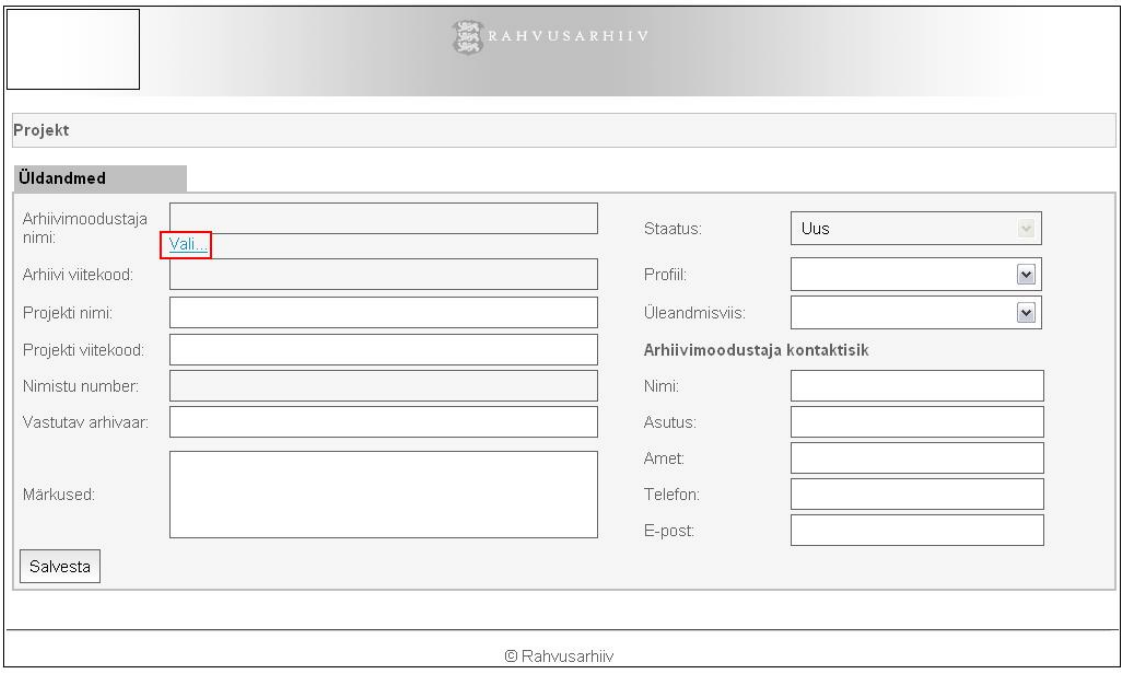
ДОПОЛНЕНИЯ



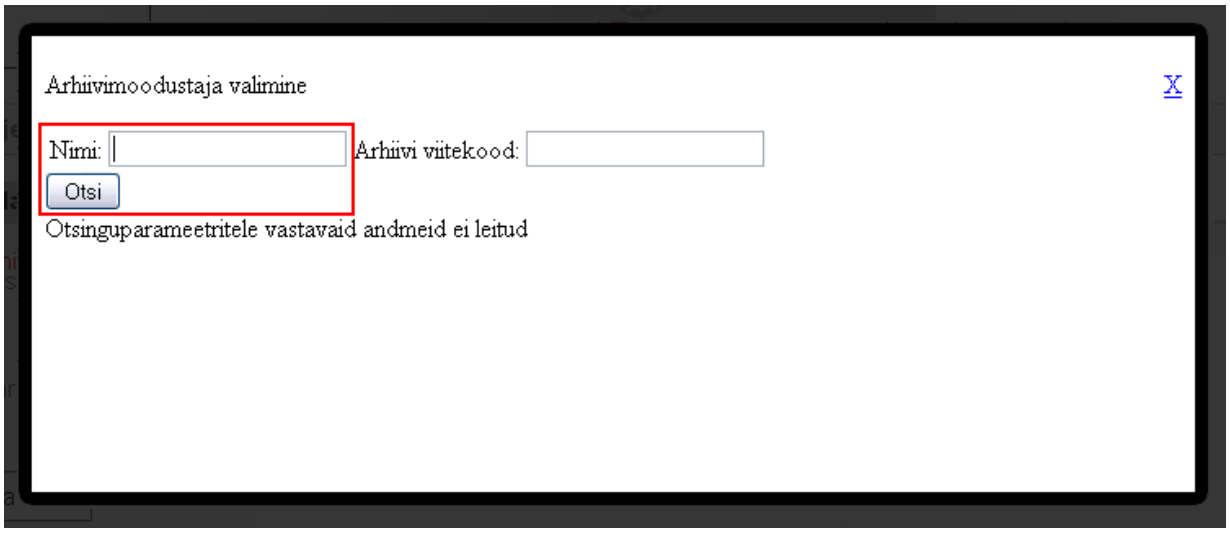
Дополнение 1. Записанный на IDE тест в представлении *Selenese*



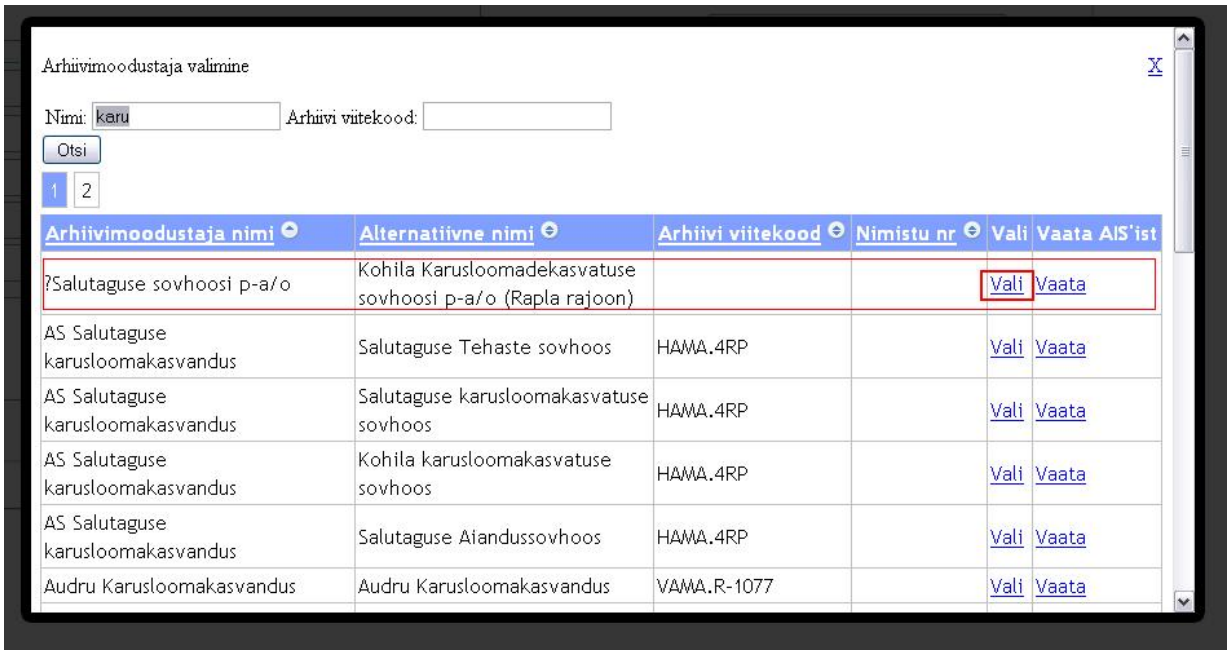
Дополнение 2. Список проектов: Создать новый проект



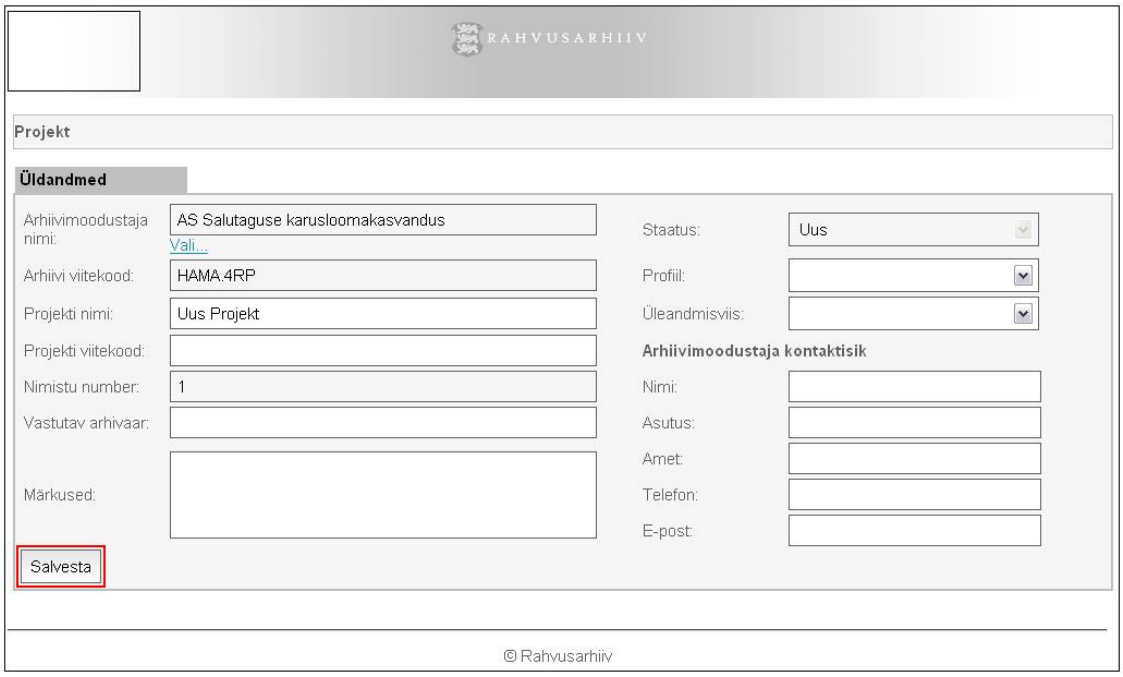
Дополнение 3. Форма создания нового проекта: Заполнение обязательного поля



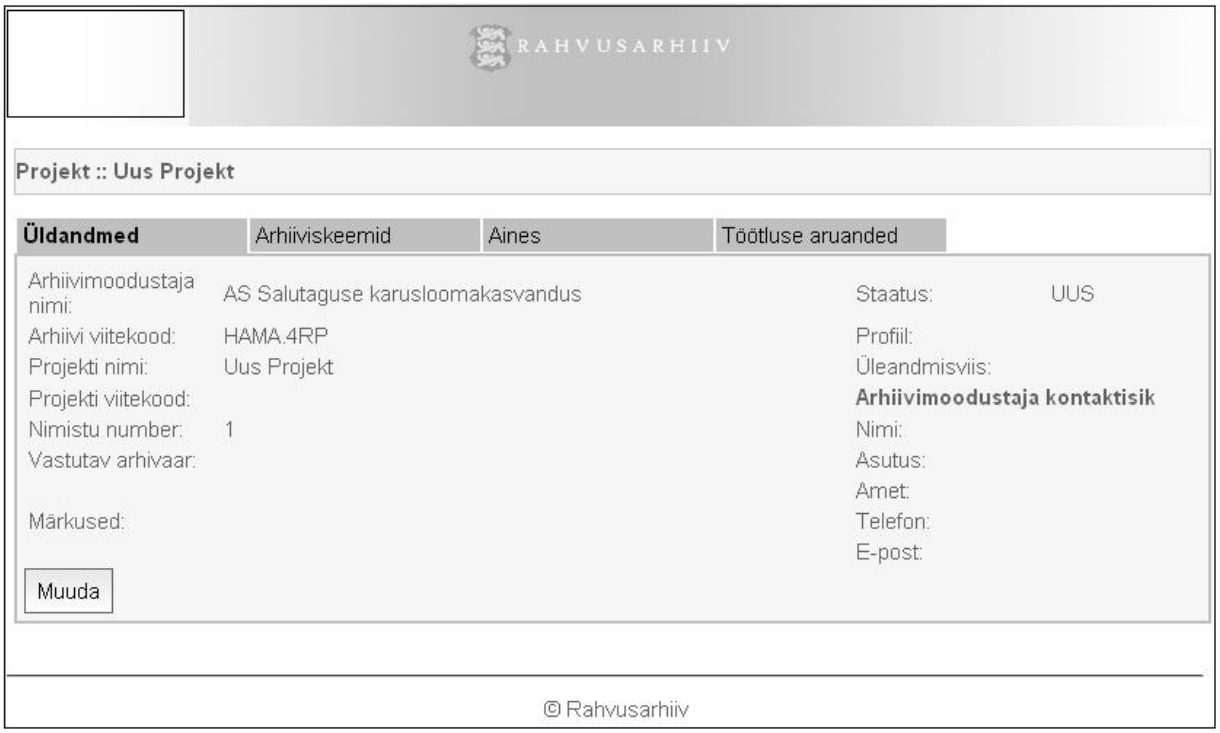
Дополнение 4. Модальное окно: Выполнение поиска для заполнения обязательного поля



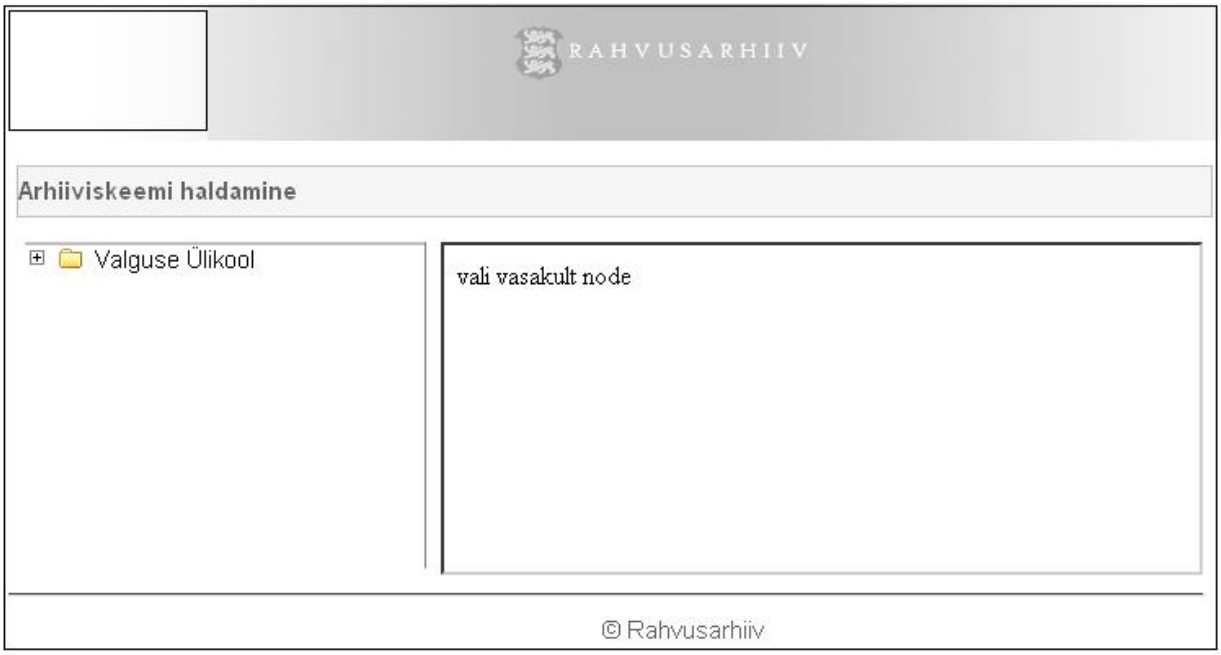
Дополнение 5. Модальное окно: Выведенные в окно результаты поиска



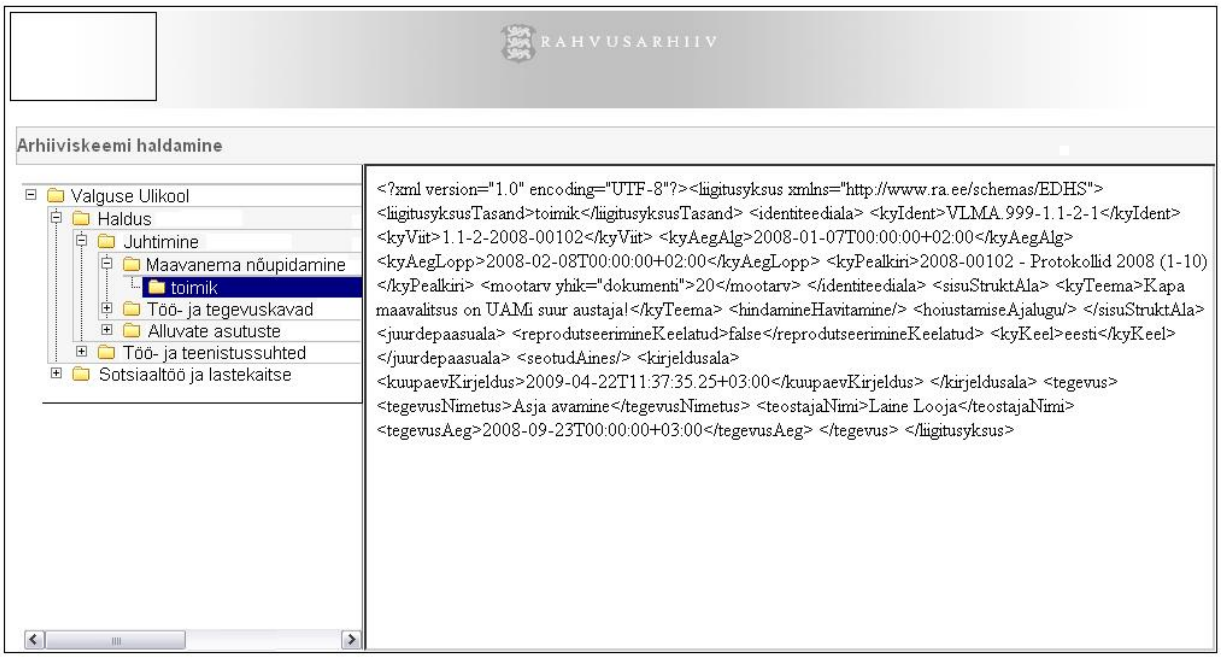
Дополнение 6. Форма создания нового проекта: Сохранить проект



Дополнение 7. Сохраненный проект: "*Üldandmed*" закладка



Дополнение 8. Страница управления архивными схемами: Просмотр архивной схемы



Дополнение 9. Страница управления архивными схемами: отображение содержимого документа

Шаг	Команда	Цель	Значение
1	open	/kleio/	
2	clickAndWait	submit	
3	clickAndWait	link=Koosta uus projekt	
4	click	link=Vali...	
5	type	searchArchiveConstituentName	karu
6	clickAndWait	sendSearch	
7	click	link=Vali	
8	type	projectName	Uus Projekt
9	clickAndWait	saveProject	

Дополнение 10. Записанный с помощью *Selenium IDE* тест: Создание проекта (Тест 1)

Шаг	Команда	Цель	Значение	Пояснение
1	open	/kleio/		//открыть страницу "*/kleio/*"
2	clickAndWait	submit		//нажать на кнопку "*Logi sisse*" и дождатья загрузки страницы
3	waitForElementPresent	//a[contains(text(),'Koosta uus projekt')]		//ждать до появления кнопки "*Koosta uus projekt*"
4	click	link=Koosta uus projekt		//нажать на кнопку "*Koosta uus projekt*"
5	waitForElementPresent	//span[@id='projectEdit']		//ждать загрузки "*Üldandmed*" закладки
6	verifyAttribute	//span[@id='projectEdit']/@class	active	//убедиться, что "*Üldandmed*" закладка активна
7	click	link=Vali...		//нажать на ссылку "*Vali ... *"
8	waitForElementPresent	//form[@id='searchConstituent']/input		//ждать до появления поля для ввода критериев поиска
9	type	searchArchiveConstituentName	karu	//ввести в поле значение "*karu*"
10	click	sendSearch		//нажать на кнопку поиска
11	waitForElementPresent	//div[@class='t-data-grid']/		//ждать до выдачи таблицы с результатами поиска
12	click	//div[@id='archiveConstituentListDiv']/div/ta		//нажать на ссылку "*Vali*" первой строки в таблице

		ble/tbody/tr[1]/td[5]/a		
13	waitForElementPresent	projectName		//ждать до появления поля ввода для имени проекта
14	type	projectName	Uus Projekt	//ввести в поле значение "*Uus Projekt*"
15	clickAndWait	saveProject		//нажать на кнопку "*Salvesta*"
16	waitForElementPresent	editMode		//дождаться появления кнопки "*Muuda*"

Дополнение 11. Отредактированный тест: Создание проекта (Тест 1)

Шаг	Команда	Цель	Значение
1	open	/kleio/	
2	clickAndWait	submit	
3	click	link=New Project	
4	click	eventLink_0	
5	type	upload-12572d08b5f	C:\Documents and Settings\jarmoant \Desktop\liigitusyksus.xml
6	click	loadSchemaFromFile	

Дополнение 12. Записанный с помощью *Selenium IDE* тест: Загрузка архивной схемы (Тест 2)

Шаг	Команда	Цель	Значение	Пояснение
1	open	/kleio/		//открыть страницу "*/kleio/*"
2	clickAndWait	submit		//нажать на кнопку "*Logi sisse*" и дождатья загрузки
3	clickAndWait	//tr[@class='t-last']/td[1]/a		// нажать на последний созданный проект в таблице и ждать загрузки формы
4	click	//a[contains(text(),'Arhiiviskeemid')]		//нажать на "*Arhiiviskeemid*" закладку
5	verifyAttribute	//span[@id='projectArchiveStructures']/@class	active	//проверить активность "*Arhiiviskeemid*" закладки
6	type	//input[@name='upload']	C:\Documents and Settings\jarmoant\Desktop\liigitusyksus.xml	//ввести в поле загрузки файлов путь нахождения файла "*liigitusyksus.xml*"
7	click	loadSchemaFromFile		// нажать на кнопку "*Lisa arhiiviskeem failist*"
8	verifyElementPresent	//table[@class='t-data-grid']		//проверить появление таблицы с загруженным

				файлом

Дополнение 13. Отредактированный тест: Загрузка архивной схемы (Тест 2)

Шаг	Команда	Цель	Значение
1	open	/kleio/	
2	clickAndWait	submit	
3	clickAndWait	link=New Project	
4	click	eventLink_0	
5	clickAndWait	link=Lisatud failist	
6	click	kleiotest/1000614/schemas/3d6c528c.xml_ERA.9004	
7	click	kleiotest/1000206/schemas/cef73361.xml_VLMA.999	
8	click	kleiotest/1000206/schemas/cef73361.xml_VLMA.999-1	
9	click	kleiotest/1000206/schemas/cef73361.xml_VLMA.999-1.1	
10	click	kleiotest/1000206/schemas/cef73361.xml_VLMA.999-1.1-2	
11	click	kleiotest/1000206/schemas/cef73361.xml_VLMA.999-1.1-2-1	
12	click	//li[@id='kleiotest/1000206/schemas/cef73361.xml_VLMA.999-1.1-2-1']/a/ins	

Дополнение 14. Записанный с помощью *Selenium IDE* тест: Навигация по аривной схеме (Тест 3)

Шаг	Команда	Цель	Значение	Пояснение
1	open	/kleio/		//открыть страницу "*/kleio/*"
2	clickAndWait	submit		//нажать на кнопку "*Logi sisse*" и дождатья загрузки
3	clickAndWait	//tr[@class='t-last']/td[1]/a		// нажать на последний созданный проект в таблице и ждать загрузки формы
4	click	//a[contains(text(),'Arhiiviskeemid')]		//нажать на "*Arhiiviskeemid*" закладку
5	verifyAttribute	//span[@id='projectArchiveStructures']/@class	active	//проверить активность "*Arhiiviskeemid*" закладки
6	clickAndWait	//table[@class='t-data-grid']/tbody/tr[contains(@class,'t-last')]/td/a		//нажать на последний загруженный файл в таблице и ждать загрузки
7	waitForElementPresent	//div[@id='archiveTree']/ul/li[contains(@id, '.xml')]		//дождаться загрузки архивной схемы в левом окне
8	click	//li		//нажать на элемент архивной схемы "*Valguse Ülikool*"
9	waitForElementPresent	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]		//дождаться раскрытия элемента "*Haldus*"
10	waitForAttribute	//div[@id='archiveTree']	closed*	//дождаться статуса

		/ul[@class='ltr']/li/ul/li[1]/@class		начинающегося с "*closed*" для элемента "*Haldus*"
11	click	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]		// нажать на элемент архивной схемы "*Haldus*"
12	waitForElementPresent	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]		//дождаться раскрытия элемента "*Juhtimine*"
13	waitForAttribute	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/@class	closed*	//дождаться статуса начинающегося с "*closed*" для элемента "*Juhtimine*"
14	click	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]		// нажать на элемент архивной схемы "*Juhtimine*"
15	waitForElementPresent	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]		//дождаться раскрытия элемента "*Maavanema nõupidamine*"
16	waitForAttribute	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/@class	closed*	//дождаться статуса начинающегося с "*closed*" для элемента "*Maavanema nõupidamine*"
17	click	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]		// нажать на элемент архивной схемы "*Maavanema nõupidamine*"
18	waitForElementPresent	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li		//дождаться раскрытия элемента "*toimik*"
19	waitForAttribute	//div[@id='archiveTree']	closed*	//дождаться статуса

		/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/@class		начинающегося с "*closed*" для элемента "*toimik*"
20	click	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/		// нажать на элемент архивной схемы *“toimik”*
21	waitForElementPresent	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/a/ins		//дождаться раскрытия элемента
22	waitForAttribute	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/@class	last leaf	//дождаться статуса начинающегося с "*last leaf* " для элемента "*toimik*"
23	click	//div[@id='archiveTree']/ul[@class='ltr']/li/ul/li[1]/ul/li[1]/ul/li[1]/ul/li/a		// нажать на "*toimik*" элемент архивной схемы
24	verifyAttribute	//iframe[@id='ASDetailViewFrame']/@src	*.xml*	//проверить, что в правое окно было загружено содержимое *.xml* файла

Дополнение 15. Отредактированный тест: Навигация по аривной схеме (Тест 3)

Шаг	Команда	Цель	Значение
1	open	/kleio/	
2	clickAndWait	submit	
3	click	link=New Project	
4	click	eventLink_1	
5	type	dataSource	C:\Arhiiviaines\Aines1
6	click	loadSip	

Дополнение 16. Записанный с помощью *Selenium IDE* тест: Загрузка данных (Тест 4)

Шаг	Команда	Цель	Значение	Пояснение
1	open	/kleio/		//открыть страницу "*/kleio/*"
2	clickAndWait	submit		//нажать на кнопку "*Logi sisse*" и дождаться загрузки
3	clickAndWait	//tr[@class='t-last']/td[1]/a		// нажать на последний созданный проект в таблице и ждать загрузки формы
4	click	//a[contains(text(),'Aines')]		//нажать на "*Aines*" закладку
5	verifyAttribute	//span[@id='projectMaterial']/@class	active	//проверить активность "*Aines*" закладки
6	type	//input[contains(@id,'dataSource')]	C:\Arhiiviaines\Aines1	//ввести в поле загрузки файлов путь нахождения файла
7	click	loadSip		// нажать на кнопку "*Lae aines*"
8	verifyElementP	//div[@class='message']		//проверить

	resent			появление сообщения о загрузке
9	assertText	//div[@class='message']/span	Laen andmeid kaustast*	//проверить текст сообщения
10	refresh			//обновить страницу
11	verifyElementNotPresent	//div[@class='message']/span		//проверить: сообщение должно пропасть

Дополнение 17. Отредактированный тест: Загрузка данных (Тест 4)

```
package my.testng;
import org.testng.annotations.AfterClass;
public class kleioBase extends SeleneseTestCase {
	@BeforeClass // аннотация начала работы с браузером
	public void setUp() throws Exception {
		setUp("http://ww015425:7070/", "*iexplore"); //указывает базовый URL приложения и браузер, на котором будет производиться тестирование. *iexplore - в случае Intrenet Explorer и *firefox в случае Firefox
		selenium.setSpeed("2000"); // устанавливает скорость выполнения тестов. 2000 - время ожидания между выполнением шагов в милисекундах
		selenium.open("/kleio/login"); //открыть указанную страницу приложения
		selenium.waitForPageToLoad("30000"); // дождаться загрузки страницы. Время ожидания 30000 милисекунд
		selenium.click("submit"); // нажать на кнопку "submit"
		selenium.waitForPageToLoad("30000");
		selenium.windowMaximize(); // расширить окно браузера на полный экран
		selenium.windowFocus(); //удерживает фокус на окне проигрываемого теста
	}
	@AfterClass // аннотация завершения работы с браузером
	public void tearDown() {
		selenium.close();
		selenium.stop();
	}
}
```

Дополнение 18. Базовый класс *kleioBase* для начала и завершения работы с браузером

```
package my.testng;
import org.testng.annotations.Test;
public class createProject extends kleioBase {
	@Test
	public void testCreateProject() throws Exception {
		/**
		 * Подобного вида for-цикл выражет команду
		 * "waitForElementPresent".
		 * Эта команда дожидается появления на странице элемента,
		 * содержащего текст "Koosta uus projekt"
		 */
		for (int second = 0;; second++) {
			if (second >= 60)
				fail("timeout");
			try {
				if (selenium
					.isElementPresent("//a[contains(text(),'Koosta
uus projekt')]"))
					break;
			} catch (Exception e) {
			}
			Thread.sleep(1000);
		}
		selenium.click("link=Koosta uus projekt");
		for (int second = 0;; second++) {
			if (second >= 60)
				fail("timeout");
			try {
				if (selenium.isElementPresent
("//span[@id='projectEdit']"))
					break;
			} catch (Exception e) {
			}
			Thread.sleep(1000);
		}
	verifyEquals(selenium.getAttribute("//span[@id='projectEdit']/@class"),
				"active");
		selenium.click("link=Vali...");
```

```
            for (int second = 0;; second++) {
                  if (second >= 60)
                        fail("timeout");
                  try {
                        if (selenium
                  .isElementPresent("//form[@id='searchConstituent']/input"))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
            selenium.type("searchArchiveConstituentName", "karu");
            selenium.click("sendSearch");
            for (int second = 0;; second++) {
                  if (second >= 60)
                        fail("timeout");
                  try {
                        if (selenium.isElementPresent("//div[@class='t-data-
grid']/"))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
            selenium.click("//div[@id='archiveConstituentListDiv']
/div/table/tbody/tr[1]/td[5]/a");
            for (int second = 0;; second++) {
                  if (second >= 60)
                        fail("timeout");
                  try {
                        if (selenium.isElementPresent("projectName"))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
            selenium.type("projectName", "New Project");
            selenium.click("saveProject");
            for (int second = 0;; second++) {
                  if (second >= 60)
```

```
                        fail("timeout");
                  try {
                        if (selenium.isElementPresent("editMode"))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
      }
}
```

Дополнение 19. *createProject* класс реализует создание нового проекта (Тест 1)

```
package my.testng;
import org.testng.annotations.Test;
public class failUpload extends kleioBase {
	@Test
	public void testFailUpload() throws Exception {
		selenium.click("//tr[@class='t-last']/td[1]/a");
		selenium.waitForPageToLoad("30000");
		selenium.click("//a[contains(text(),'Arhiiviskeemid')]");
		verifyEquals(selenium.getAttribute(
"//span[@id='projectArchiveStructures']/@class"),"active");
		selenium.type("//input[contains(@id,'upload')]","C:\\Documents and
Settings\\jarmoant\\Desktop\\Server\\liigitusyksus.xml");
		selenium.click("loadSchemaFromFile");
		for (int second = 0;; second++) {
			if (second >= 60) fail("timeout");
			try { if (selenium.isElementPresent("//table[@class='t-data-
grid']")) break; } catch (Exception e) {}
			Thread.sleep(1000);
		}
	}
}
```

Дополнение 20. *failUpload* класс реализует загрузку архивной схемы (Тест 2)

```
package my.testng;
import org.testng.annotations.Test;
public class failUploadForIE extends kleioBase {
      @Test
      public void testFailUploadForIE() throws Exception {
            selenium.click("//tr[@class='t-last']/td[1]/a");
            selenium.waitForPageToLoad("30000");
            selenium.click("//a[contains(text(),'Arhiiviskeemid')]");
            verifyEquals(selenium.getAttribute(
"//span[@id='projectArchiveStructures']/@class"),"active");
            chooseFile("//input[contains(@id,'upload')]","C:\\Documents and
Settings\\jarmoant\\Desktop\\Server\\liigitusyksus.xml"); // команда ввода
"type" заменена на метод "chooseFile"
            selenium.click("loadSchemaFromFile");
            for (int second = 0;; second++) {
                  if (second >= 60) fail("timeout");
                  try { if (selenium.isElementPresent("//table[@class='t-data-
grid']")) break; } catch (Exception e) {}
                  Thread.sleep(1000);
            }
            verifyTrue(selenium.isElementPresent("//table[@class='t-data-
grid']"));
            /**
             * chooseFile метод определяет позицию кнопки "Browse",
             * запускает отдельный Thread
             * и нажимает на кнопку "Browse"
             */
      protected void chooseFile(String element, String fileName) {
            Number positionLeft = selenium.getElementPositionLeft(element);
            Number positionTop = selenium.getElementPositionTop(element);
            new FileChooserThread(fileName).start(); // запускает новый Thread
типа: FileChooserThread, в который передает имя файла
            selenium.clickAt(element, positionLeft + "," + positionTop); //
нажать на кнопку "Browse"
      }
}
```

Дополнение 21. Решение загрузки архивной схемы для проигрывания в браузере IE

```
package my.testng;
import java.awt.Robot;
import java.awt.event.KeyEvent;
import javax.swing.KeyStroke;
public class FileChooserThread extends Thread {
      public FileChooserThread(String file) {
            super(new FileRunner(file));
      }// конструктор, который создает объект
}
class FileRunner implements Runnable {
      private String fullName;
      public FileRunner(String fileName) {
            this.fullName = fileName; }// переданное имя файла
            /**эмуляция ввода имени файла с клавиатуры
             */
      public void run() {
            try {
                  Thread.sleep(3000); // дожидается выполнения команды clickAt
в методе chooseFile
                  Robot robot = new Robot();
                  for (char c : fullName.toCharArray()) {
                        if (c == ':') {
                              robot.keyPress(KeyEvent.VK_SHIFT);
                              robot.keyPress(KeyEvent.VK_SEMICOLON);
                              robot.keyRelease(KeyEvent.VK_SHIFT);
                        } else if (c == '/') {
                              robot.keyPress(KeyEvent.VK_BACK_SLASH);
                        } else {
                              robot.keyPress(KeyStroke.getKeyStroke(
                              Character.toUpperCase(c), 0).getKeyCode());
                        }
                  }
                  robot.keyPress(KeyEvent.VK_ENTER);
            } catch (Exception e) { throw new RuntimeException(e);}
      }
}
```

Дополнение 22. Эмуляция ввода данных с клавиатуры

```
package my.testng;
import org.testng.annotations.Test;
public class ArchiveScheme extends kleioBase {
	@Test
	public void testArchiveScheme() throws Exception {
		selenium.click("link=Test_2");
		selenium.waitForPageToLoad("30000");
		selenium.click("//a[contains(text(),'Arhiiviskeemid')]");
		verifyEquals(selenium.getAttribute("//span[@id=
'projectArchiveStructures']/@class"),"active");
		selenium.click("//table[@class='t-data-
grid']/tbody/tr[contains(@class,'t-first')]/td/a");
		for (int second = 0;; second++) {
			if (second >= 60)
				fail("timeout");
			try {
				if (selenium.isElementPresent(
"//div[@id='archiveTree']/ul/li[contains(@id, '.xml')]"))
					break;
			} catch (Exception e) {
			}
			Thread.sleep(1000);
		}
		String leafPath = "//div[@id='archiveTree']/ul[@class='ltr']/li";
		String nextLeaf = "/ul/li[1]";
		selenium.click(leafPath);
		try {
			while (selenium.getAttribute(leafPath + nextLeaf +
"/@class").matches("^closed[\\s\\S]*$")) {
				for (int second = 0;; second++) {
					if (second >= 60)
						fail("timeout");
					try {
						if (selenium.isElementPresent(leafPath +=
nextLeaf))
							break;
					} catch (Exception e) {
					}
					Thread.sleep(1000);
				}
```

```
                        selenium.click(leafPath);
                  }
            } catch (Exception e) {
            }
            for (int second = 0;; second++) {
                  if (second >= 60)
                        fail("timeout");
                  try {
                        if (selenium.isElementPresent(leafPath + "/a/ins"))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
            for (int second = 0;; second++) {
                  if (second >= 60)
                        fail("timeout");
                  try {
                        if ("last leaf".equals(selenium.getAttribute(leafPath
                                    + "@class")))
                              break;
                  } catch (Exception e) {
                  }
                  Thread.sleep(1000);
            }
            selenium.click(leafPath + "/a");
            verifyTrue(selenium.getAttribute("//iframe[@id=
'ASDetailViewFrame']/@src").matches("^[\\s\\S]*\\.xml[\\s\\S]*$"));
      }
}
```

Дополнение 23. *ArchiveScheme* класс реализует навигацию по архивной схеме (Тест 3)

```
package my.testng;
import org.testng.annotations.Test;
public class stuffUpload extends kleioBase {
      @Test
      public void testStuffUpload() throws Exception {
            selenium.click("//tr[@class='t-last']/td[1]/a");
            selenium.waitForPageToLoad("30000");
            selenium.click("//a[contains(text(),'Aines')]");
            verifyEquals(selenium.getAttribute("//span[@id='projectMaterial']
/@class"), "active");
            selenium.type("//input[contains(@id,'dataSource')]",
"C:\\Arhiiviaines\\Aines1");
            selenium.click("loadSip");
            verifyTrue(selenium.isElementPresent("//div[@class='message']"));
            try{ while (selenium.getText("//div[@class='message']
/span").matches("^Laen andmeid kaustast[\\s\\S]*$")) {selenium.refresh();
            } } catch (Exception e){}
            verifyFalse(selenium.isElementPresent("//div[@class='message']"));
      }
}
```

Дополнение 24. *stuffUpload* класс реализует загрузку данных (Тест 4)

```
package my.testng;
import org.testng.annotations.Test;
public class DataDrivenBase extends kleioBase {
	@Test
	public void testDataDrivenBase() throws Exception {
		selenium.click("//tr[@class='t-last']/td[1]/a");
		selenium.waitForPageToLoad("30000");
		selenium.click("//a[contains(text(),'Aines')]");
		verifyEquals(selenium
				.getAttribute("//span[@id='projectMaterial']/@class"),
"active");
	}
}
```

Дополнение 25. Базовый класс *DataDrivenBase* для загрузки данных из *excel*-файла

```
package my.testng;
import java.io.File;
public class stuffDataDriven extends DataDrivenBase {
	@DataProvider(name = "DP1")
	public Object[][] createData1() {
		Object[][] retObjArr = getTableArray(
				"resources/data2.xls", "DataPool", "stuffTestData");
		return (retObjArr);
	}
	@Test(dataProvider = "DP1")
	public void testStuffDataDriven(String stuffUpload) {
		selenium.type("//input[contains(@id,'dataSource')]", stuffUpload);
		selenium.click("loadSip");
		verifyTrue(selenium.isElementPresent("//div[@class='message']"));
		try {while
(selenium.getText("//div[@class='message']/span").matches(
		"^Laen andmeid kaustast[\\s\\S]*$")) {
				selenium.refresh();
		} } catch (Exception e){}
		verifyFalse(selenium.isElementPresent("//div[@class='message']"));
	}

	public String[][] getTableArray(String xlFilePath, String sheetName,
			String tableName) {
		String[][] tabArray = null;
		try {
			Workbook workbook = Workbook.getWorkbook(new
File(xlFilePath));
			Sheet sheet = workbook.getSheet(sheetName);
			int startRow, startCol, endRow, endCol, ci, cj;
			Cell tableStart = sheet.findCell(tableName);
			startRow = tableStart.getRow();
			startCol = tableStart.getColumn();
			Cell tableEnd = sheet.findCell(tableName, startCol + 1,
					startRow + 1, 100, 64000, false);
			endRow = tableEnd.getRow();
			endCol = tableEnd.getColumn();
```

```
                  System.out.println("startRow=" + startRow + ", endRow=" +
endRow + ", " + "startCol=" + startCol + ", endCol=" + endCol);
                  tabArray = new String[endRow - startRow - 1][endCol -
startCol - 1];
                  ci = 0;
                  for (int i = startRow + 1; i < endRow; i++, ci++) {
                        cj = 0;
                        for (int j = startCol + 1; j < endCol; j++, cj++) {
                              tabArray[ci][cj] = sheet.getCell(j,
i).getContents();
                        }
                  }
            } catch (Exception e) {
                  System.out.println("error in getTableArray()");
            }
            return (tabArray);
      }
}
```

Дополнение 26. *stuffDataDriven* класс реализует загрузку данных из *excel*-файла (Тест 5)

Дополнение 27. Отчет об ошибке в тесте номер 2 для браузера *Internet Explorer*