Tellurium

User Guide

Tellurium Automated Testing Framework (Tellurium)

09/17/09 Rel. 0.6.0 Vers. 2.0

How to Use This Guide

Audience

The Tellurium User Guide is written for the software developer who is testing code and who needs robust, expressive, flexible, and reusable testing software methods found in Tellurium that provides the environment for that task.

Content Summary

Chapter 1 Introduction: Discusses the following:

- Motivation for creating Tellurium, the Tellurium approach for Web testing
- Challenges and problems addressed by Tellurium.
- Tellurium concepts and features: Introduces and describes the following:
  - UI objects, UI Module (heart of Tellurium), UiID (the UI Object)
  - Two types of Composite Locators
  - Group Locating which is an attribute of the Tellurium UI module
  - UI Templates and the use of Table and List
  - Javascript Events using the DOJO JavaScript? framework
  - Use of the Logical Container in Tellurium
  - Use of jQuery Selector which improves the test speed in IE
  - Table Methods
  - jQuery Selector Cache Option mechanism in Tellurium
  - Use of the Include syntax in Tellurium
  - Use of setCustomConfig for project level test settings
  - Use of Custom Selenium Extensions with TelluriumConfig?.groovy
  - Use of Data Driven Testing in Tellurium
  - Tellurium Dump Method for printing UI objects and descendants runtime locators
  - Tellurium Selenium grid support
  - Tellurium Core Mock Http Server for testing HTML sources directly
  - Tellurium's three ways to provide testing support

Chapter 2 Getting Started: Provides a description of the three Tellurium methods for creating a Tellurium project. In addition, Tellurium UI Objects are described and discussed including each pre-defined UI Object.

Chapter 3 Tellurium Subprojects: Describes the multiple sub-projects including Reference Projects, Maven Archetypes, TrUMP, Widget Extensions, Core, and Engine projects.

Chapter 4 Tellurium Architecture: Describes the Tellurium architecture and discusses how Tellurium works as a framework for testing.

Chapter 5 Tellurium APIs: Provides tables of various API methods and the action/result of each method when used. The methods discussed are for DSL, Data Access and Test Support DSLs.

Appendices

Appendix A Examples: Provides examples of the following Tellurium test cases:

- Custom UI Object
- UI Module
- Groovy Test Case
- Java Test Case
- Data Drive Testing
- DSL Test Script

Appendix B FAQs: Provides a list of Tellurium frequently asked questions and answers.

Appendix C Tellurium Support TBD

Appendix D Resources TBD

Glossary

Table of Contents

1 Introduction

Motivation

Automated web testing has always been one of the hottest and most important topics in the software testing arena when it comes to the rising popularity of Rich Internet Applications (RIA) and Ajax-based web applications. With the advent of new web techniques such as RIA and Ajax, automated web testing tools must keep current with changes in technology and be able to address the following challenges:

- **JavaScript Events:** JavaScript is everywhere on the web today. Many web applications are JavaScript heavy. To test JavaScript, the automated testing framework should be able to trigger JavaScript events in a convenient way.
- **Ajax for Dynamic Web Content**: Web applications have many benefits over desktop applications. For example, these applications have no installation and updates are instantaneous and easier to support. Ajax is a convenient way to update a part of the web page without refreshing the whole page. AJAX makes web applications richer and more user-friendly. The web context for an Ajax application is usually dynamic. For example, in a data grid, the data and number of rows keeps changing at runtime.
- **Robust/Responsive to Changes**: A good automated web-testing tool should be able to address the changes in the web context to some degree so that users do not need to keep updating the test code.
- **Easy to Maintain**: In an agile testing world, software development is based on iterations, and new features are added on in each sprint. The functional tests or user acceptance tests must be refactored and updated for the new features. The testing framework should provide the flexibility for users to maintain the test code easily.
- **Re-usability**: Many web applications use the same UI module for different parts of the application. The adoption of JavaScript frameworks such as Dojo and ExtJS increases the chance of using the same UI module for different web applications. A good testing framework should also be able to provide the re-usability of test modules.
- **Expressiveness**: The testing framework provides users without much coding experience the ability to easily write test code or scripts in a familiar way, such as using a domain specific language (DSL).

The Tellurium Automated Testing Framework (Tellurium) is designed around these considerations and has defined as its focus the following goals:

- Robust/responsive to changes; allow changes to be localized
- Addresses dynamic web contexts such as JavaScript events and Ajax
- Easy to refactor and maintain
- Modular; test modules are reusable
- Expressive and easy to use

Tellurium, the New Approach for Web Testing

The Tellurium Automated Testing Framework (Tellurium) is an open source automated testing framework for web applications that addresses the challenges and problems of today's web testing.

Most existing web testing tools/frameworks focus on individual UI elements such as links and buttons. Tellurium takes a new approach for automated web testing by using the concept of the UI module.

The *UI module* is a collection of UI elements grouped together. Usually, the UI module represents a composite UI object in the format of nested basic UI elements. For example, the Google search UI module can be expressed as follows:

```
ui.Container(uid: "GoogleSearchModule", clocator: [tag: "td"], group:
"true"){
   InputBox(uid: "Input", clocator: [title: "Google Search"])
   SubmitButton(uid: "Search", clocator: [name: "btnG", value: "Google
Search"])
   SubmitButton(uid: "ImFeelingLucky", clocator: [value: "I'm Feeling
Lucky"])
}
```

Tellurium is built on the foundation of the UI module. The UI module makes it possible to build locators for UI elements at runtime. First, this makes Tellurium robust and responsive to changes from internal UI elements. Second, the UI module makes Tellurium expressive. UI elements can be referred to simply by appending the names (uid) along the path to the specific element. This also enables *Tellurium's Group Locating* feature, making composite objects reusable, and addressing dynamic web pages.

Tellurium is implemented in Groovy and Java. The test cases can be written in Java, Groovy, or pure Domain Specific Language (DSL) scripts. Tellurium evolved out of Selenium. However, the UI testing approach is completely different. Tellurium is not a "record and replay" style framework, and it enforces the separation of UI modules from test code, making refactoring easy.

For example, once the Google Search UI module is defined as previously shown, the test code is written as follows:

```
type "GoogleSearchModule.Input", "Tellurium test"
click "GoogleSearchModule.Search"
```

Tellurium sets the Object to Locator Mapping (OLM) automatically at runtime so that UI objects can be defined simply by their attributes using Composite Locators. Tellurium uses the Group Locating Concept (GLC) to exploit information inside a collection of UI components so that locators can find their elements.

Tellurium also defines a set of DSLs for web testing. Furthermore, Tellurium uses UI templates to define sets of dynamic UI elements at runtime. As a result, Tellurium is robust, expressive, flexible, reusable, and easy to maintain.

The main features of Tellurium include:

- Abstract UI objects to encapsulate web UI elements
- UI module for structured test code and re-usability
- DSL for UI definition, actions, and testing
- Composite Locator to use a set of attributes to describe a UI element
- Group locating to exploit information inside a collection of UI components
- Dynamically generate runtime locators to localize changes
- UI templates for dynamic web content
- XPath support
- jQuery selector support to improve test speed in IE
- Locator caching to improve speed
- Javascript event support
- Use Tellurium Firefox plugin, Trump, to automatically generate UI modules
- Dojo and ExtJS widget extensions
- Data driven test support
- Selenium Grid support
- JUnit and TestNG support
- Ant and Maven support

How Challenges and Problems Are Addressed in Tellurium

First, Tellurium does not use "record and replay." Instead, it uses the Tellurium Firefox plugin TrUMP to generate the UI module (not test code) for you. Then test code based on the UI module is created.

In this way, the UI and the test code are decoupled. The structured test code in Tellurium makes it much easier to refactor and maintain the code.

The composite locator uses UI element attributes to define the UI, and the actual locator (for example, XPath or jQuery selector), is generated at runtime. Any updates to the composite locator lead to different runtime locators, and the changes inside the UI module are localized.

The Group locating is used to remove the dependency of the UI objects from external UI elements (for example, external UI changes do not affect the current UI module for most cases), so that test code is robust and responsive to changes up to a certain level.

Tellurium uses the *respond* attribute in a UI object to specify JavaScript events, and the rest is handled automatically by the framework itself.

UI templates are a powerful feature in Tellurium used to represent many identical UI elements or a dynamic size of different UI elements at runtime. This is extremely useful in testing dynamic web contexts such as a data grid.

The *Option* UI object is designed to automatically address dynamic web contexts with multiple possible UI patterns.

Re-usability is achieved by the UI module when working within one application and by Tellurium Widgets when working across different web applications. With the Domain Specific Language (DSL) in Tellurium, UI modules can be defined and test code written in a very expressive way.

Tellurium also provides flexibility to write test code in Java, Groovy, or pure DSL scripts.

2 Getting Started

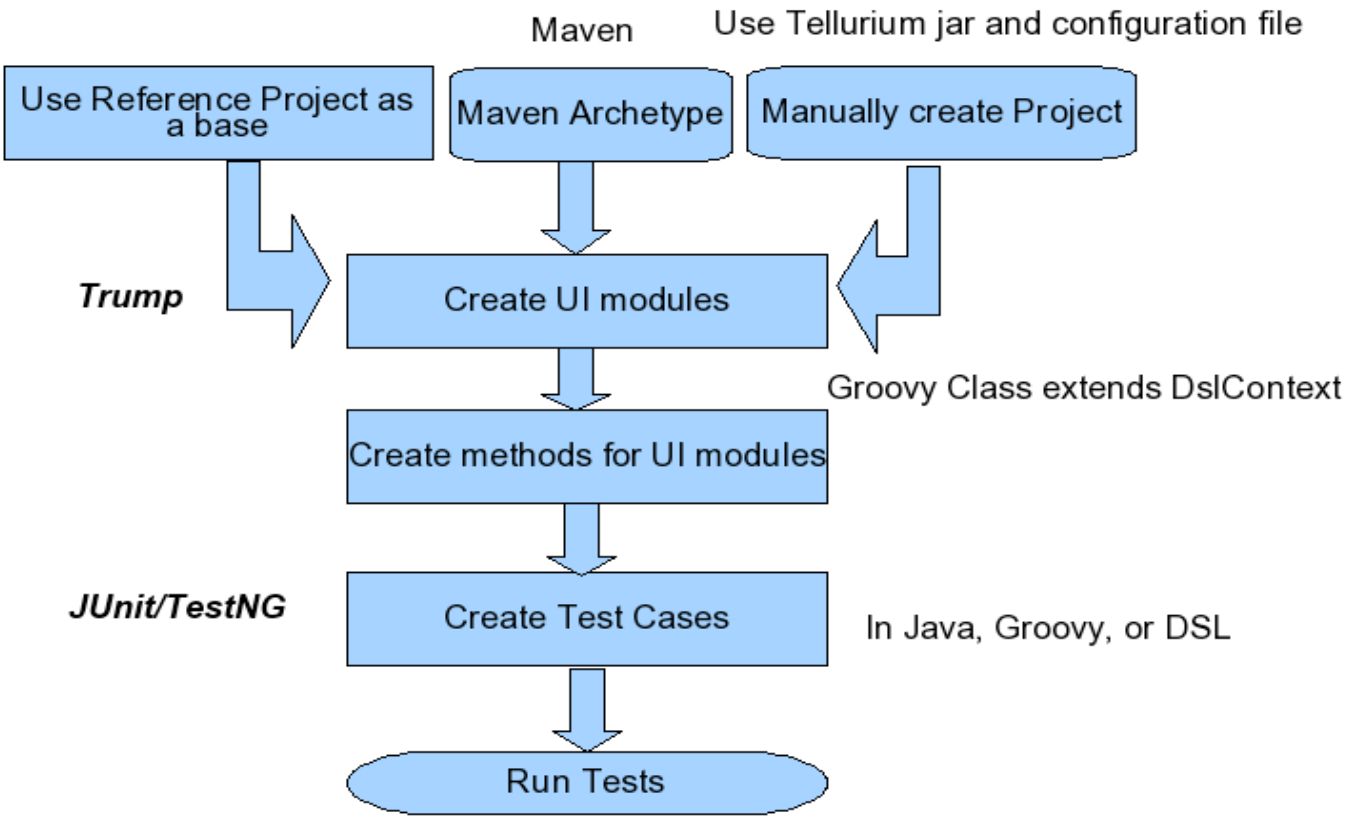
This chapter discusses the Tellurium methods for creating a Tellurium project followed by descriptions of the primary components used for testing the newly created web framework including the UI Module, UI Object and attributes, Logical Container, jQuery Selector, UI Templates, and UI Testing Support.

Create a Tellurium Project

Create a Tellurium Project in one of the following methods:

- Use the reference project as a base
- Use the Tellurium Maven archetype
- Manually create a Tellurium project using the tellurium jar and a Tellurium configuration file
- Alternatively, create a Tellurium Maven project manually using the sample POM file. (See Figure 2-1.)

Figure 2-1 Sample POM File



Tellurium Maven Archetypes

The easiest way to create a Tellurium project is to use Tellurium Maven archetypes. Tellurium provides two Maven archetypes:

- tellurium-junit-archetype
- tellurium-testng-archetype (for Tellurium JUnit test projects and Tellurium TestNG test projects respectively.)

As a result, a user can create a Tellurium project using one Maven command.

For a Tellurium JUnit project, use:

```
mvn archetype:create -DgroupId=your_group_id -
DartifactId=your_artifact_id \
     -DarchetypeArtifactId=tellurium-junit-archetype -
DarchetypeGroupId=tellurium \
     -DarchetypeVersion=0.7.0-SNAPSHOT \
     -
```

```
DarchetypeRepository=http://kungfuters.org/nexus/content/repositories/sn
apshots
```

For a Tellurium TestNG project, use:

```
mvn archetype:create -DgroupId=your_group_id -
DartifactId=your_artifact_id \
     -DarchetypeArtifactId=tellurium-testng-archetype -
DarchetypeGroupId=tellurium \
     -DarchetypeVersion=0.7.0-SNAPSHOT \
     -
DarchetypeRepository=http://kungfuters.org/nexus/content/repositories/sn
apshots
```

For an Ant user:

1. Download the following:
   - Tellurium core 0.6.0 jar file from the Tellurium project download page
   - Tellurium dependency file
   - Tellurium configuration file
2. Unpack the Tellurium dependency file to your project /lib directory together with the Tellurium core 0.6.0 jar file.
3. Name the Tellurium configuration file as TelluriumConfig.groovy and place it in the project root directory.

For Ant build scripts, refer to the sample Tellurium Ant build scripts.

Setup Tellurium Project in IDEs

A Tellurium Project can be run in IntelliJ, NetBeans, Eclipse, or other IDEs that have Groovy support.

If using Maven, open the POM file to let the IDE automatically build the project files.

IntelliJ IDEA is commercial and a free trial version for 30 days that can be downloaded from http://www.jetbrains.com/idea/download/index.html. A detailed guide is found on How to create your own Tellurium testing project with IntelliJ 7.0.

For NetBeans users, detailed Guides can be found on the NetBeans Starters' guide page and How to create your own Tellurium testing project with NetBeans 6.5.

For Eclipse users, download the Eclipse Groovy Plugin from http://dist.codehaus.org/groovy/distributions/update/ to run the Tellurium project.

For detailed instructions, read How to create your own Tellurium testing project with Eclipse.

Create a UI Module

Tellurium provides TrUMP to automatically create UI modules. Trump can be downloaded from the Tellurium project site:

http://code.google.com/p/aost/downloads/list

Choose the Firefox 2 or Firefox 3 version depending upon the user's Firefox version, or download the Firefox 3 version directly from the Firefox addons site at:

https://addons.mozilla.org/en-US/firefox/addon/11035

Once installed, restart Firefox. Record UI modules by simply clicking the UI element on the web and then click the "Generate" button.

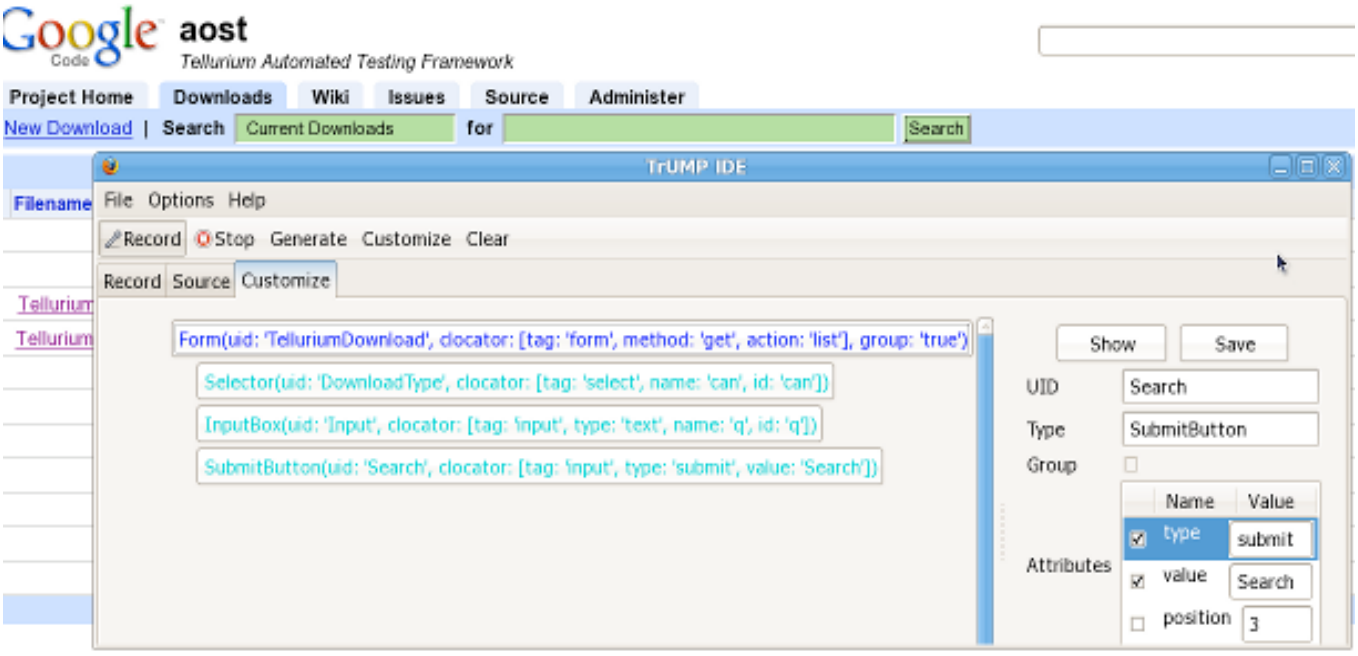
To customize the UI, click the "Customize" button.

In the example, open the Tellurium download page found on:

http://code.google.com/p/aost/downloads/list

Record the download search module as follows: (See Figure 2-2.)

Figure 2-2 Search Module



After the UI module is customized, export it as the module file NewUiModule?.groovy to the demo project and add a couple of methods to the class:

```
class NewUiModule extends DslContext {

  public void defineUi() {
    ui.Form(uid: "TelluriumDownload", clocator: [tag: "form", method:
"get", action: "list"],
            group: "true")
{
      Selector(uid: "DownloadType", clocator: [tag: "select", name:
"can", id: "can"])
      InputBox(uid: "Input", clocator: [tag: "input", type: "text",
name: "q", id: "q"])
      SubmitButton(uid: "Search", clocator: [tag: "input", type:
"submit", value: "Search"])
    }
  }

  //Add your methods here
  public void searchDownload(String keyword) {
    keyType "TelluriumDownload.Input", keyword
```

```
    click "TelluriumDownload.Search"
    waitForPageToLoad 30000
  }

  public String[] getAllDownloadTypes() {
    return getSelectOptions("TelluriumDownload.DownloadType")
  }

  public void selectDownloadType(String type) {
    selectByLabel "TelluriumDownload.DownloadType", type
  }
}
```

Create Tellurium Test Cases

Once the UI module is created, create a new Tellurium test case NewTestCase by extending TelluriumJavaTestCase class.

```
public class NewTestCase extends TelluriumJavaTestCase {
    private static NewUiModule app;

    @BeforeClass
    public static void initUi() {
        app = new NewUiModule();
        app.defineUi();
    }

    @Before
    public void setUpForTest() {
        connectUrl("http://code.google.com/p/aost/downloads/list");
    }

    @Test
    public void testTelluriumProjectPage() {
        String[] allTypes = app.getAllDownloadTypes();
        assertNotNull(allTypes);
        assertTrue(allTypes[1].contains("All Downloads"));
        app.selectDownloadType(allTypes[1]);
        app.searchDownload("TrUMP");
    }
```

```
}
```

Compile the project and run the new test case.

UI Module

The UI Module is the heart of Tellurium. The UI module is a collection of UI elements grouped together. Usually, the UI module represents a composite UI object in the format of nested basic UI elements. For example, the download search module in Tellurium's project site is defined as follows:

```
ui.Form(uid: "downloadSearch", clocator: [action: "list", method:
"get"], group: "true") {
   Selector(uid: "downloadType", clocator: [name: "can", id: "can"])
   InputBox(uid: "searchBox", clocator: [name: "q"])
   SubmitButton(uid: "searchButton", clocator: [value: "Search"])
}
```

Tellurium is built on the foundation of the UI module. The UI module makes it possible to build locators for UI elements at runtime. First, this makes Tellurium robust and responsive to changes from internal UI elements. Second, the UI module makes Tellurium expressive.

A UI element is referred to simply by appending the names (uids) along the path to the specific element. This also enables Tellurium's "Group Locating" feature, making composite objects reusable and addressing dynamic web pages.

This frees up the testers to write better tests rather than spending precious testing time identifying and resolving test failures due to XPath changes.

Basic UI Object

Tellurium provides a set of predefined UI objects to be used directly when setting up a test.

The basic UI object is an abstract class. Users cannot instantiate it directly. The basic UI Object works as the base class for all UI objects and it includes the following attributes:

ATTRIBUTE	DESCRIPTION
UI Object	Basic Tellurium component
UiId	UI object's identifier
Namespace	Used for XHTML
Locator	UI Object Locator including Base Locator and Composite Locator
Group	Group Locating attribute that only applies to a collection type of UI object such as Container, Table, List, Form
Respond	JavaScript events the UI object responds to.

The value is a list and the base UI object provides the following methods:

- boolean isElementPresent()
- boolean isVisible()
- boolean isDisabled()
- waitForElementPresent(int timeout), where the time unit is ms.
- waitForElementPresent(int timeout, int step)
- String getText()
- getAttribute(String attribute)

All UI Objects inherit the above attributes and methods. Do not call these methods directly but use DSL syntax instead.

For example, use:

```
click "GoogleSearchModule.Search"
```

In this way, Tellurium first maps the UIID "`GoogleSearchModule.Search`" to the actual UI Object. Then the user calls the click method on the UI Object. If that UI Object does not have the click method defined, an error displays.

UI Object Default Attributes

Tellurium UI objects have default attributes as shown in Table 2-1:

Table 2-1 UI Object Default Attributes

Tellurium Object	Locator Default Attributes	Extra Attributes	UI Template
Button	tag: "input"		no
Container		group	no
CheckBox	tag: "input", type: "checkbox"		no
Div	tag: "div"		no
Form	tag: "form"	group	no
Image	tag: "img"		no
InputBox	tag: "input"		no
RadioButton	tag: "input", type: "radio"		no
Selector	tag: "select"		no
Span	tag: "span"		no
SubmitButton	tag: "input", type: "submit"		no
UrlLink	tag: "a"		no
List		separator	yes
Table	tag: "table"	group, header	yes
StandardTable	tag: "table"	group, header, footer	yes
Frame		group, id, name, title	no
Window		group, id, name, title	no

UI Object List Description

Button

Button represents various Buttons on the web and its default tag is "input". The following methods can be applied to Button:

- click()
- doubleClick()
- clickAt(String coordination)

Example:

```
Button(uid: "searchButton", clocator: [value: "Search", name: "btn"])
```

Submit Button

SubmitButton is a special Button with its type being "submit".

Example:

```
SubmitButton(uid: "search_web_button", clocator: [value: "Search the
Web"])
```

Check Box

The CheckBox on the web is abstracted as "CheckBox" Ui object. The default tag for CheckBox is "input" and its type is "checkbox". CheckBox comes with the following methods:

- check()
- boolean isChecked()
- uncheck()
- String getValue()

Example:

```
CheckBox(uid: "autoRenewal", clocator: [dojoattachpoint:
'dap_auto_renew'])
```

Div

Div is often used in the Dojo framework and it can represent many objects. Its tag is "div" and it has the following method:

click()

Example:

```
Div(uid: "dialog", clocator: [class: 'dojoDialog', id:
'loginDialog'])
```

Image

Image is used to abstract the "img" tag and it comes with the following additional methods:

- getImageSource()
- getImageAlt()
- String getImageTitle()

Example:

```
Image(uid: "dropDownArrow", clocator: [src: 'drop_down_arrow.gif'])
```

Icon

Icon is similar to the Image object, but user can perform actions on it. As a result, it can have the following additional methods:

- click()
- doubleClick()
- clickAt(String coordination)

Example:

```
Icon(uid: "taskIcon", clocator:[tag: "p", dojoonclick: 'doClick',
img: "Show_icon.gif"] )
```

Radio ButtText Box on

RadioButton is the abstract object for the Radio Button Ui. As a result, its default tag is "input" and its type is "radio". RadioButton has the following additional methods:

- check()
- boolean isChecked()
- uncheck()
- String getValue()

Example:

```
RadioButton(uid: "autoRenewal", clocator: [dojoattachpoint:
'dap_auto_renew'])
```

Text Box

TextBox is the abstract Ui object from which the user returns to the text. For example, it comes with the following method:

String waitForText(int timeout)

Note: TextBox can have various types of tags.

Example:

```
TextBox(uid: "searchLabel", clocator: [tag: "span"])
```

Input Box

InputBox is the Ui where user types in input data. As its name stands, InputBox's default tag is "input". InputBox has the following additional methods:

- type(String input)
- keyType(String input), used to simulate keyboard typing
- typeAndReturn(String input)
- clearText()
- boolean isEditable()
- String getValue()

Example:

```
InputBox(uid: "searchBox", clocator: [name: "q"])
```

URL Link

UrlLink stands for the web url link, i.e., its tag is "a". UrlLink has the following additional methods:

- String getLink()
- click()
- doubleClick()
- clickAt(String coordination)

Example:

```
UrlLink(uid: "Grid", clocator: [text: "Grid", direct: "true"])
```

Selector

Selector represents the Ui with tag "select". The user can select from a set of options. Selector has methods, such as:

- selectByLabel(String target)
- selectByValue(String value)
- addSelectionByLabel(String target)
- addSelectionByValue(String value)
- removeSelectionByLabel(String target)
- removeSelectionByValue(String value)
- removeAllSelections()
- String getSelectOptions()
- String getSelectedLabels()
- String getSelectedLabel()
- String getSelectedValues()
- String getSelectedValue()
- String getSelectedIndexes()
- String getSelectedIndex()
- String getSelectedIds()

- String getSelectedId()
- boolean isSomethingSelected()

Example:

```
Selector(uid: "issueType", clocator: [name: "can", id: "can"])
```

Container

Container is an abstract object that can hold a collection of Ui objects. As a result, the Container has a special attribute "useGroupInfo" and its default value is false. If this attribute is true, the Group Locating is enabled. But make sure all the Ui objects inside the Container are children nodes of the Container in the DOM, otherwise, you should not use the Group Locating capability.

Example:

```
ui.Container(uid: "google_start_page", clocator: [tag: "td"], group:
"true"){
    InputBox(uid: "searchbox", clocator: [title: "Google Search"])
    SubmitButton(uid: "googlesearch", clocator: [name: "btnG", value:
"Google Search"])
    SubmitButton(uid: "Imfeelinglucky", clocator: [value: "I'm
Feeling Lucky"])
}
```

Form

Form is a type of Container with its tag being "form" and it represents web form. Like Container, it has the capability to use Group Locating and it has a special method:

submit()

This method is useful and can be used to submit input data if the form does not have a submit button.

Example:

```
ui.Form(uid: "downloadSearch", clocator: [action: "list", method:
"get"], group: "true") {
    Selector(uid: "downloadType", clocator: [name: "can", id: "can"])
    TextBox(uid: "searchLabel", clocator: [tag: "span"])

    InputBox(uid: "searchBox", clocator: [name: "q"])
    SubmitButton(uid: "searchButton", clocator: [value: "Search"])
}
```

Table

Table is one of the most complicated Ui Object and also the most often used one. Obviously, its tag is "table" and a table can have headers besides rows and columns.

Table is a good choice for a data grid. Tellurium can handle its header, rows, and columns automatically. Table has one important feature: a different UiID than other Ui objects.

For example, if the id of the table is "table1", then its i-th row and j-th column is referred as `"table1[i][j]"` and its m-th header is `"table1.header[m]"`.

Another distinguished feature of Table is that a user can define Ui templates for its elements.

For example, the following example defines different table headers and the template for the first column, the element on the second row and the second column, and the template for all the other elements in other rows and columns.

```
ui.Table(uid: "downloadResult", clocator: [id: "resultstable", class:
"results"],
         group: "true")
{
    //define table header
    //for the border column
    TextBox(uid: "header: 1", clocator: [:])
    UrlLink(uid: "header: 2", clocator: [text: "%%Filename"])
    UrlLink(uid: "header: 3", clocator: [text: "%%Summary + Labels"])
    UrlLink(uid: "header: 4", clocator: [text: "%%Uploaded"])
    UrlLink(uid: "header: 5", clocator: [text: "%%Size"])
    UrlLink(uid: "header: 6", clocator: [text: "%%DownloadCount"])
    UrlLink(uid: "header: 7", clocator: [text: "%%..."])

    //define Ui object for the second row and the second column
    InputBox(uid: "row: 2, colum: 2" clocator: [:])
    //define table elements
    //for the border column
    TextBox(uid: "row: *, column: 1", clocator: [:])
    //For the rest, just UrlLink
    UrlLink(uid: "all", clocator: [:])
}
```

Be aware, the templates inside the Table follow the name convention:

- For the i-th row, j-th column, the uid should be "row: i, column: j"
- The wild case for row or column is "*"
- "all" stands for matching all rows and columns

As a result, "**row** : *, **column** : **3**" refers to the 3rd column for all rows. Once the templates are defined for the table, Tellurium uses a special way to find a matching for a Ui element **"tj able[i][]"** in the table.

For example, the following rules apply:

- Tellurium tries to find the template defined for the i-th row, j-th column.
- If not found, Tellurium tries to search for a general template "row: *, column: j", for example, the template for column j.
- If not found, Tellurium tries to search for another general template "row: i, column: *", i.e., the template for row i.
- If not found either, Tellurium tries to find the template matching all rows and columns.
- If still out of luck, Tellurium uses a TextBox? as the default element for this element.

Generally speaking, Tellurium always searches for the special case first, then broadening the search for more general cases, until all matching cases are found. In this way, user can define very flexible templates for tables.

Table is a type of Container and thus, it can use the Group Locating feature. Table has the following special methods:

- boolean hasHeader()
- int getTableHeaderColumnNum()
- int getTableMaxRowNum()
- int getTableMaxColumnNum()

From Tellurium 0.6.0, you can also specify the tbody attribute for the Table object. This may be helpful if the user has multiple tbody elements inside a single table tab.

For example, specify `tbody` as follows:

```
Container(uid: "tables", clocator:[:]){
     Table(uid: "first", clocator: [id: "someId", tbody: [position:
"1"]]){
          ......
      }
     Table(uid: "second", clocator: [id: "someId", tbody: [position:
"2"]]){
          ......
      }

      ...

}
```

UiID Attribute

In Tellurium, the UI object is referred to by its UiID or the UI object dentifier.

For nested Ui objects, the UiID of the UI Object is a concatenated UI objects' uids along its path to the UI Object.

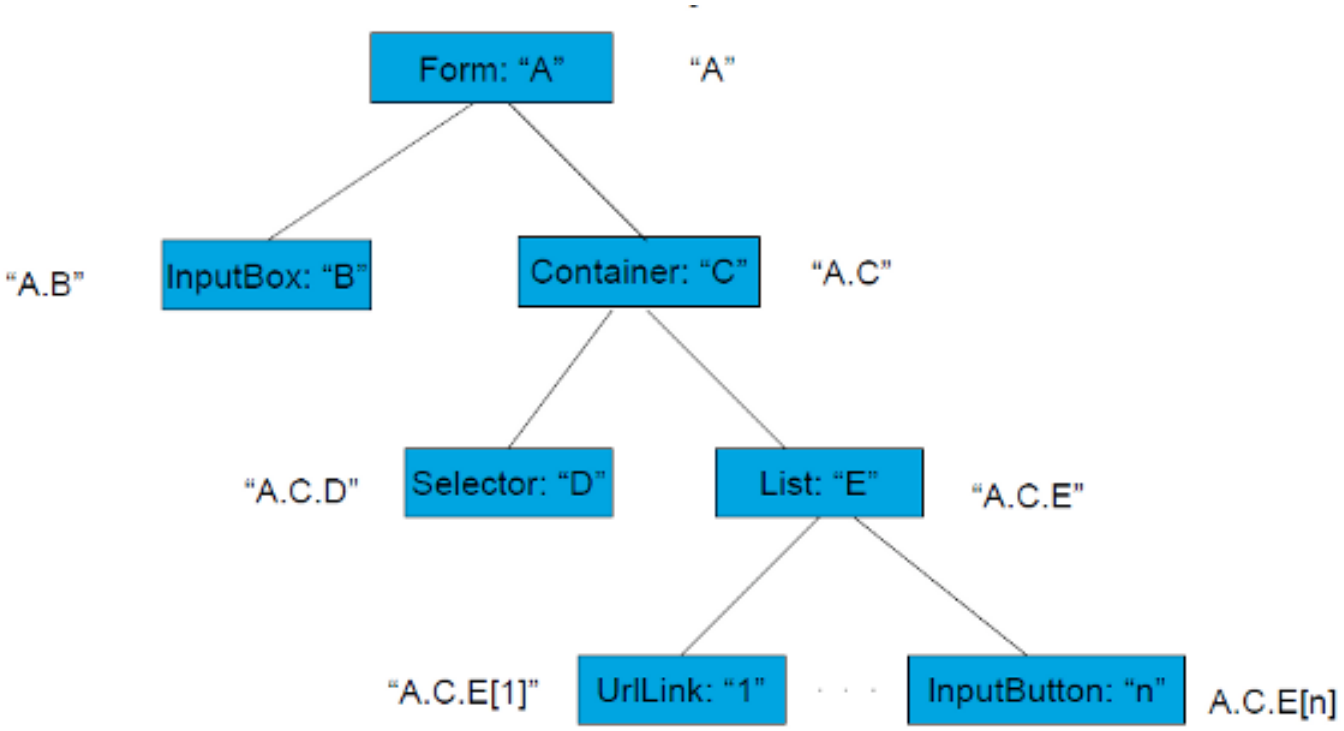
For example, in the following nested UI Module shown below, the TextBox is referred to as the "`parent_ui.child_ui.grand_child.textbox1`".

```
ui.Container(uid: "parent_ui"){
   InputBox(uid: "inputbox1", locator: "...")
   Button(uid: "button1", locator: "...")
   Container(uid: "child_ui"){
     Selector(uid: "selector1", locator: "...")
     ...
     Container(uid: "grand_child"){
       TextBox(uid: "textbox1", locator: "...")
       ...
     }
   }
}
```

The exceptions are tables and lists, which use `[x][y]` or `[x]` to reference the elements inside. For example, `labels_table[2][1]` and `GoogleBooksList.subcategory[2]`. The Table header can be referred in the format of `issueResult.header[2]`.

More general cases are shown in Figure 2-3 General Cases Search Example:

Figure 2-3: General Cases Search Example



For example, the UiID of the List *E* in the above diagram is `A.C.E` and the InputButton in the List *E* is referred by its index n. For example: A.C.E[n].

Locator Attributes

Tellurium supports two types of UI Object locators:

1. Base locator
2. Composite locator

The *Base locator* is a relative XPath.

The *Composite locator*, denoted by "`clocator`", specifies a set of attributes for the UI object. The actual locator is derived automatically by Tellurium at runtime.

The *Composite locator* is defined as follows:

```
class CompositeLocator {
    String header
    String tag
    String text
    String trailer
    def position
    boolean direct
    Map<String, String> attributes = [:]
}
```

To use the Composite locator, use "clocator" with a map as its value.

For example:

```
clocator: [key1: value1, key2: value2, ...]
```

The default attributes include "header", "tag", "text", "trailer", "position", and "direct". They are all optional. The "direct" attribute specifies whether this UI object is a direct child of its parent UI, and the default value is "false".

If there are additional attributes, they are defined in the same way as the default attributes. For example:

```
clocator: [tag: "div", value: "Tellurium home"]
```

Most Tellurium objects come with default values for certain attributes. For example, the tag attribute.

If these attributes are not specified, the default attribute values are used. In other words, if the default attribute values of a Tellurium UI object are known, omit them in clocator.

For example, if the RadioButtonhttp://code.google.com/p/aost/w/edit/RadioButton Object's

default tag is "input," and the default type is "radio," omit them and write the clocator as follows:

```
clocator: [:]
```

which is equivalent to:

```
clocator: [tag: "input", type: "radio"]
```

Group Attribute: Group Locator

In the Tellurium UI module, the "group" attribute is seen often. For example:

```
ui.Container(uid: "google_start_page", clocator: [tag: "td"], group:
"true"){
  InputBox(uid: "searchbox", clocator: [title: "Google Search"])
  SubmitButton(uid: "googlesearch", clocator: [name: "btnG", value:
"Google Search"])
  SubmitButton(uid: "Imfeelinglucky", clocator: [value: "I'm Feeling
Lucky"])
}
```

The group attribute is a flag for the Group Locating Concept. Usually, the XPath generated by Selenium IDE, XPather, or other tools is a single path to the target node such as:

```
//div/table[@id='something']/div[2]/div[3]/div[1]/div[6]
```

Sibling node information is not used in this example as the XPath depends too much on information from nodes far away from the target node. In Tellurium, every effort is made to localize the information and reduce this dependency by using sibling information or local information.

For example, in the above google UI module example, the group locator concept searches for the location of the "td" tag with its children as "InputBox", "googlesearch" button, and "Imfeelinglucky" button. In this way, the dependencies of the UI elements inside a UI module on external UI elements are reduced, making the UI definition more robust.

Respond Attribute: JavaScript Events

Tellurium provides a "respond" attribute used to define any event requiring the UI object to respond.

Most web applications include Javascript, and thus the web testing framework must be able to handle Javascript events. What is important is firing the appropriate events to trigger the event handlers.

Selenium has already provided methods to generate events such as:

```
fireEvent(locator, "blur")
fireEvent(locator, "focus")
mouseOut(locator)
mouseOver(locator)
```

Tellurium was born with Javascript events in mind since it was initially designed to test applications written using the DOJO JavaScript framework.

For example, we have the following radio button:

```
<input type='radio' name='mas_address_key' value='5779'
onClick='SetAddress_5779()'>
```

Alternately one can define the radio button as follows:

```
RadioButton(uid: "billing", clocator: [name: 'mas_address_key',
value: '5779'])
```

The above code does not respond to the Click event as the Tellurium RadioButton only supports the "check" and "uncheck" actions. This is enough for the normal case. As a result, no "click" event/action is generated during testing.

To address this problem, Tellurium added the "respond" attribute to Tellurium UI objects. The "respond" attribute is used to define any event requiring the UI object to respond. The Radio Button is redefined as:

```
 ui.Container(uid: "form", clocator: [whatever]){
     RadioButton(uid: "billing", clocator: [name: 'mas_address_key',
value: '5779'],
                 respond: ["click"])
 }
```

Then issue the following command:

```
  click "form.billing"
```

Even if the RadioButton? does not have the click method defined by default, it is still able to dynamically add the click method at runtime and call it.

A more general example is:

```
InputBox(uid: "searchbox", clocator: [title: "Google Search"],
          respond: ["click", "focus", "mouseOver", "mouseOut",
"blur"])
```

Except for the "click" event, all of the "focus", "mouseOver", "mouseOut", and "blur" events are automatically fired by Tellurium during testing. Do not worry about the event order for the respond attribute as Tellurium automatically re-orders the events and then processes them appropriately.

jQuery Selector (Improving Test Speed)

Use the jQuery Selector to improve the test speed in IE as IE lacks native XPath support and the XPath is slow. Tellurium exploits jQuery selector capability to improve test speed dramatically.

Tellurium supports both XPath and jQuery selector and still uses XPath as the default locator. To use jQuery selector, explicitly command Tellurium to use the jQuery selector.

To **enable** jQuery selector call the following method:

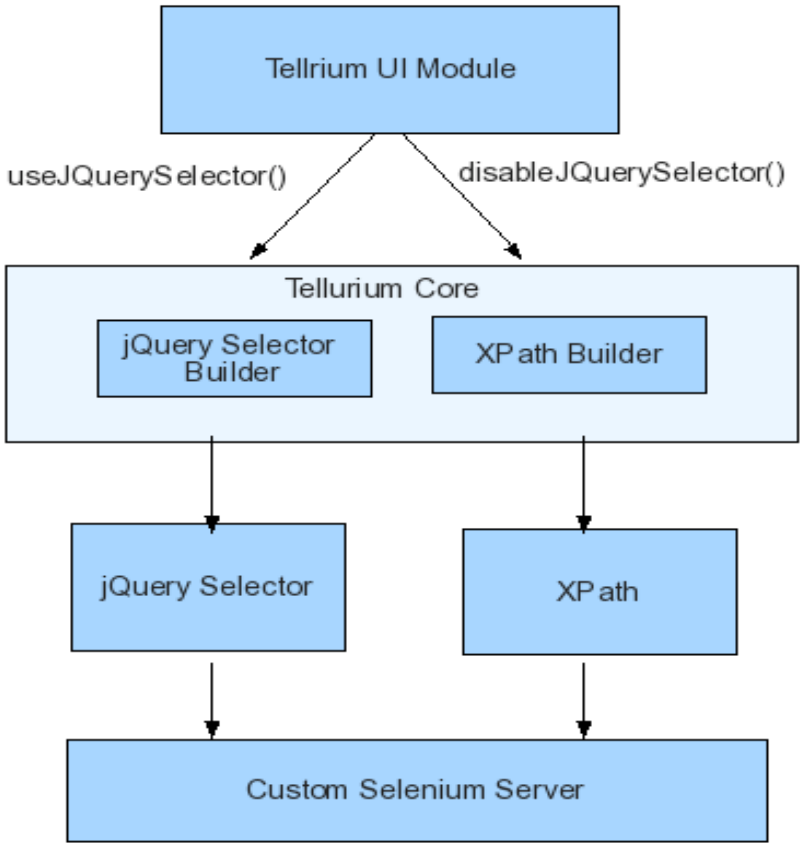
`useJQuerySelector():` to use jQuery selector.

To **disable** jQuery selector call the following method:

`disableJQuerySelector():` to switch back to XPath locator.

jQuery also switches back to the default locator. For example. XPath. jQuery is illustrated in the diagram as shown in Figure 2-4:

Figure 2-4: jQuery Diagram



Some of the jQuery functions are provided by the custom Selenium server, which is created by the Tellurium Engine project. Be aware that jQuery selector only works for the Composite locator, the `clocator`. *It is not for use with the base locator, such as the XPath, in the UI module*.

How Does the jQuery Selector Work?

jQuery Selector is used to customize Selenium Core to load the jQuery library at startup time. Add `jquery.js` into the `TestRunner.html` and `RemoteRunner.html`.

Another way is dump all `jquery.js` into `user-extensions.js`. As the Tellurium Engine prototype customizes Selenium core anyway, the former method is used.

After that, register a custom Locate strategy "jquery", using the following Selenium API call:

```
addLocationStrategy ( strategyName,functionDefinition )
```

This defines a new function for Selenium to locate elements on the page.

For example, if you define the strategy "foo", and someone runs click("foo=blah"), Tellurium runs the function, passing the string "blah". Click on the element that the function returns, or throw an "Element not found" error if the function returns null.

Selenium passes three arguments to the location strategy function:

a. **locator**: String the user passed in
b. **inWindow**: Currently selected window
c. **inDocument**: Currently selected document

The function must return null if the element cannot be found.

Arguments:

strategyName: the name of the strategy to define. This uses only letters a-zA-Z with no spaces or other punctuation.

functionDefinition: a string defining the body of a function in JavaScript. For example:

```
return inDocument.getElementById(locator);
```

Accordingly, Tellurium defines the custom locate strategy as follows:

```
addLocationStrategy("jquery", '''
  var found = $(inDocument).find(locator);
  if(found.length == 1 ){
    return found[0];
  }else if(found.length > 1){
    return found.get();
  }else{
    return null;
  }
''')
```

The code is pretty straightforward. When one element is found, return its DOM reference.

Note: Selenium does not accept returning an array with only one element.

If multiple elements are found, Tellurium uses the `jQuery get()` method to return an array of DOM references. Otherwise, return null.

Custom Attribute Locator

The actual code is a bit more complicated as the custom attribute locator must be considered. Tellurium uses the same format of attribute locator as the XPath.

For example:

```
locator@attribute
```

Procedures:

1. Create a set of custom Selenium methods
2. Add the appropriate Selenium RC Java methods.

   For example, the following Selenium method was created:

```
Selenium.prototype.getAllText = function(locator){
  var $e = $(this.browserbot.findElement(locator));
  var out = [];
  $e.each(function(){
     out.push($(this).text());
  });

  return JSON.stringify(out);
};
```

3. Create the corresponding Java method by extending Selenium

```
class CustomSelenium extends DefaultSelenium {

  def String getAllText(String locator){
     String[] arr = [locator];
     String st = commandProcessor.doCommand("getAllText", arr);
     return st;
  }
}
```

New DSL Methods

jQuery Selector provides the following additional Selenium methods, utilized in `DslContext` to form a set of new DSL methods:

`String getAllText(String locator):`

1. Get all the text from the set of elements corresponding to the jQuery Selector.

`String getCSS(String locator, String cssName) :`

2. Get the CSS properties for the set of elements corresponding to the jQuery Selector.

`Number getJQuerySelectorCount(String locator) :`

3. Get the number of elements matching the corresponding jQuery Selector.

Additional jQuery Attribute Selectors

jQuery also supports the following attribute selectors:

`attribute:` have the specified attribute.

`attribute=value:` have the specified attribute with a certain value.

`attribute!=value:` either do not have the specified attribute or have the specified attribute but not with a certain value.

`attribute^=value:` have the specified attribute and it starts with a certain value.

`attribute$=value:` have the specified attribute and it ends with a certain value.

`attribute*=value:` have the specified attribute and it contains a certain value.

Locator Agnostic Methods

Apart from the above, Tellurium provides a set of locator agnostic methods.

For example, the method automatically decides to use XPath or jQuery dependent on the `exploreJQuerySelector` flag, which can be turned on and off by the following two methods:

1. public void `useJQuerySelector()`

2. public void `disableJQuerySelector()`

Tellurium also provides the corresponding XPath specific and jQuery selector specific methods for your convenience. However, it is recommended that you use the locator agnostic methods until there is a good reason not to.

The new XPath and jQuery Selector specific methods are as follows:

1. Get the Generated locator from the UI module

   Locator agnostic:

   ```
   String getLocator(String uid)
   ```

   JQuery selector specific:

   ```
   String getSelector(String uid)
   ```

   XPath specific:

   ```
   String getXPath(String uid)
   ```

2. Get the Number of Elements matching the Locator

   Locator agnostic:

   ```
   Number getLocatorCount(String locator)
   ```

   JQuery selector specific:

   ```
   Number getJQuerySelectorCount(String jQuerySelector)
   ```

   XPath specific:

   ```
   Number getXpathCount(String xpath)
   ```

3. Table Methods

   Locator agnostic:

   ```
   int getTableHeaderColumnNum(String uid)
   int getTableFootColumnNum(String uid)
   int getTableMaxRowNum(String uid)
   int getTableMaxColumnNum(String uid)
   int getTableMaxRowNumForTbody(String uid, int ntbody)
   int getTableMaxColumnNumForTbody(String uid, int ntbody)
   int getTableMaxTbodyNum(String uid)
   ```

   JQuery selector specific:

   ```
   int getTableHeaderColumnNumBySelector(String uid)
   int getTableFootColumnNumBySelector(String uid)
   int getTableMaxRowNumBySelector(String uid)
   int getTableMaxColumnNumBySelector(String uid)
   int getTableMaxRowNumForTbodyBySelector(String uid, int
   ntbody)
   ```

```
int getTableMaxColumnNumForTbodyBySelector(String uid, int
ntbody)
int getTableMaxTbodyNumBySelector(String uid)
```

XPath specific:

```
int getTableHeaderColumnNumByXPath(String uid)
int getTableFootColumnNumByXPath(String uid)
int getTableMaxRowNumByXPath(String uid)
int getTableMaxColumnNumByXPath(String uid)
int getTableMaxRowNumForTbodyByXPath(String uid, int
ntbody)
int getTableMaxColumnNumForTbodyByXPath(String uid, int
ntbody)
int getTableMaxTbodyNumByXPath(String uid)
```

4. Verify if an Element is Disabled

   Locator agnostic:

   ```
   boolean isDisabled(String uid)
   ```

   JQuery selector specific:

   ```
   boolean isDisabledBySelector(String uid)
   ```

   XPath specific:

   ```
   boolean isDisabledByXPath(String uid)
   ```

5. Get the Attribute

   Locator agnostic:

   ```
   def getAttribute(String uid, String attribute)
   ```

6. Check the CSS Class

   Locator agnostic

   ```
   def hasCssClass(String uid, String cssClass)
   ```

7. Get CSS Properties

   JQuery selector specific:

   ```
   String[] getCSS(String uid, String cssName)
   ```

8. Get All Data from a Table

   JQuery selector specific:

```
String[] getAllTableCellText(String uid)
String[] getAllTableCellTextForTbody(String uid, int index)
```

9. Get List Size

   Locator agnostic:

   ```
   int getListSize(String uid)
   ```

   JQuery selector specific:

   ```
   getListSizeBySelector(String uid)
   ```

   XPath specific:

   ```
   getListSizeByXPath(String uid)
   ```

There are issues to be aware of with jQuery Selector:

- If you have a duplicate "id" attribute on the page, jQuery selector always returns the first DOM reference, ignoring other DOM references with the same "id" attribute.
- Some attributes may not be working in jQuery. For example, the "action" attribute in a form. Tellurium has a black list to automatically filter out the attributes that are not honored by jQuery selector.
- The "src" attribute in Image has to be a full URL such as http://www.google.com. One workaround is to put '*' before the URL.

jQuery Selector Cache Option

jQuery cache is a mechanism used to further improve the testing speed by reusing the found DOM reference for a given jQuery selector. Tellurium benchmark results show that the jQuery cache can improve the speed of testing by up to 14% over the regular jQuery selector and over 27% for some extreme cases.

But the improvement of jQuery cache over regular jQuery selector has upper bounds, which is that portion of jQuery locating time out of the round trip time from Tellurium core to Selenium core.

Caution: jQuery cache is considered to be experimental at this current stage. Use it at your own discretion.

Make the jQuery Selector Cache Option Configurable

To make the cache option configurable in the Tellurium UI Module, Tellurium introduces a cacheable attribute to the UI object. This is set to be true by default.

For a Container type object, there is an additional attribute: **noCacheForChildren** to control whether or not to cache its children.

One example of a UI module using jQuery Selector Cache Option is defined as follows:

```
  ui.Table(uid: "issueResultWithCache", cacheable: "true",
noCacheForChildren: "true",
           clocator: [id: "resultstable", class: "results"], group:
"true") {

        ......

       //define table elements
       //for the border column
       TextBox(uid: "row: *, column: 1", clocator: [:])
       TextBox(uid: "row: *, column: 8", clocator: [:])
       TextBox(uid: "row: *, column: 10", clocator: [:])
       //For the rest, just UrlLink
       UrlLink(uid: "all", clocator: [:])
    }
```

Where the cacheable can overwrite the default value found in the UI object and **noCacheForChildren**, in the above example, Tellurium is forced into not cacheing its children.

When the jQuery Cache mechanism is chosen, the Tellurium core passes both the jQuery selector and a meta command to the Tellurium Engine, embedded inside the Selenium core at this stage.

The format taking the Tellurium issue searching UI is shown in the following example:

```
jquerycache={
  "locator":"form[method=get]:has(#can, span:contains(for),
input[type=text][name=q],
              input[value=Search][type=submit]) #can",
  "optimized":"#can",
  "uid":"issueSearch.issueType",
  "cacheable":true,
  "unique":true
}
```

Where the locator is the regular jQuery selector and is optimized by the jQuery selector optimizer in the Tellurium core. The locator is used for the child UI element to derive a partial jQuery selector from its parent. The optimized jQuery selector is used for actual DOM searching.

The other three parameters are Meta commands. `uid` is the UID for the corresponding jQuery selector. The cacheable tells the Engine whether to cache the selector or not. The reason is that some UI elements on the web are dynamic; for instance, the UI element is inside a data grid. As a result, it may not be useful to cache the jQuery selector in this instance.

The last unique method tells the Engine whether the jQuery selector expects multiple elements or not. This is very useful to handle in a case where jQuery selector is expected to return one element, but actually returns multiple ones. In such a case, a Selenium Error is thrown.

On the Engine side, the cache is defined as:

```
//global flag to decide whether to cache jQuery selectors
this.cacheSelector = false;

//cache for jQuery selectors
this.sCache = new Hashtable();

this.maxCacheSize = 50;

this.cachePolicy = discardOldCachePolicy;
```

The cache system includes a global flag used to decide whether to use the cache capability, a hash table to store the cached data, a cache size limit, and a cache eviction policy once the cache is filled up.

jQuery Selector Cache Structure

The cached data structure is defined as:

```
//Cached Data, use uid as the key to reference it
function CacheData(){
    //jQuery selector associated with the DOM reference,
which is a whole selector
    //without optimization so that it is easier to find the
reminding selector
    //for its children
    this.selector = null;
    //optimized selector for actual DOM search
    this.optimized = null;
    //jQuery object for DOM reference
    this.reference = null;
    //number of reuse
    this.count = 0;
    //last use time
```

```
        this.timestamp = Number(new Date());
    };
```

jQuery Selecor Cache Eviction Policies

The Tellurium Engine provides the following jQuery Selector cache eviction policies:

1. **`DiscardNewPolicy:`** Discard new jQuery selector.
2. **`DiscardOldPolicy:`** Discard the oldest jQuery selector measured by the last update time.
3. **`DiscardLeastUsedPolicy:`** Discard the least used jQuery selector.
4. **`DiscardInvalidPolicy:`** Discard the invalid jQuery selector first.

It is important to know when the cached data become invalid. There are three mechanisms to utilize:

1. Listen for page refresh event and invalidate the cache data accordingly
2. Intercept Ajax response to decide when the web is an update and if the cache is to be updated.
3. Validate the cached jQuery selector before using it.

Currently the Tellurium Engine uses the 3rd mechanism to verify if the cached data is valid or not. The first two mechanisms are still under development.

Whenever the Tellurium Core passes a locator to the Engine, the Engine first looks at the meta command cacheable. If this flag is true, it first tries to look up the DOM reference from the cache. If no cached DOM reference is available, implement a fresh jQuery search and cache the DOM reference.

Otherwise, validate the cached DOM reference and use it directly. If the cacheable flag is false, the Engine looks for the ancestor of the UI element by its UID and initiates a jQuery search starting from its ancestor, if possible.

jQuery Selector Cache Control

Tellurium Core provides the following methods for jQuery Selector cache control:

`Methods`

UI Templates

Tellurium UI templates have two purposes:

1. When there are many identical UI elements, use one template to represent them all
2. When there are variable/dynamic sizes of UI elements at runtime, the patterns are known, but not the size.

More specifically, Table and List are two Tellurium objects that define UI templates.

1. **Table** defines two dimensional UI templates
2. **List** defines one dimensional UI templates

The Template has special UIDs such as "2", "all", or "row: 1, column: 2".

Looking at use case (1), the HTML source is:

```
<ul class="a">
  <li>
      <A HREF="site?tcid=a"
         class="b">AA
      </A>
  </li>
  <li>
      <A HREF="site?tcid=b"
         class="b">BB
      </A>
  </li>
  <li>
      <A HREF="site?;tcid=c"
         class="b">CC
      </A>
  </li>
  <li>
      <A HREF="site?tcid=d"
         class="b">DD
      </A>
  </li>
  <li>
      <A HREF="site?tcid=e"
         class="b">EE
      </A>
  </li>
  <li>
      <A HREF="site?tcid=f"
         class="b">FF
      </A>
```

```
            </li>
        </ul>
```

In this example there are six links. Without templates, one would put six UrlLink objects in the UI module. By using the templates, the work is easier and simplified.

```
ui.List(uid: "list", clocator: [tag: "ul", class: "a"],
separator:"li")
{
    UrlLink(uid: "all", clocator: [class: "b"])
}
```

For use case (2), a common application is the data grid. Look at the "`issueResult`" data grid on the Tellurium Issues page for an easy and simplified result as shown below:

```
ui.Table(uid: "issueResult", clocator: [id: "resultstable", class:
"results"],
         group: "true")
{
    TextBox(uid: "header: 1",  clocator: [:])
    UrlLink(uid: "header: 2",  clocator: [text: "%%ID"])
    UrlLink(uid: "header: 3",  clocator: [text: "%%Type"])
    UrlLink(uid: "header: 4",  clocator: [text: "%%Status"])
    UrlLink(uid: "header: 5",  clocator: [text: "%%Priority"])
    UrlLink(uid: "header: 6",  clocator: [text: "%%Milestone"])
    UrlLink(uid: "header: 7",  clocator: [text: "%%Owner"])
    UrlLink(uid: "header: 9",  clocator: [text: "%%Summary +
Labels"])
    UrlLink(uid: "header: 10", clocator: [text: "%%..."])

    //define table elements
    //for the border column
    TextBox(uid: "row: *, column: 1", clocator: [:])
    //For the rest, just UrlLink
    UrlLink(uid: "all", clocator: [:])
}
```

The resulting definitions shown are very simple, time-saving and easy to use.

If the user has multiple templates such as the "`issueResult`" shown in the table above, the rule generally applied to the templates is: "**specific one first, general one later**".

"Include" Frequently Used Sets of Elements in UI Modules

When there is a frequently used set of elements, re-defining them repeatedly in your UI module is not necessary. Simply use the Tellurium "Include" syntax to re-use the pre-defined UI elements.

```
Include(uid: UID, ref: REFERRED_UID)
```

Use "ref" to reference the object to be included, then specify the UID for the object.

Note: If a different UID is required, there is no need to specify it.

if the Object UID is not the same as the original one, Tellurium clones a new object for users so that multiple objects with different UIDs are available.

For example, first define the following reused UI module:

```
   ui.Container(uid: "SearchModule", clocator: [tag: "td"], group:
"true") {
     InputBox(uid: "Input", clocator: [title: "Google Search"])
     SubmitButton(uid: "Search", clocator: [name: "btnG", value:
"Google Search"])
     SubmitButton(uid: "ImFeelingLucky", clocator: [value: "I'm
Feeling Lucky"])
   }
```

Then, include it into the new UI module as follows:

```
   ui.Container(uid: "Google", clocator: [tag: "table"]) {
     Include(ref: "SearchModule")
     Include(uid: "MySearchModule", ref: "SearchModule")
     Container(uid: "Options", clocator: [tag: "td", position: "3"],
group: "true") {
       UrlLink(uid: "LanguageTools", clocator: [tag: "a", text:
"Language Tools"])
       UrlLink(uid: "SearchPreferences", clocator: [tag: "a", text:
"Search Preferences"])
       UrlLink(uid: "AdvancedSearch", clocator: [tag: "a", text:
"Advanced Search"])
     }
   }
```

Javascript Events

Most web applications include Javascript, and thus the web testing framework must be able to handle Javascript events. What is important is firing the appropriate events to trigger the event handlers.

Selenium has already provided methods to generate events such as:

```
fireEvent(locator, "blur")
fireEvent(locator, "focus")
mouseOut(locator)
mouseOver(locator)
```

Tellurium was born with Javascript events in mind since it was initially designed to test applications written using the DOJO JavaScript? framework.

For example, the user has the following radio button:

```
<input type='radio' name='mas_address_key' value='5779'
onClick='SetAddress_5779()'>
```

Alternately one can define the radio button as follows:

```
RadioButton(uid: "billing", clocator: [name: 'mas_address_key',
value: '5779'])
```

The above code does not respond to the Click event as the Tellurium RadioButton? only supports the "check" and "uncheck" actions. This is enough for the normal case. As a result, no "click" event/action is generated during testing.

To address this problem, Tellurium added the "respond" attribute to Tellurium UI objects. The "respond" attribute is used to define any events requiring the UI object to respond. The Radio Button can be redefined as:

```
 ui.Container(uid: "form", clocator: [whatever]){
     RadioButton(uid: "billing", clocator: [name: 'mas_address_key',
value: '5779'],
                 respond: ["click"])
 }
```

Then issue the following command:

```
  click "form.billing"
```

Even if the RadioButton does not have the click method defined by default, it is still able to dynamically add the click method at runtime and call it.

A more general example is:

```
 InputBox(uid: "searchbox", clocator: [title: "Google Search"],
          respond: ["click", "focus", "mouseOver", "mouseOut",
"blur"])
```

Except for the "click" event, all of the "focus", "mouseOver", "mouseOut", and "blur" events are automatically fired by Tellurium during testing. Do not worry about the event order for the respond attribute as Tellurium automatically re-orders the events and then processes them appropriately.

Logical Container

The Container object in Tellurium is used to hold child objects that are in the same subtree in the DOM object. However, there are always exceptions. For example, the Logical Container (or Virtual Container - Logical Container is preferred) can violate this rule.

What is a Logic Container? It is a Container with an empty locator. For instance:

```
Container(uid: "logical"){
   ......
}
```

But empty `!=` nothing. There are some scenarios where the Logical Container can play an important role. The Container includes an `uid` for a reference and it logically groups the UI element together.

For example, in the following example the UI elements under the Tag li are different:

```
      <div class="block_content">
          <ul>
              <li>
                  <h5>
                      <a href="" title="">xxx</a>
                  </h5>
                  <p class="product_desc">
                      <a href=".." title="More">...</a>
                  </p>
                  <a href="..." title=".." class="product_image">
                      <img src="..." alt="..." height="129"
width="129"/>
                  </a>
                  <p>
                      <a class="button" href="..."
```

```
title="View">View</a>
                    <a title="Add to cart">Add to cart</a>
                </p>
            </li>
            <li>
                similar UI
            </li>
            <li>
                similar UI
            </li>
        </ul>
    </div>
```

The issue is how to write the UI template for them. The logical Container then comes into play. For example, the UI module is written as follows:

```
ui.Container(uid: "content", clocator: [tag: "div", class:
"block_content"]){
     List(uid: "list", clocator: [tag: "ul"], separator:"li") {
         Container("all"){
             UrlLink(uid: "title", clocator: [title: "View"])
                     ......
                     other elements inside the li tag
         }
     }
}
```

Another usage of the logical Container is to convert the test case recorded by Selenium IDE to Tellurium test cases. For example, using the search UI on the Tellurium download page, first record the following Selenium test case using Selenium IDE:

```
import com.thoughtworks.selenium.SeleneseTestCase;

public class SeleniumTestCase extends SeleneseTestCase {
        public void setUp() throws Exception {
                setUp("http://code.google.com/", "*chrome");
        }
        public void testNew() throws Exception {
                selenium.open("/p/aost/downloads/list");
                selenium.select("can", "label=regexp:\\sAll
Downloads");
                selenium.type("q", "TrUMP");
                selenium.click("//input[@value='Search']");
```

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

Do not be confused by the locator "can" and "q", as they are UI element IDs and are easily expressed in XPath. The "`label=regexp:\\sAll Downloads`" part shows that Selenium uses regular express to match the String and the "\s" stands for a space. As a result, write the UI module based on the above code.

```
public class TelluriumDownloadPage extends DslContext {

  public void defineUi() {
    ui.Container(uid: "downloads") {
      Selector(uid: "downloadType", locator: "//*[@id='can']")
      InputBox(uid: "input", locator: "//*[@id='q']")
      SubmitButton(uid: "search", locator:
"//input[@value='Search']")
    }
  }

  public void searchDownload(String downloadType, String
searchKeyWords) {
    selectByLabel "downloads.downloadType", downloadType
    keyType "downloads.input", searchKeyWords
    click "downloads.search"
    waitForPageToLoad 30000
  }
}
```

The Tellurium test case is created accordingly:

```
public class TelluriumDownloadPageTestCase extends
TelluriumJavaTestCase {

    protected static TelluriumDownloadPage ngsp;

    @BeforeClass
    public static void initUi() {
        ngsp = new TelluriumDownloadPage();
        ngsp.defineUi();
    }

    @Test
```

```
    public void testSearchDownload(){
        connectUrl("http://code.google.com/p/aost/downloads/list");
        ngsp.searchDownload(" All Downloads", "TrUMP");
    }
}
```

Customize Individual Test Settings Using `setCustomConfig`

`TelluriumConfig.groovy` provides project level test settings. Use `setCustomConfig` to customize individual test settings.

```
public void setCustomConfig(boolean runInternally, int port, String
browser,
                            boolean useMultiWindows, String
profileLocation)

public void setCustomConfig(boolean runInternally, int port, String
browser,
                  boolean useMultiWindows, String profileLocation,
String serverHost)
```

For example, specify custom settings as follows:

```
public class GoogleStartPageJavaTestCase extends
TelluriumJavaTestCase {
    static{
        setCustomConfig(true, 5555, "*chrome", true, null);
    }

    protected static NewGoogleStartPage ngsp;

    @BeforeClass
    public static void initUi() {
        ngsp = new NewGoogleStartPage();
        ngsp.defineUi();
        ngsp.useJavascriptXPathLibrary();
    }

    @Test
    public void testGoogleSearch() {
        connectUrl("http://www.google.com");
```

```
            ngsp.doGoogleSearch("tellurium selenium Groovy Test");
        }

        ......

    }
```

User Custom Selenium Extensions

To support user custom selenium extensions, Tellurium Core adds the following configurations to **`TelluriumConfig.groovy`**.

```
    embeddedserver {

      ......

      //user-extension.js file, for example "target/user-extensions.js"
      userExtension = ""
    }
    connector{

      ......

      //user's class to hold custom selenium methods associated with
user-extensions.js
      //should in full class name, for instance,
"com.mycom.CustomSelenium"
      customClass = ""
    }
```

Where the userExtension points to the user's user-extensions.js file. For example, use the following user-extensions.js:

```
Selenium.prototype.doTypeRepeated = function(locator, text) {
    // All locator-strategies are automatically handled by
"findElement"
    var element = this.page().findElement(locator);

    // Create the text to type
    var valueToType = text + text;
```

```
    // Replace the element text with the new text
    this.page().replaceText(element, valueToType);
};
```

The customClass is the user's class for custom Selenium methods, extending Tellurium org.tellurium.connector.CustomCommand class:

```
public class MyCommand extends CustomCommand{

    public void typeRepeated(String locator, String text){
                String[] arr = [locator, text];
                commandProcessor.doCommand("typeRepeated", arr);
        }
    }
}
```

Tellurium core automatically loads up user-extensions.js and custom commands. The n, user uses:

```
customUiCall(String uid, String method, Object[] args)
```

to call the custom methods. For instance:

```
customUiCall "Google.Input", typeRepeated, "Tellurium Groovy"
```

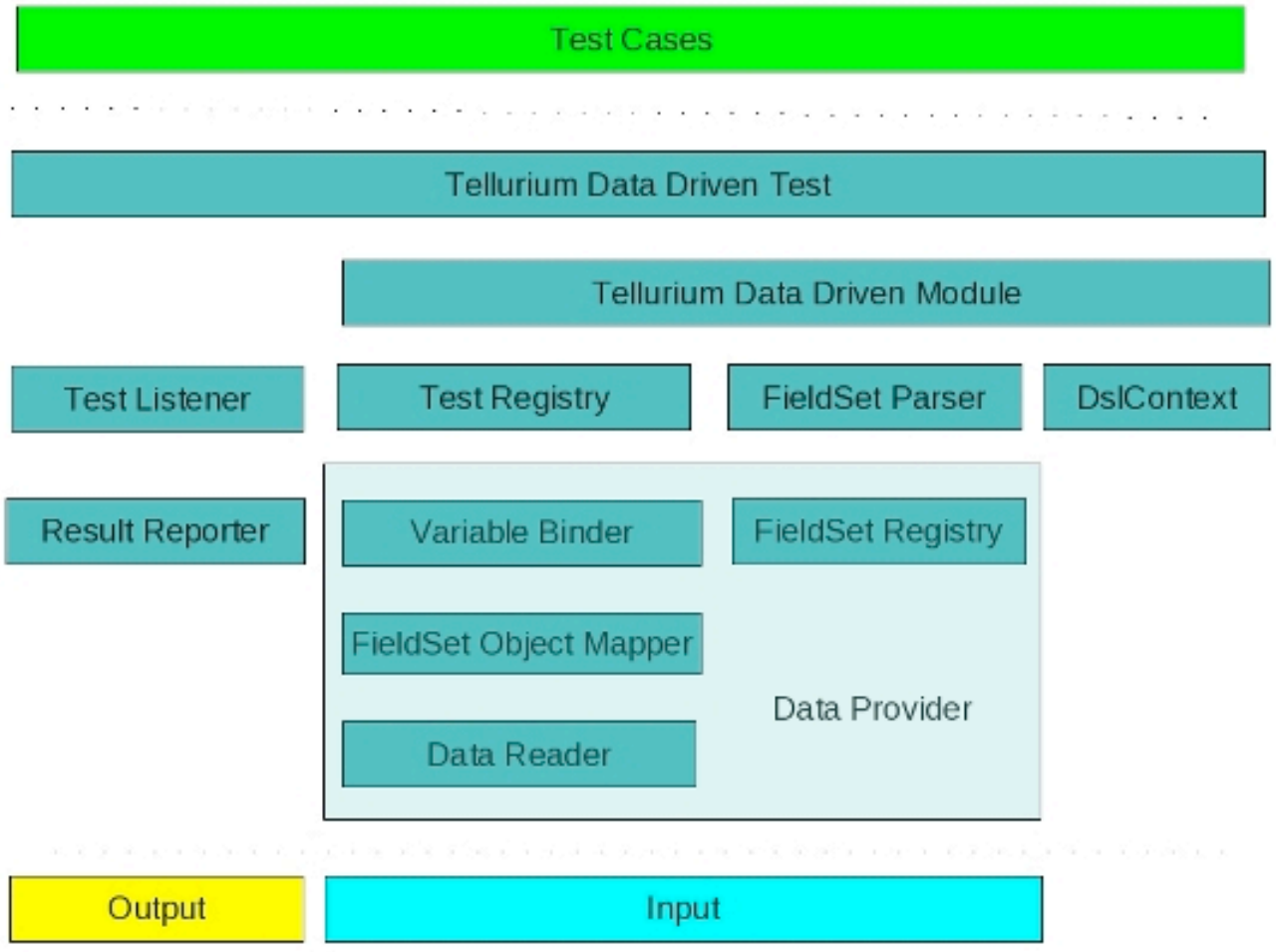
The customUiCall method handles all the UI to locator mapping for users. Tellurium also provides the following method for users to make direct calls to the Selenium server.

```
customDirectCall(String method, Object[] args)
```

Data Driven Testing

Data Driven Testing is a different way to write tests. For example, separate test data from the test scripts and the test flow is not controlled by the test scripts, but by the input file instead. In the input file, users can specify which tests to run, what are input parameters, and what are expected results. Data driven testing in Tellurium is illustrated in Figure 2-5 with the following system diagram:

Figure 2-5 Data Driven Testing in Tellurium System Diagram



The Tellurium Data Driven Test consists of three main parts:

1. Data Provider
2. TelluriumDataDrivenModule
3. TelluriumDataDrivenTest

Data Provider

The Data Provider is responsible for reading data from input stream and converting data to Java variables.

Tellurium includes the following Data Provider methods:

1. loadData file_name, load input data from a file
2. useData String_name, load input data from a String in the test script
3. bind(field_name), bind a variable to a field in a field set
4. closeData, close the input data stream and report the test results
5. cacheVariable(name, variable), put variable into cache
6. getCachedVariable(name, variable), get variable from cache where the file_name includes the file path. For example:

```
loadData "src/test/example/test/ddt/GoogleBookListCodeHostInput.txt"
```

Tellurium supports pipe format and CSV format input file.

loadData

To change the file reader for different formats, change the following settings in the configuration file **TelluriumConfig.groovy:**

```
datadriven{
    dataprovider{
        //specify which data reader you like the data provider to use
        //the valid options include "PipeFileReader", "CSVFileReader"
at this point
        reader = "PipeFileReader"
    }
}
```

useData

Tellurium's useData is designed to specify test data in the test scripts directly. It loads input from a String. The String is usually defined in Groovy style using triple quota, for example:

```
protected String data = """
    google_search | true | 865-692-6000 | tellurium
    google_search | false| 865-123-4444 | tellurium selenium test
    google_search | true | 755-452-4444 | tellurium groovy
    google_search | false| 666-784-1233 | tellurium user group
    google_search | true | 865-123-5555 | tellurium data driven
"""
    ...

  useData data
```

bind

`bind` is the command used to bind a variable to an input Field Set field at runtime. `FieldSet` is the format of a line of data. For example:

```
def row = bind("GCHLabel.row")
```

is used to bind the row variable to the "row" field in the FieldSet "GCHLabel". Tellurium does not explicitly differentiate input parameters from the expected results in the input data. To bind variables to the input data then use any of them as the expected results for result comparison.

cacheVariable and getCachedVariable

cacheVariable and getCachedVariable are used to pass intermediate variables among tests.

- cacheVariable is used to put a variable into a cache
- getCachedVariable is used to get back the variable

For example:

```
int headernum = getTableHeaderNum()
cacheVariable("headernum", headernum)

...
```

```
int headernum = getCachedVariable("headernum")
...
```

closeData

When testing is completed, use "closeData" to close the input data stream. In the meantime, the result reporter outputs the test results in the format specified in the configuration file.

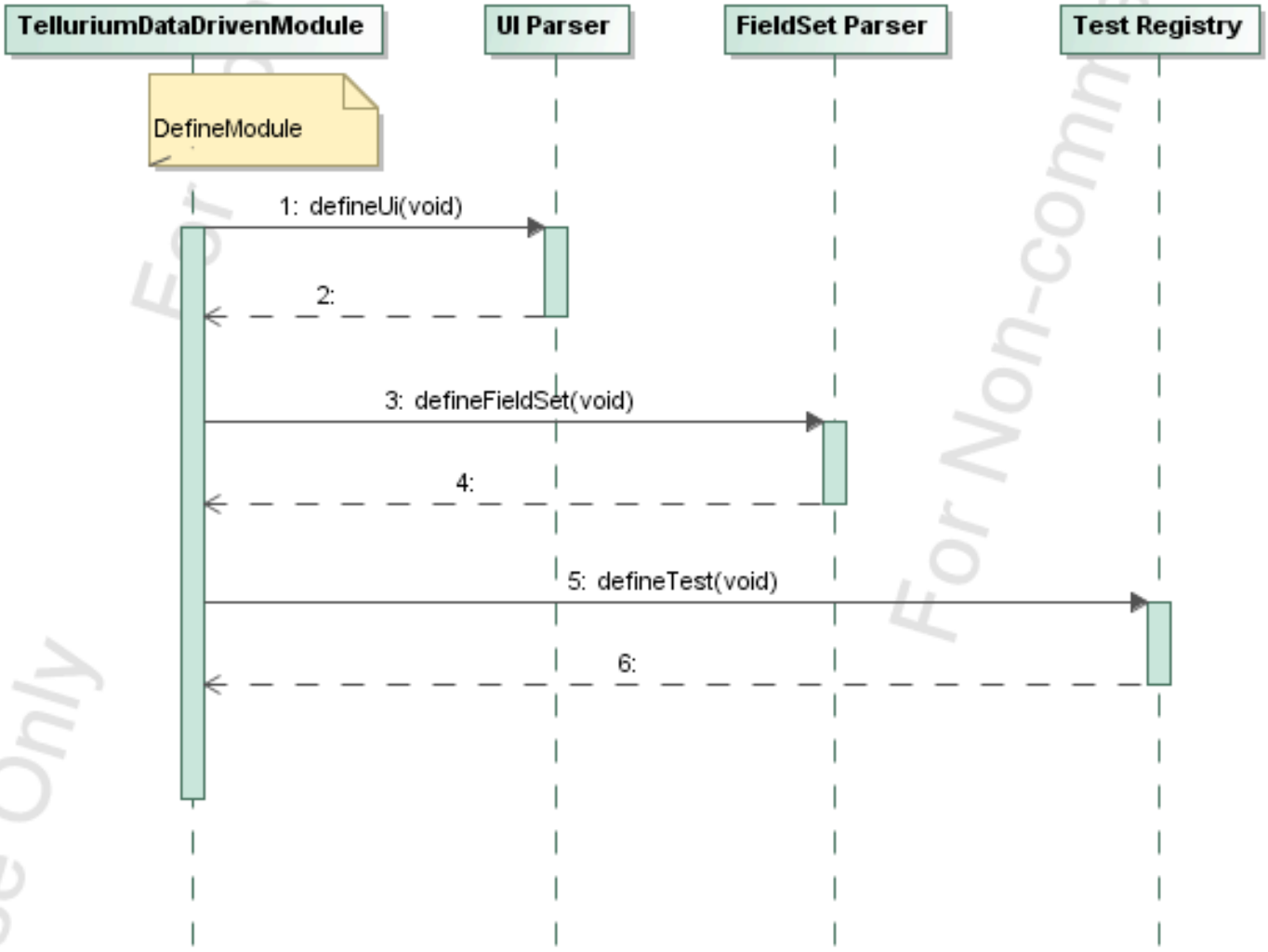
For example: the XML file as shown in the TelluriumConfig.groovy file:

```
test{
    result{
       //specify what result reporter used for the test result
       //valid options include "SimpleResultReporter",
"XMLResultReporter",
       //and "StreamXMLResultReporter"
       reporter = "XMLResultReporter"
       //the output of the result
       //valid options include "Console", "File" at this point
       //if the option is "File", you need to specify the file name,
       //other wise it will use the default
       //file name "TestResults.output"
       output = "Console"
       //test result output file name
       filename = "TestResult.output"
    }
}
```

TelluriumDataDrivenModule

`TelluriumDataDrivenModule` is used to define modules, where users can define UI Modules, FieldSets, and tests as shown in the following Figure 2-6 sequence diagram. Users should extend this class to define their own test modules.

Figure 2-6 TelluriumDataDrivenModule Sequence Diagram



TelluriumDataDrivenModule provides one method "defineModule" for users to implement. Since it extends the DslContext class, users define UI modules as in regular Tellurium UI Modules. For example:

```
ui.Table(uid: "labels_table", clocator: [:], group: "true"){
   TextBox(uid: "row: 1, column: 1", clocator: [tag: "div",
           text: "Example project labels:"])
   Table(uid: "row: 2, column: 1", clocator: [header:
"/div[@id=\"popular\"]"]){
        UrlLink(uid: "all", locator: "/a")
   }
}
```

FieldSet

FieldSet defines the format of one line of input data. FieldSet consists of fieldssuch as columns, in the input data. There is a special field "test", where users can specify what tests this line of data applies to. For example:

```
fs.FieldSet(name: "GCHStatus", description: "Google Code Hosting
input") {
    Test(value: "getGCHStatus")
    Field(name: "label")
    Field(name: "rowNum", type: "int")
    Field(name: "columNum", type: "int")
}
```

FieldSet defines the input data format for testing Google code hosting web page.

Note: The Test field must be the first column of the input data.

The default name of the test field is "test" and does not need to be specified. If the value attribute of the test field is not specified, it implies this same format. For example, FieldSet is used for different tests.

A regular field includes the following attributes:

```
class Field {
        //Field name
        private String name

        //Field type, default is String
        private String type = "String"
```

```
        //optional description of the Field
        private String description

        //If the value can be null, default is true
        private boolean nullable = true

        //optional null value if the value is null or not specified
        private String nullValue

        //If the length is not specified, it is -1
        private int length = -1

        //optional String pattern for the value
        //if specified, use it for String validation
        private String pattern
    }
```

Tellurium can automatically handle Java primitive types.

typeHandler

Another flexibility Tellurium provides is allowing users to define their own custom type handlers to deal with more complicated data types by using "typeHandler". For example:

```
//define custom data type and its type handler
typeHandler "phoneNumber",
"org.tellurium.test.PhoneNumberTypeHandler"

//define file data format
fs.FieldSet(name: "fs4googlesearch", description: "example field set
for google search"){
    Field(name: "regularSearch", type: "boolean",
          description: "whether we should use regular search or use
I'm feeling lucky")
    Field(name: "phoneNumber", type: "phoneNumber", description:
"Phone number")
    Field(name: "input", description: "input variable")
}
```

The above script defined a custom type "PhoneNumber" and the Tellurium automatically calls this type handler to convert the input data to the "PhoneNumber" Java type.

Define Test

The "defineTest" method is used to define a test in the TelluriumDataDrivenModule. For example, the following script defines the "clickGCHLabel" test:

```
defineTest("clickGCHLabel"){
    def row = bind("GCHLabel.row")
    def column = bind("GCHLabel.column")

    openUrl("http://code.google.com/hosting/")
    click  "labels_table[2][1].[${row}][${column}]"

    waitForPageToLoad 30000
}
```

Note: The bind command binds variables row, column to the fields "row" and "column" in the FieldSet "GCHLabel".

compareResult

Tellurium also provides the command "compareResult" for users to compare the actual result with the expected result. For example, the following script compares the expected label, row number, and column number with the acutal ones at runtime:

```
defineTest("getGCHStatus"){
    def expectedLabel = bind("GCHStatus.label")
    def expectedRowNum = bind("GCHStatus.rowNum")
    def expectedColumnNum = bind("GCHStatus.columNum")

    openUrl("http://code.google.com/hosting/")
    def label = getText("labels_table[1][1]")
    def rownum = getTableMaxRowNum("labels_table[2][1]")
    def columnum = getTableMaxColumnNum("labels_table[2][1]")

    compareResult(expectedLabel, label)
    compareResult(expectedRowNum, rownum)
    compareResult(expectedColumnNum, columnum)
    pause 1000
}
```

Sometimes users may require custom "compareResult" to handle more complicated situations. For example, when users compare two lists, users can override the default "compareResult" behaviour by specifying custom code in the closure:

```
compareResult(list1, list2){
    assertTrue(list1.size() == list2.size())
    for(int i=0; i<list1.size();i++){
        //put your custom comparison code here
    }
}
```

checkResult

If users want to check a variable in the test, the "checkResult" method is used coming with a closure where users define the actual assertions inside:

```
checkResult(issueTypeLabel) {
    assertTrue(issueTypeLabel != null)
}
```

Like "compareResult", "checkResult" captures all assertion errors. The test resumes even when the assertions fail. The result is reported in the output.

logMessage

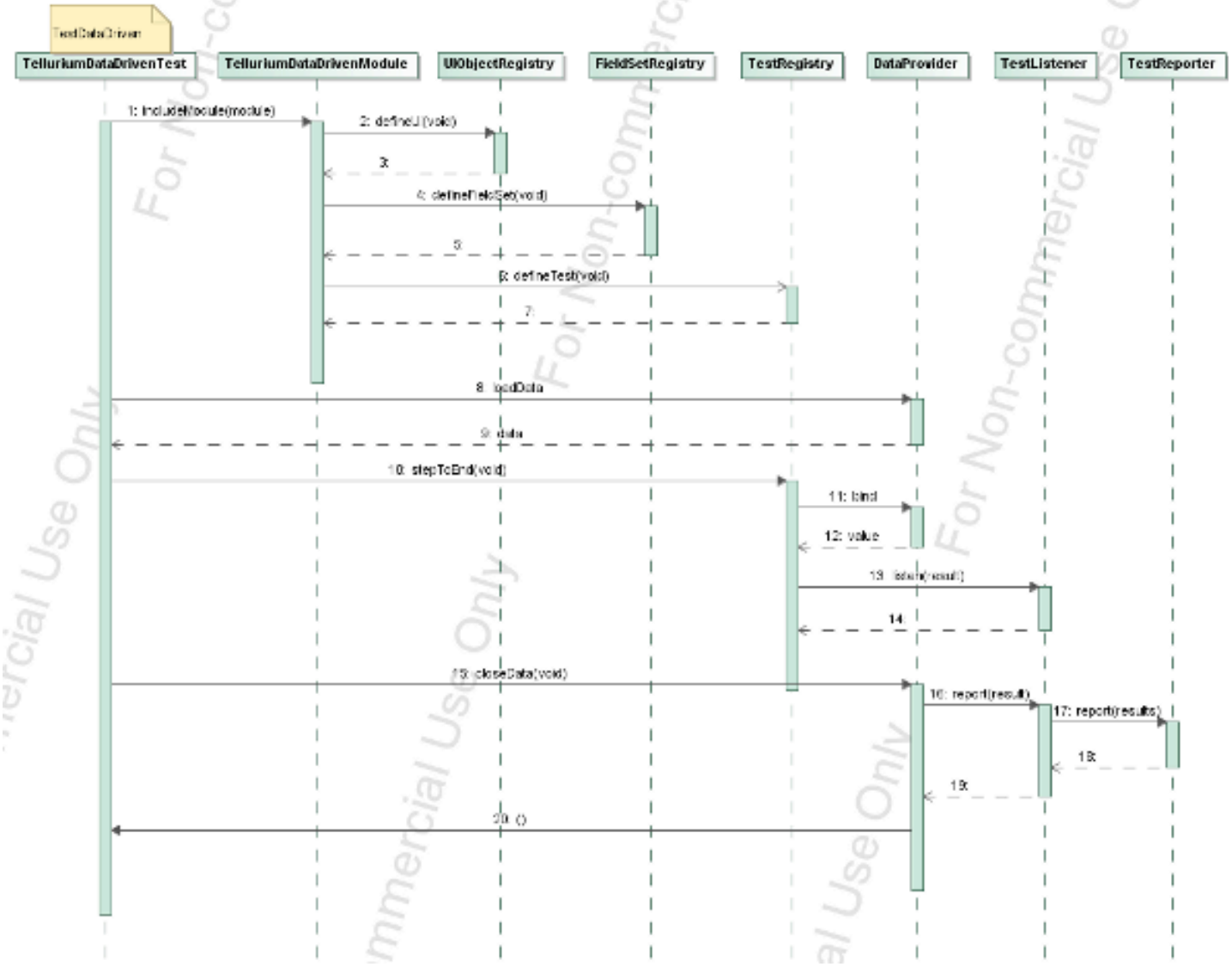
In addition, the "logMessage" is used by users to log any messages in the output.

```
logMessage "Found ${actual.size()} ${issueTypeLabel} for owner " +
issueOwner
```

TelluriumDataDrivenTest

TelluriumDataDrivenTest is the class users should extend to run the actual data driven testing. It is more like a data driven testing engine. There is only one method, "testDataDriven", which users implement. The sequence diagram for the testing process is shown in Figure 2-7:

Figure 2-7 TelluriumDataDrivenTest System Diagram



Complete the following steps to use TelluriumDataDrivenTest:

1. Use "includeModule" to load defined Modules
2. Use "loadData" or "useData" to load input data stream
3. Use "stepToEnd" to read the input data line by line and pick up the specified test and run it, until reaches the end of the data stream
4. Use "closeData" to close the data stream and output the test results

What the "includeModule" does is to merge in all Ui modules, FieldSets, and tests defined in that module file to the global registry.

"stepToEnd" looks at each input line, first find the test name and pass in all input parameters to it, and then run the test. The whole process is illustrated in the following example:

```
class GoogleBookListCodeHostTest extends
TelluriumDataDrivenTest{

    public void testDataDriven() {

        includeModule
 example.google.GoogleBookListModule.class
        includeModule
 example.google.GoogleCodeHostingModule.class

        //load file
        loadData
"src/test/example/test/ddt/GoogleBookListCodeHostInput.tx
t"

        //read each line and run the test script until
the end of the file
        stepToEnd()

        //close file
        closeData()
    }
}
```

The input data for this example are as follows:

```
##TEST should be always be the first column

##Data for test "checkBookList"
##TEST | CATEGORY | SIZE
checkBookList|Fiction|8
checkBookList|Fiction|3

##Data for test "getGCHStatus"
##TEST | LABEL | Row Number | Column Number
getGCHStatus |Example project labels:| 3 | 6
getGCHStatus |Example project| 3 | 6

##Data for test "clickGCHLabel"
##TEST | row | column
clickGCHLabel | 1 | 1
clickGCHLabel | 2 | 2
clickGCHLabel | 3 | 3
```

Note: The line starting with "##" is the comment line and the empty line is ignored.

If users want to control the testing execution flow by themselves, Tellurium also provides this capability even though its use is ***not recommended***.

Tellurium provides two additional commands, "step" and "stepOver".

- "step" is used to read one line of input data and run it.
- "stepOver" is used to skip one line of input data.

In this meanwhile, Tellurium also allows the user to specify additional test scripts using closure. For example:

```
step{
    //bind variables
    boolean regularSearch = bind("regularSearch")
    def phoneNumber = bind("fs4googlesearch.phoneNumber")
    String input = bind("input")

    openUrl "http://www.google.com"
    type "google_start_page.searchbox", input
    pause 500
```

```
        click "google_start_page.googlesearch"
        waitForPageToLoad 30000
    }
```

This usually implies that the input data format is unique or the test script knows about what format the current input data are using.

The Dump Method

Tellurium Core added a dump method to print out the UI object's and its descendants' runtime locators that Tellurium Core generates.

```
public void dump(String uid)
```

where the uid is the UI object id to be dumped. An important feature is that the dump() method does not require the user to run the actual tests. That is to say, the user does not need to run the Selenium server and launch the web browser. Simply define the UI module, then call the dump() method.

For example, to define the UI module for Tellurium Issue Search module, complete as follows:

```
public class TelluriumIssueModule extends DslContext {

  public void defineUi() {

    //define UI module of a form include issue type selector and
issue search
    ui.Form(uid: "issueSearch", clocator: [action: "list", method:
"get"], group: "true"){
      Selector(uid: "issueType", clocator: [name: "can", id: "can"])
      TextBox(uid: "searchLabel", clocator: [tag: "span", text:
"*for"])
      InputBox(uid: "searchBox", clocator: [type: "text", name: "q"])
      SubmitButton(uid: "searchButton", clocator: [value: "Search"])
    }
  }
}
```

The user can use the dump method in the following way:

```
  TelluriumIssueModule tisp = new TelluriumIssueModule();
  tisp.defineUi();
  tisp.dump("issueSearch");
```

The above code prints out the runtime XPath locators.

Dump locator information for issueSearch is as follows:

```
--------------------------------------------------------
issueSearch: //descendant-or-self::form[@action="list" and
@method="get"]
issueSearch.issueType: //descendant-or-
self::form[descendant::select[@name="can" and
     @id="can"] and descendant::span[contains(text(),"for")] and
     descendant::input[@type="text" and @name="q"] and
     descendant::input[@value="Search" and @type="submit"] and
     @action="list" and @method="get"]/descendant-or-self::select
                          [@name="can" and @id="can"]
issueSearch.searchLabel: //descendant-or-
self::form[descendant::select[@name="can"
     and @id="can"] and descendant::span[contains(text(),"for")] and
     descendant::input[@type="text" and @name="q"] and
descendant::input[
     @value="Search" and @type="submit"] and @action="list" and
@method="get"]
     /descendant-or-self::span[contains(text(),"for")]
issueSearch.searchBox: //descendant-or-
self::form[descendant::select[@name="can"
     and @id="can"] and descendant::span[contains(text(),"for")] and
     descendant::input[@type="text" and @name="q"] and
descendant::input[
     @value="Search" and @type="submit"] and @action="list" and
@method="get"]
     /descendant-or-self::input[@type="text" and @name="q"]
issueSearch.searchButton: //descendant-or-
self::form[descendant::select[@name="can"
     and @id="can"] and descendant::span[contains(text(),"for")] and
     descendant::input[@type="text" and @name="q"] and
descendant::input[
     @value="Search" and @type="submit"] and @action="list" and
@method="get"]
     /descendant-or-self::input[@value="Search" and @type="submit"]
--------------------------------------------------------
```

Selenium Grid Support

Selenium Grid transparently distributes tests on multiple machines so that the tests are run in parallel. Recently support for the Selenium Grid has been added to Tellurium. Now Tellurium tests can be run against different browsers using Selenium Grid. Tellurium core is updated to support Selenium Grid sessions.

For example, assume 3 machines are set up to run Tellurium tests on the Selenium Grid. All the steps can be completed on the user's local box. To do this locally, remove the machine names with `localhost`. Each machine in this set up has a defined role as described below:

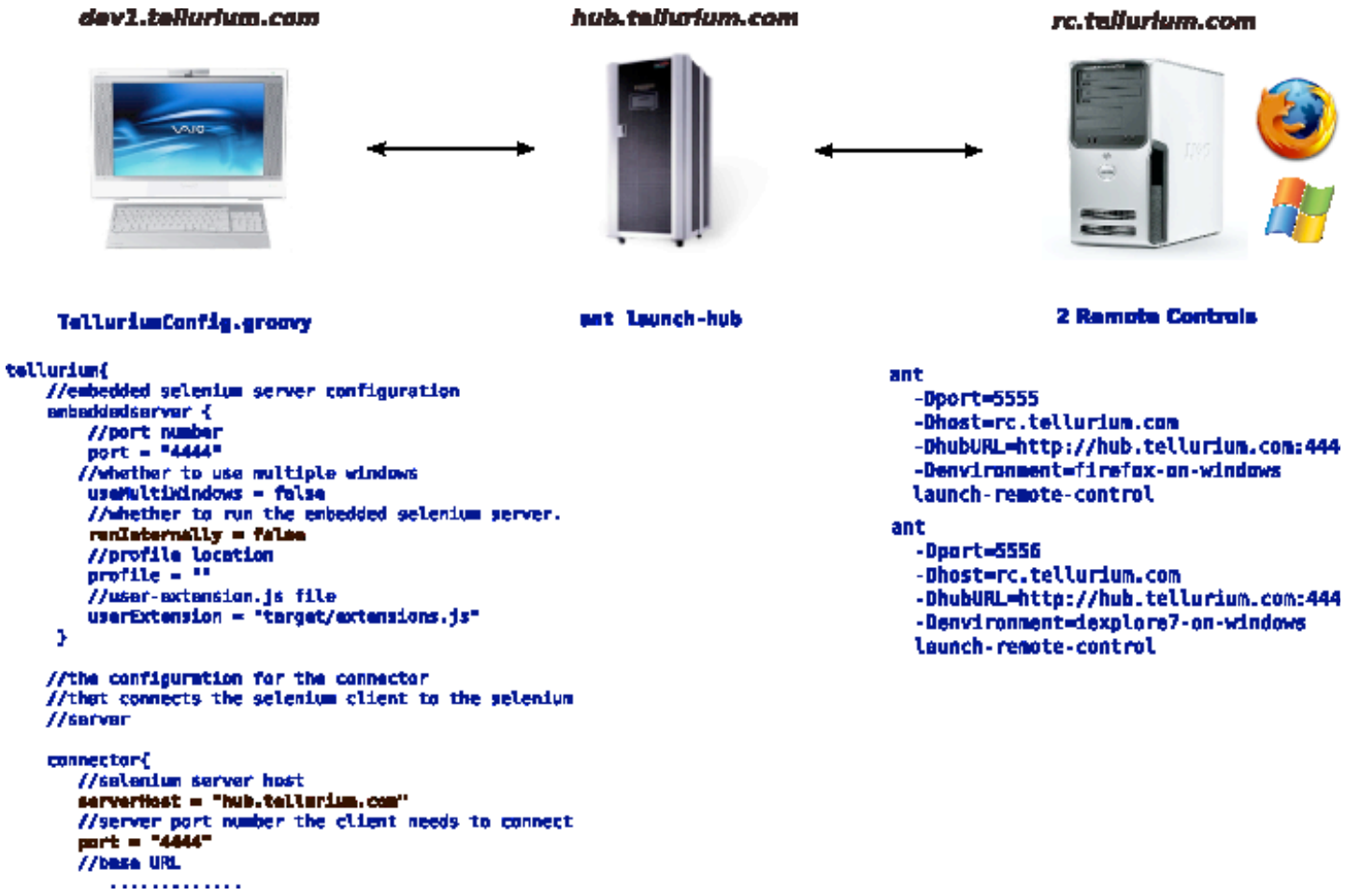
1. **dev1.tellurium.com** Tellurium test development machine.
2. **hub.tellurium.com** Selenium Grid hub machine that drives the tests.
3. **rc.tellurium.com** Multiple Selenium RC server running and registered to the Selenium Grid HUB.

The actual test execution is completed on this machine. Register as many Selenium RC servers as required. However, be realistic about the hardware specification.

Download the Selenium Grid from the following URL and extract the contents of the folder on each of these machines.

Tellurium uses Selenium Grid 1.0.3, the current released version. http://selenium-grid.seleniumhq.org/download.html. Figure 2-8 shows an illustration of the environment.

Figure 2-8 Selenium Grid Support Environment



Selenium Grid Support Test Procedure

1. Launch the Selenium Grid Hub on the hub machine. Open up a terminal on the HUB machine **hub.tellurium.com** and go to the download directory of the Selenium Grid.

```
> cd /Tools/selenium-grid-1.0.3
> ant launch-hub
```

Result: The Selenium HUB is launched on the machine with different browsers.

2. Navigate to the following URL location to ensure that the HUB is working properly:

http://hub.tellurium.com:4444/console

3. View the web page with 3 distinct columns:
    a. Configured Environments
    b. Available Remote Controls
    c. Active Remote Controls
4. Have a list of browsers configured by default to run the tests while the list for Available Remote Controls and Active Remote Controls is empty.
5. Launch the Selenium RC servers and register them with the selenium HUB. Open up a terminal on rc.tellurium.com and go to the selenium grid download directory.

```
> cd /Tools/selenium-grid-1.0.3
> ant -Dport=5555 -Dhost=rc.tellurium.com -
DhubURL=http://hub.tellurium.com:4444 \
      -Denvironment="Firefox on Windows" launch-remote-control
```

Result: The command starts a Selenium RC server on this machine.

6. Register the Selenium RC server with the Selenium Grid hub machine as specified by the hubURL.

Note: To register another Selenium RC server on this machine for internet explorer repeat the step on a different port.

```
> cd /Tools/selenium-grid-1.0.3
> ant -Dport=5556 -Dhost=rc.tellurium.com -
DhubURL=http://hub.tellurium.com:4444 -Denvironment="IE on
Windows" launch-remote-control
```

    a. *port* the remote control is listening to. Must be unique on the machine the remote control runs from.
    b. *hostname* Hostname or IP address of the machine the remote control runs on. Must be visible from the Hub machine.
    c. *hub url* Which hub the remote control should register/unregister to. As the hub is running on hostname hub.tellurium.com, the URL is http://hub.tellurium.com:4444

7. Point your browser to the Hub console Once you are successful in replicating a setup similar to the one described above, (http://hub.tellurium.com:4444/console).
8. Verify that all the remote controls registered correctly. Available remote controls list should be updated and have the 2 selenium servers available to run the tests.

9. Run the Tellurium tests against different browsers once the Selenium Hub and the Selenium RC servers on the Grid environment have started.

10. Go to the Tellurium test development machine, the **dev1.tellurium.com**.

11. Open up the TelluriumConfig.groovy.

12. Change the values of the Selenium server and port to ensure the Tellurium requests for the new sessions from the Selenium HUB are received.

13. Verify that the Selenium HUB points to Tellurium tests run on rc.tellurium.com based on the browser of choice.

14. Change the values for the following properties:

    a. *runInternally*: ensures that the Selenium Server on the local machine is not launched.

    b. *serverHost*: the selenium grid hub machine that has the information about the available selenium rc servers.

    c. *port*: port that Selenium HUB is running on. By default, this port is 4444. This can be changed in the grid_configuraton.yml file if this port is not available on your HUB machine.

    d. *browser*: the browser that comes under the configured environments list on the selenium HUB machine. These values can be changed to a user's choice in the grid_configuration.yml file.

```
tellurium{
    //embedded selenium server configuration
    embeddedserver {
        //port number
        port = "4444"
        //whether to use multiple windows
        useMultiWindows = false
        //whether to run the embedded selenium server.
        //If false, you need to manually set up a selenium
server
        runInternally = false
        //profile location
        profile = ""
        //user-extension.js file
        userExtension = "target/classes/extension/user-
extensions.js"
    }
```

```
    //event handler
    eventhandler{
        //whether we should check if the UI element is presented
        checkElement = false
        //wether we add additional events like "mouse over"
        extraEvent = true
    }
    //data accessor
    accessor{
        //whether we should check if the UI element is presented
        checkElement = true
    }
    //the configuration for the connector that connects the
selenium client
    //to the selenium server
    connector{
        //selenium server host
        //please change the host if you run the Selenium server
remotely
        serverHost = "hub.tellurium.com"
        //server port number the client needs to connect
        port = "4444"
        //base URL
        baseUrl = "http://localhost:8080"
        //Browser setting, valid options are
        //  *firefox [absolute path]
        //  *iexplore [absolute path]
        //  *chrome
        //  *iehta
        browser = "Firefox on Windows"
        //user's class to hold custom selenium methods
associated with user-extensions.js
        //should in full class name, for instance,
"com.mycom.CustomSelenium"
        customClass = "org.tellurium.test.MyCommand"
    }
```

15. The set up is now complete.

16. Run the tests as usual using either the Maven command or the IDE. Notice that the tests are running on rc.tellurium.com and the list for Active Remote Controls is also updated on the hub URL (http://hub.tellurium.com:4444/console) during

the test execution.

Mock Http Server

This feature only exists in Tellurium Core 0.7.0 SNAPSHOT. The MockHttpServer is an embedded http server leveraging the Java 6 http server and it is very convenient method of testing HTML sources directly without running a web server.

Tellurium defines two classes:

1. MockHttpHandler
2. MockHttpServer

MockHttpHandler Class

The MockHttpHandler class processes the http request:

```
public class MockHttpHandler implements HttpHandler {

  private Map<String, String> contents = new HashMap<String,
String>();

  private String contentType = "text/html";

  public void handle(HttpExchange exchange) {
     ......
  }
}
```

The MockHttpHandler method is handle (HttpExchange exchange) and its actions are:

- Reads the request URI
- Finds the corresponding response HTML source from the hash map contents
- Sends the response back to the http client

By default, the response is treated as an HTML source. The user can change this by using the following setter:

```
public void setContentType(String contentType)
```

MockHttpHandler includes two methods to add URI and its HTML source to the hash map contents:

```
1. public void registerBody(String url, String body)
2. public void registerHtml(String url, String html)
```

The MockHttpHandler comes with a default HTML template as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Mock HTTP Server</title>
    </head>
    <body>
    BODY_HTML_SOURCE
    </body>
</html>
```

If `registerBody(String url, String body)` is used, the MockHttpHandler uses the above HTML template to wrap the HTML body. Overwrite the default HTML template by calling `registerHtml(String url, String html)` directly, which uses the whole HTML source provided in the variable *html*.

Usually, the MockHttpHandler is encapsulated by the MockHttpServer and the user does not need to work on it directly.

The MockHttpServer includes an embedded http server, a http handler, and a http port:

```
public class MockHttpServer {
  //default port
  private int port = 8080;

  private HttpServer server = null;
  private MockHttpHandler handler;

  public MockHttpServer() {
    this.handler = new MockHttpHandler();
    this.server = HttpServer.create();
  }

  public MockHttpServer(int port) {
    this.handler = new MockHttpHandler();
    this.port = port;
    this.server = HttpServer.create();
  }

  public MockHttpServer(int port, HttpHandler handler) {
    this.port = port;
```

```
        this.handler = handler;
        this.server = HttpServer.create();
    }

    ......
}
```

MockHttpServer

The MockHttpServer provides three different constructors so the user can overwrite the default values. The MockHttpServer encapsulates the MockHttpHander by providing the following methods:

```
1. public void setContentType(String contentType)
2. public void registerHtmlBody(String url, String body)
3. public void registerHtml(String url, String html)
```

The user can stop and start the server with the following methods:

```
1. public void start()
2. public void stop()
```

Use a modified version of a HTML source provided by one Tellurium user as an example and create the UI module Groovy class as follows:

```
public class ListModule extends DslContext {
  public static String LIST_BODY = """
<div class="thumbnails">
    <ul>
        <li class="thumbnail">
            <img alt="Image 1"
                 src="/images_root/image_pictures/01.jpg"/>
        </li>
        <li class="thumbnail">
            <img alt="Image 2"
                 src="/images_root/image_pictures/02.jpg"/>
        </li>
        <li class="thumbnail">
            <img alt="Image 3"
                 src="/images_root/image_pictures/03.jpg"/>
        </li>
        <li class="thumbnail">
        </li>
```

```
            <li class="thumbnail active">
                <img alt="Image 4"
                     src="/images_root/image_pictures/04.jpg"/>
            </li>
            <li class="thumbnail potd">
                <div class="potd-icon png-fix"/>
                <img alt="Image 5"
                     src="/images_root/image_pictures/05.jpg"/>
            </li>
        </ul>
    </div>
      """

      public void defineUi() {
        ui.Container(uid: "rotator", clocator: [tag: "div", class:
    "thumbnails"]) {
          List(uid: "tnails", clocator: [tag: "ul"], separator: "li") {
            UrlLink(uid: "all", clocator: [:])
          }
        }
      }
    }
```

The reason the HTML source in a Groovy file is included is that the """ quote in Groovy is very easy to present complicated HTML source as a String variable. In Java, the user must concatenate each line of the HTML Source to make it a String variable.

The `defineUi()` defines the UI module for the given HTML source. The major part of the UI module is a List, which uses UI templates to represent a list of links. Tellurium makes it easy and concise to use UI templates to represent UI elements.

Based on the ListModule UI module, define a Tellurium JUnit test case as follows:

```
public class ListTestCase  extends TelluriumJavaTestCase {
    private static MockHttpServer server;

    @BeforeClass
    public static void setUp(){
        server = new MockHttpServer(8080);
        server.registerHtmlBody("/list.html", ListModule.LIST_BODY);
        server.start();
    }
```

```
    @Test
    public void testGetSeparatorAttribute(){
        ListModule lm = new ListModule();
        lm.defineUi();

        connectUrl("http://localhost:8080/list.html");

        attr = (String)lm.getParentAttribute("rotator.tnails[6]",
"class");
        assertEquals("thumbnail potd", attr);
    }

    @AfterClass
    public static void tearDown(){
        server.stop();
    }
}
```

Testing Support

In the package **org.tellurium.test**, Tellurium provides three different ways to write Tellurium tests:

1. TelluriumJavaTestCase
2. TelluriumTestNGTestCase
3. TelluriumGroovyTestCase

TelluriumJavaTestCase

Used for JUnit, and supports the following JUnit 4 annotations:

- @BeforeClass
- @AfterClass
- @Before
- @After
- @Test
- @Ignore

TelluriumTestNGTestCase

Used for TestNG. Similarly, using the following annotations:

- @BeforeSuite
- @AfterSuite
- @BeforeTest
- @AfterTest
- @BeforeGroups
- @AfterGroups
- @BeforeClass

- @AfterClass
- @BeforeMethod
- @AfterMethod
- @DataProvider
- @Parameters
- @Test

TelluriumGroovyTestCase

Used for Groovy test cases.

Data Driven Testing: Tellurium provides the class **`TelluriumDataDrivenModule`** for users to define data driven testing modules.

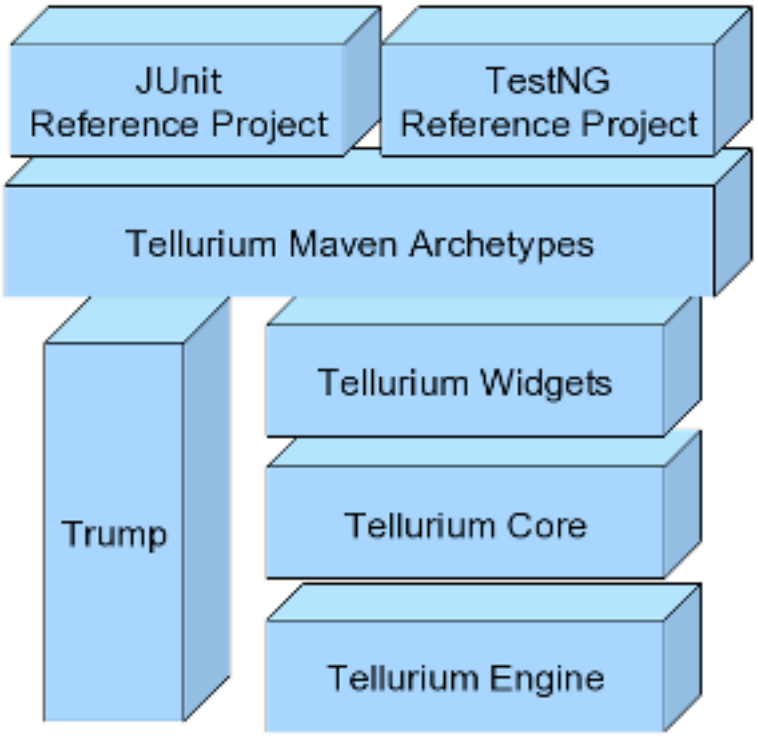
Class **`TelluriumDataDrivenTest`** is used to drive the actual tests.

Tellurium also provides users the capability of writing Tellurium tests and Tellurium data driven tests in pure DSL scripts. The **`DslScriptExecutor`** is used to run the .dsl files.

3 Tellurium Subprojects

Tellurium began as a small core project and quickly generated multiple sub-projects including: JUnit and TestNG Reference projects, Maven Archetypes, TrUMP, Widget extensions, Core, and Engine projects as shown in the following diagram in Figure 3-1:

Figure 3-1 Tellurium Subprojects



- Tellurium JUnit and TestNG Reference Projects: Use the Tellurium project site for examples illustrating how to use different features in Tellurium and how to create Tellurium test cases.
- Tellurium Maven Archetypes: Maven archetypes to generate skeleton Tellurium JUnit and Tellurium TestNG projects using one Maven command.
- Tellurium UI Module Plugin (TrUMP): A Firefox plugin to automatically generate the UI module after users select the UI elements from the web under testing.
- Tellurium Extensions: Dojo Javascript widgets and ExtJS Javascript widgets.
- Tellurium Core: UI module, APIs, DSL, Object to Runtime Locator mapping, and test support.
- Tellurium Engine: Based on Selenium Core with UI module, jQuery selectors, command bundle, and exception hierarchy support.

Tellurium Reference Projects

Tellurium has created reference projects to demonstrate how to use Tellurium for a user's testing project. In the reference projects, Tellurium project web site is used as an example of illustrating how to write real-world Tellurium tests. The reference projects only use tellurium jar files and there are two sub-projects at the time of this document:

- tellurium-junit-java
- tellurium-testng-java

Basically, the two sub-projects are the same and the only difference is that tellurium-junit-java uses JUnit 4 and tellurium-testng-java uses TestNG. Hence, the only focus is on the tellurium-junit-java project.

The tellurium-junit-java project illustrates the following usages of Tellurium:

- How to create your own Tellurium testing project using tellurium jar files.
- How to create your own UI Objects and wire them into Tellurium core
- How to create UI module files in Groovy
- How to create JUnit tellurium testing files in Java
- How to create and run DSL scripts
- How to create Tellurium Data Driven tests
- How to configure Tellurium with the configuration file TelluriumConfig.groovy
- Ant build script
- Maven support
- Support Eclipse, NetBeans, and IntelliJ IDEs

Tellurium Maven Archetypes

Tellurium provides two maven archetypes. For example: tellurium-junit-archetype for Tellurium JUnit test projects and tellurium-testing-archetype for Tellurium TestNG test projects.

Run the following Maven command to create a new JUnit test project:

```
mvn archetype:create -DgroupId=your_group_id -
DartifactId=your_artifact_id \
              -DarchetypeArtifactId=tellurium-junit-archetype \
              -DarchetypeGroupId=tellurium -DarchetypeVersion=0.6.0
```

Without adding the Tellurium Maven repository, specify it in the command line as:

```
mvn archetype:create -DgroupId=your_group_id -
```

```
DartifactId=your_artifact_id \
                    -DarchetypeArtifactId=tellurium-junit-archetype \
                    -DarchetypeGroupId=tellurium -
DarchetypeVersion=0.6.0 \
     -
DarchetypeRepository=http://kungfuters.org/nexus/content/repositories/r
eleases
```

For a TestNG archetype project, use a different archetype:

```
mvn archetype:create -DgroupId=your_group_id -
DartifactId=your_artifact_id \
                    -DarchetypeArtifactId=tellurium-testng-archetype \
                    -DarchetypeGroupId=tellurium -
DarchetypeVersion=0.6.0 \
     -
DarchetypeRepository=http://kungfuters.org/nexus/content/repositories/r
eleases
```

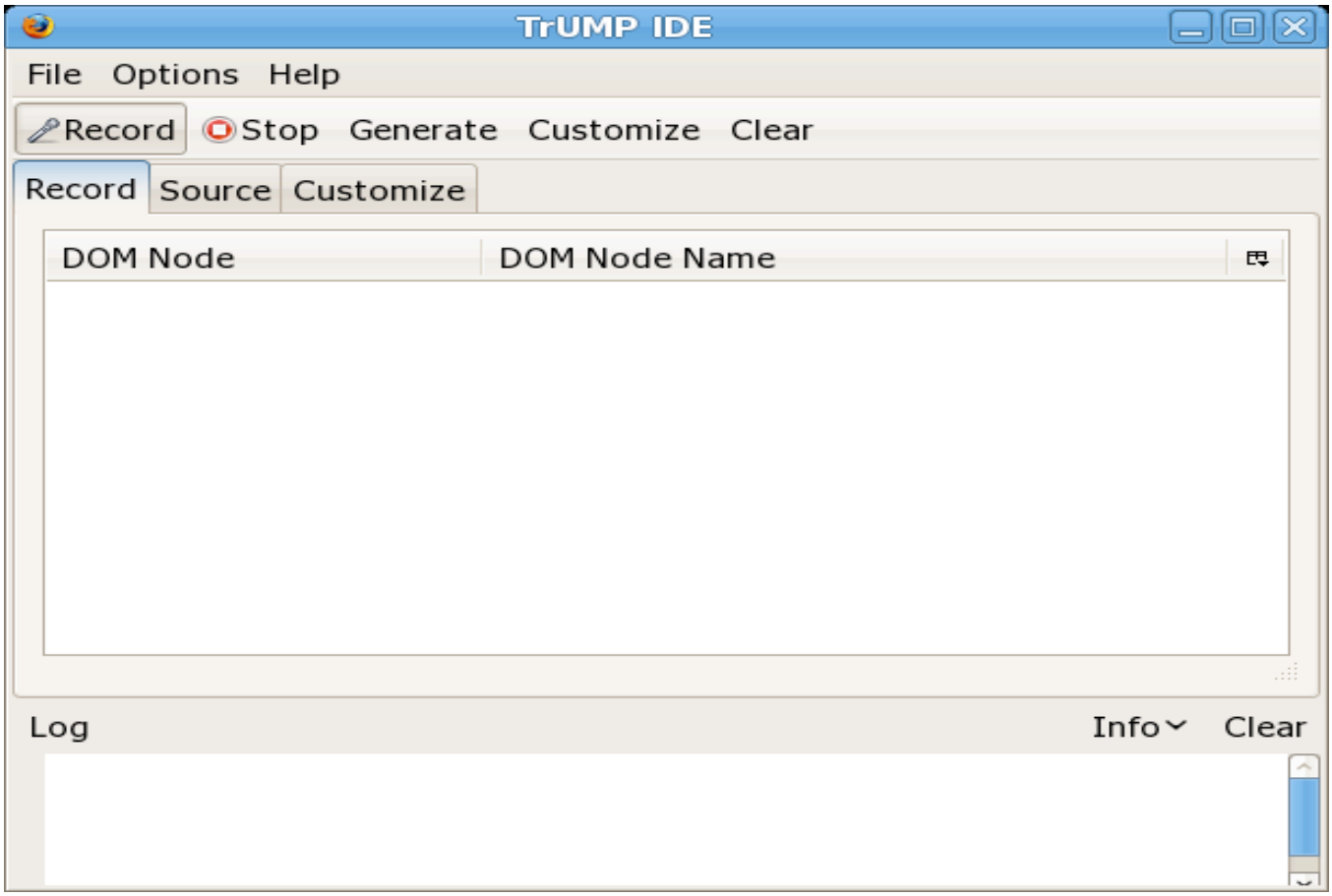
Tellurium UI Module Plugin (TrUMP)

Go to the Tellurium project download page and download the TrUMP xpi file or download the Firefox 3 version directly from the Firefox Addons site at:

https://addons.mozilla.org/en-US/firefox/addon/11035

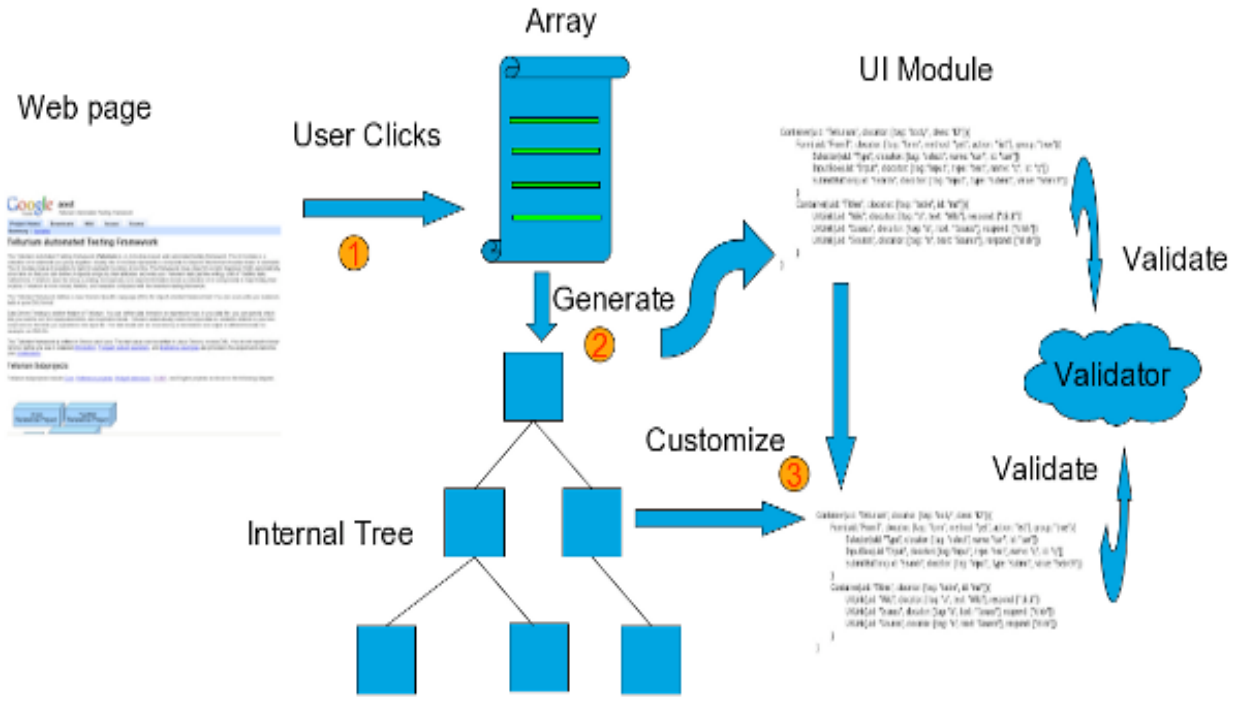
The Tellurium UI Module Plugin (TrUMP) automatically generates UI modules for users. An example of the TrUMP IDE is shown in Figure 3-2:

Figure 3-2 TrUMP IDE



The workflow of TrUMP is shown in Figure 3-3:

Figure 3-3: TrUMP Workflow



TrUMP Workflow Procedure

1. Click onto a web page. The corresponding UI element is pushed into an array.

   Note: If the element is clicked again, the UI element is removed from the array.

2. Click the "Generate" button. TrUMP implements the following two steps:

   - TrUMP generates an internal tree to represent the UI elements using a grouping algorithm. During the tree generating procedure, extra nodes are generated to group UI elements together based on their corresponding location on the DOM tree. The internal tree is very useful and holds all original data that can be used for customization.
   - Once the internal tree is built, TrUMP starts the second step, which is to build the default UI module. For each node in the internal tree, TrUMP generates a UI object based on its tag and whether or not it is a parent node.

3. Click the "Customize" button. TrUMP pulls out the original data held in the internal tree and the current attributes utilized by the UI module to create the "Customize" view. When the user clicks on an element, TrUMP lists all available optional attributes in the View for users to customize.

4. TrUMP attempts to validate the UI module automatically whenever a new UI module is generated or updated. TrUMP evaluates each UI element's XPath the same way that Tellurium generates the runtime XPath from the UI module and verifies if the generated runtime XPath is unique in the current web page.

    - If it is not unique, a red "X" mark is displayed, and the user should modify the element's attribute to make it disappear.
    - If a red "X is not displayed", the validation is completed. The user can export the generated UI module to a groovy file and start to write Tellurium tests based on the generated UI module.

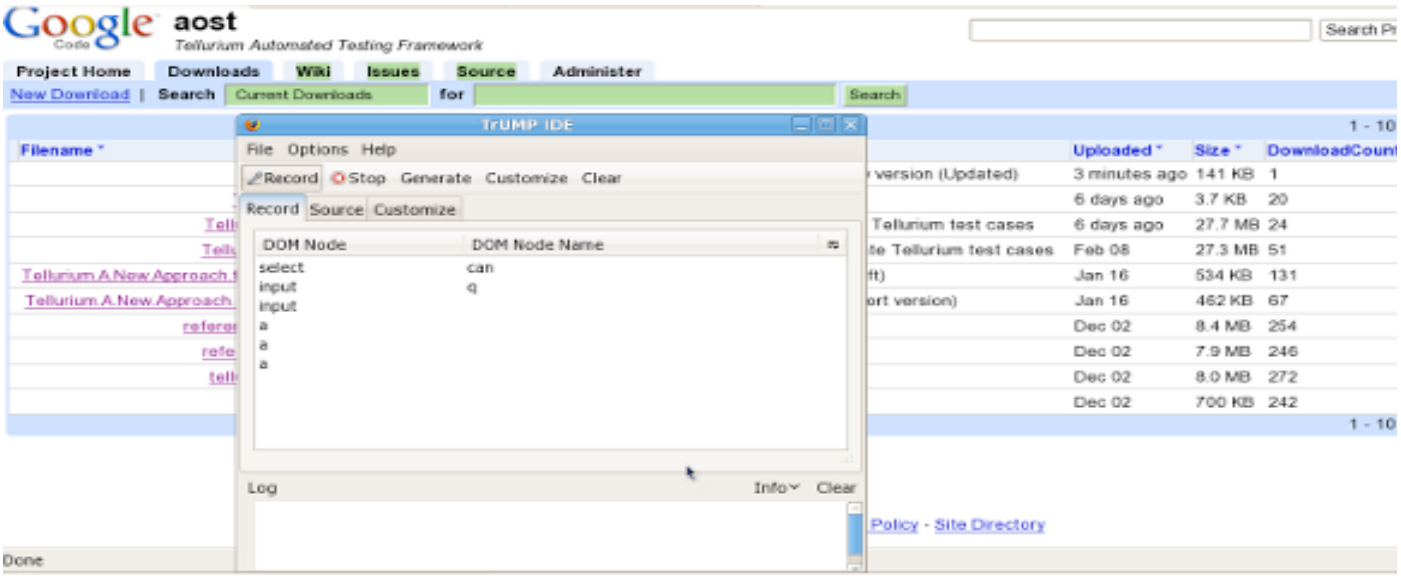
To use TrUMP, complete the following steps:

5. Select "Tools" > "TrUMP IDE" in Firefox.

    - The "Record" button is on by default.
    - Click on "Stop" to stop recording.

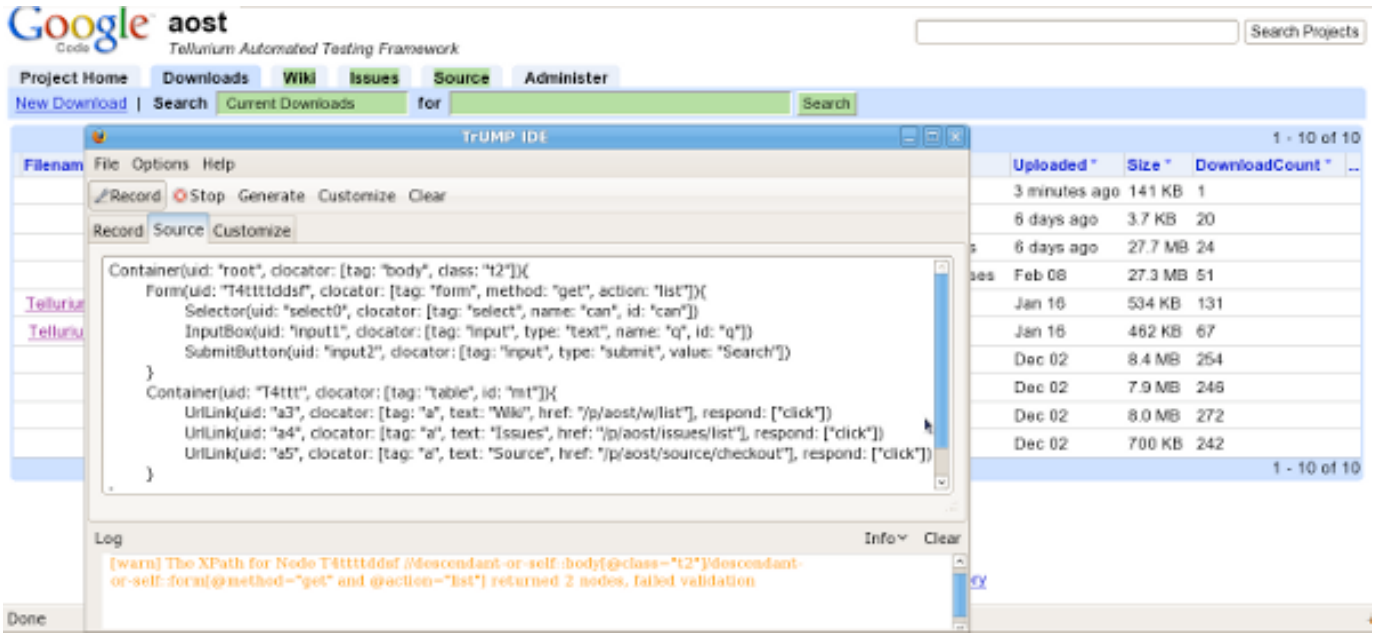
6. Start to use the TrUMP IDE to record specific UI elements that were selected on the WEB. For example, open the Tellurium Download page and click the search elements and the three links as shown in Figure 3-4:

Figure 3-4 Tellurium Download Page



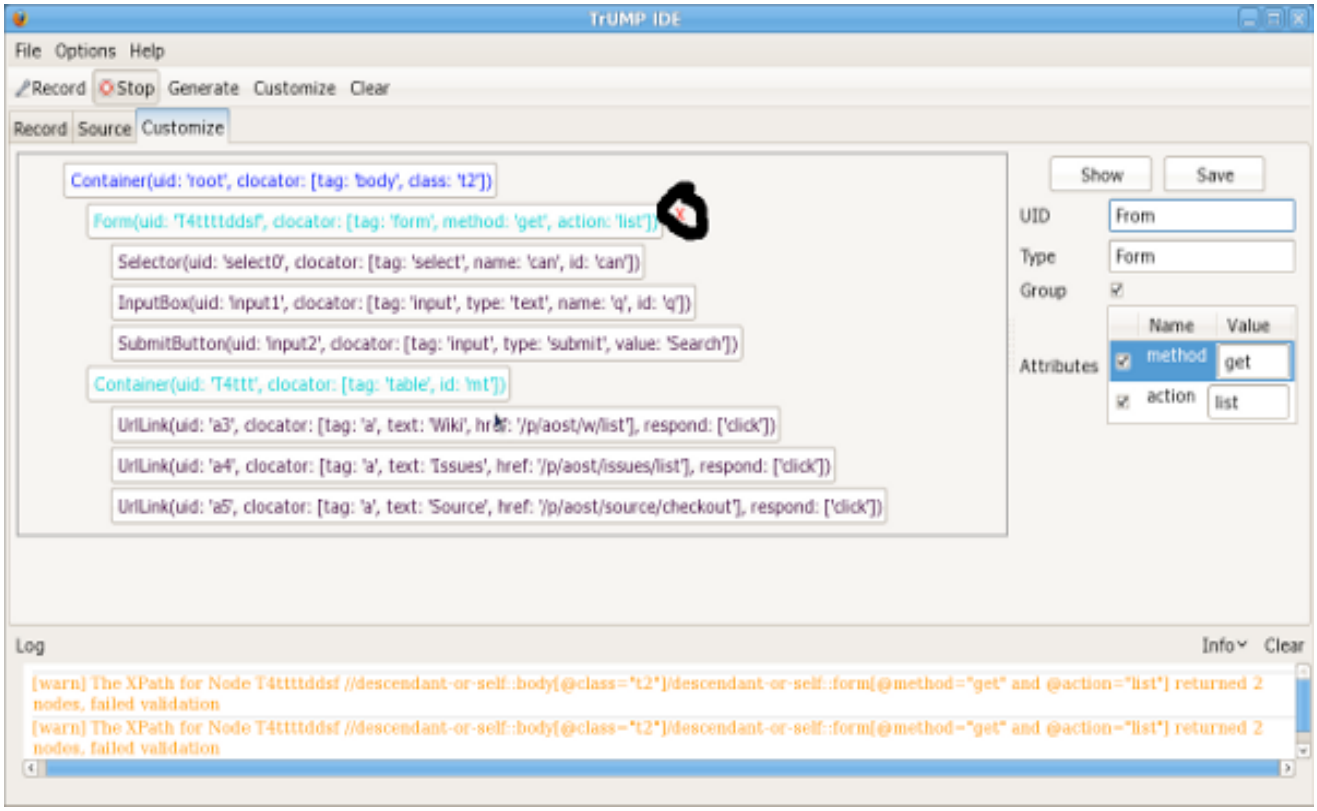
The blue color indicates selected element. Click the selected element again to de-select it. Then, click on the "Generate" button to create the Tellurium UI Module. The user is automatically directed to the "Source" window. See Figure 3-5.

Figure 3-5 Tellurium Download Page Source Window



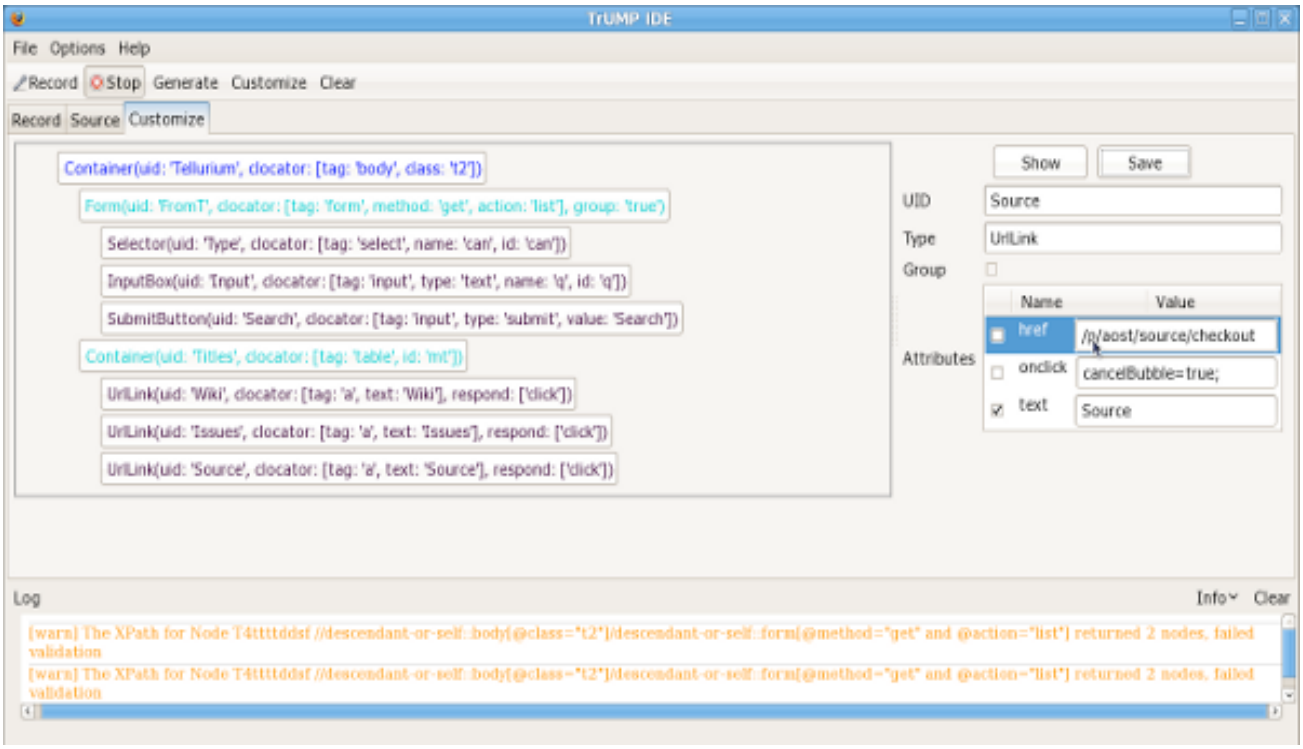
Click the "Customize" button to change the UI module such as UIDs, group locating option, and attributes selected for the UI module. See Figure 3-6

Figure 3-6 Tellurium Download Page Changed UI Module



One red "X" mark is displayed, indicating the UI element's XPath is not unique. Select group, or add more attributes to the UI element. The user sees the new customized UI as shown in Figure 3-7:

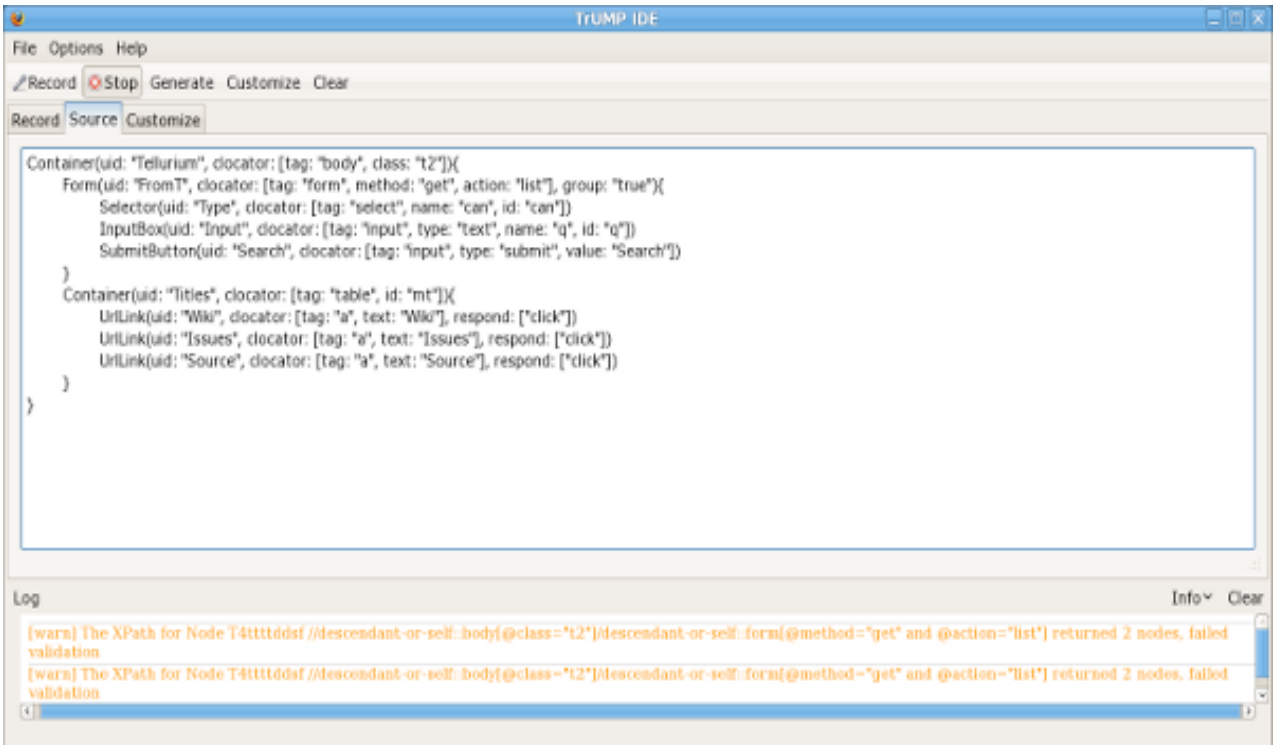
Figure 3-7 Tellurium Download Page Customized UI



Note: The red "X" mark is removed because Tellurium turned on the group locating and the element's xpath is now unique. In the meantime, the UI module in the source tab is automated and updated once the "Save" button is clicked.

The "Show" button shows the actual Web element on the web for which the UI element is represented. See Figure 3-8.

Figure 3-8 Tellurium Download Page Web Element

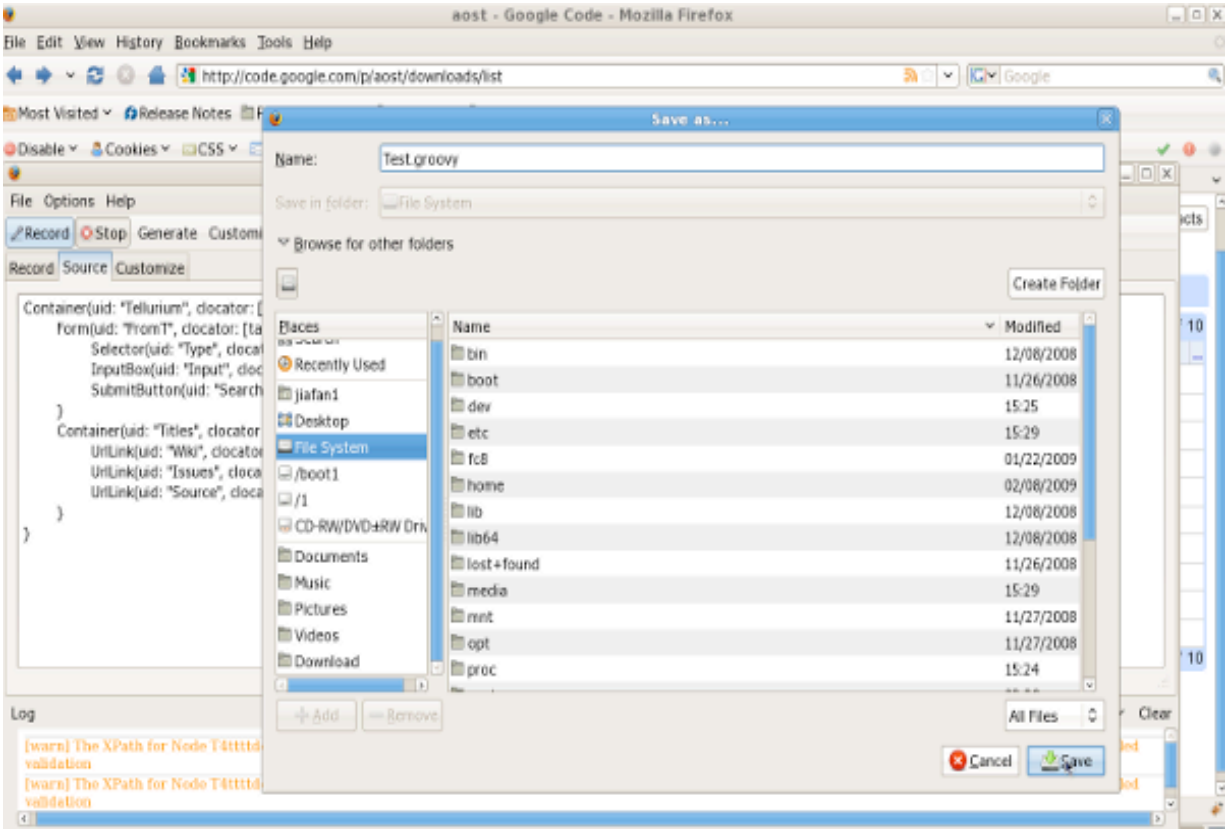


At this point, export the UI module to a groovy file. Be aware that if any error is seen complaining about the directory, first check the "export directory" in Options > Settings and set it to "C:\" or other windows directory for the Windows system before you export the file.

For Linux, the user may find there is no "OK" buton on the option tab, which is caused by the configure "browser.preferences.instantApply" is set to true by default. Point the firefox to "about:config" and change the option to false.

Once this is completed, the user sees the "OK" button. See Figure 3-9.

Figure 3-9 Tellurium Download Page UI Module in Groovy File



Open the groovy file to view the following:

```
package tellurium.ui

import org.tellurium.dsl.DslContext

/**
 *  This UI module file is automatically generated by TrUMP  0.1.0.
 *
*/

class NewUiModule extends DslContext{

  public void defineUi() {
    ui.Container(uid: "Tellurium", clocator: [tag: "body", class:
"t2"]){
      Form(uid: "Form", clocator: [tag: "form", method: "get", action:
"list"],
           group: "true")
      {
        Selector(uid: "DownloadType", clocator: [tag: "select", name:
"can", id: "can"])
```

```
        InputBox(uid: "SearchBox", clocator: [tag: "input", type:
"text", name: "q",
                                                id: "q"])
        SubmitButton(uid: "Search", clocator: [tag: "input", type:
"submit",
                                                  value: "Search"])
      }
      Container(uid: "Title", clocator: [tag: "table", id: "mt"]){
        UrlLink(uid: "Issues", clocator: [tag: "a", text: "Issues"],
respond: ["click"])
        UrlLink(uid: "Wiki", clocator: [tag: "a", text: "Wiki"],
respond: ["click"])
        UrlLink(uid: "Downloads", clocator: [tag: "a", text:
"Downloads"],
                                        respond: ["click"])
      }
    }
  }

        //Add your methods here

}
```

Tellurium Widget Extensions

The Tellurium Widget Extensions are a way of re-using UI components when testing. Tellurium provides the capability to composite UI objects into a widget object and then use that widget directly as if using a Tellurium UI object. The advantage is that a user does not need to deal with the UI at the link or button level for the widget, just work on the high level methods.

Another advantage is that this widget is reusable.

Usually, Java script frameworks provide a number of widgets. For example. the Dojo framework: Tellurium uses the widget DatePicker to prototype the Tellurium widget. For widgets, it is important to include name space to avoid name collision among different widget modules. For example, what is Dojo and ExtJs? Both have the widget Date Picker. After adding the name space, the widget is named as "DOJO_DatePicker".

The DataPicker widget is defined as a normal Tellurium UI module:

```
class DatePicker extends DojoWidget{


    public void defineWidget() {
        ui.Container(uid: "DatePicker", locator:
"/div[@class='datePickerContainer'
                    and child::table[@class='calendarContainer']]"){
            Container(uid: "Title", locator:
"/table[@class='calendarContainer']/thead

 /tr/td[@class='monthWrapper']/table[@class='monthContainer']/tbody
                    /tr/td[@class='monthLabelContainer']"){
                Icon(uid: "increaseWeek", locator:

"/span[@dojoattachpoint='increaseWeekNode']")
                Icon(uid: "increaseMonth", locator:

"/span[@dojoattachpoint='increaseMonthNode']")
                Icon(uid: "decreaseWeek", locator:

"/span[@dojoattachpoint='decreaseWeekNode']")
                Icon(uid: "decreaseMonth", locator:

"/span[@dojoattachpoint='decreaseMonthNode']")
```

```
                TextBox(uid: "monthLabel", locator:

"/span[@dojoattachpoint='monthLabelNode']")

            }

            StandardTable(uid: "calendar", locator:
"/table[@class='calendarContainer']

/tbody/tr/td/table[@class='calendarBodyContainer']"){

                TextBox(uid: "header: all", locator: "")

                ClickableUi(uid: "all", locator: "")

            }

            Container(uid: "year", locator:
"/table[@class='calendarContainer']/tfoot

 /tr/td/table[@class='yearContainer']/tbody/tr/td/h3[@class='yearLabel'
]"){

                Span(uid: "prevYear", locator:
"/span[@class='previousYear' and

                     @dojoattachpoint='previousYearLabelNode']")

                TextBox(uid: "currentYear", locator:
"/span[@class='selectedYear' and

                     @dojoattachpoint='currentYearLabelNode']")

                Span(uid: "nextYear", locator: "/span[@class='nextYear'
and

                     @dojoattachpoint='nextYearLabelNode']")

            }

        }

    }

}
```

Here the XPath is used directly for demonstration purposes only. Usually the composite locator is used instead.

Also, Tellurium defines the following widget methods:

```
    public String getCurrentYear(){

        return getText("DatePicker.year.currentYear")

    }


    public void selectPrevYear(){

        click "DatePicker.year.prevYear"
```

```
    }
```

The widget is treated as a Tellurium UI object and the builder is the same as regular Tellurium objects:

```
class DatePickerBuilder extends UiObjectBuilder{


    public build(Map map, Closure c) {
       //add default parameters so that the builder can use them if not
specified
        def df = [:]
        DatePicker datepicker = this.internBuild(new DatePicker(), map,
df)
        datepicker.defineWidget()


        return datepicker
    }
}
```

Now, the user hooks the widget into the Tellurium Core. Each widget module is compiled as a separate jar file and Tellurium defines a bootstrap class to register all the widgets inside the module.

By default, the full class name of the bootstrap class is `org.tellurium.widget.XXXX.Init`, where the class Init implements the `WidgetBootstrap` interface to register widgets and *XXXX* stands for the widget module name. It is DOJO in this case.

```
class Init implements WidgetBootstrap{
    public void loadWidget(UiObjectBuilderRegistry
uiObjectBuilderRegistry) {
        if(uiObjectBuilderRegistry != null){

 uiObjectBuilderRegistry.registerBuilder(getFullName("DatePicker"),
                                         new DatePickerBuilder())
        }
    }
}
```

Then in the tellurium configuration file `TelluriumConfig?.groovy`, include the module name there:

```
    widget{
        module{
            //define your widget modules here, for example Dojo or
ExtJs
            included="dojo"
        }
    }
```

If the user's own package name is used for the bootstrap class, for example, `com.mycompay.widget.Boot`, then specify the full name here.

```
    widget{
        module{
            //define your widget modules here, for example Dojo or
ExtJs
            included="com.mycompay.widget.Boot"
        }
    }
```

Note: Load multiple widget modules into the Tellurium Core framework by definition.

```
  included="dojo, com.mycompay.widget.Boot"
```

To use the widget, treat a widget as a regular Tellurium UI object. For example:

```
class DatePickerDemo extends DslContext{


    public void defineUi() {
        ui.Form(uid: "dropdown", clocator: [:], group: "true"){
            TextBox(uid: "label", clocator: [tag: "h4", text:
"Dropdown:"])
            InputBox(uid: "input", clocator: [dojoattachpoint:
"valueInputNode"])
            Image(uid: "selectDate", clocator: [title: "select a date",
                  dojoattachpoint: "containerDropdownNode", alt:
"date"])
            DOJO_DatePicker(uid: "datePicker", clocator: [tag: "div",
                            dojoattachpoint: "subWidgetContainerNode"])
        }
    }
```

```
}
```

Then on the module file DatePickerDemo, call the widget methods instead of dealing with low level links, buttons, etc.

To make the testing more expressive, Tellurium provides an onWidget method:

```
onWidget(String uid, String method, Object[] args)
```

In that way, call the widget methods as follows:

```
onWidget "dropdown.datePicker", selectPrevYear
```

Tellurium Core

The Tellurium Core does **Object to Locator Mapping (OLM)** automatically at runtime so that UI objects are simply defined by their attributes. For example, **Composite Locators** denoted by the "clocator".

Furthermore, Tellurium Core uses the **Group Locating Concept (GLC)** to exploit information inside a collection of UI components to help find their locators.

The Tellurium Core defines a new **Domain Specific Language (DSL)** for web testing. One very powerful feature of Tellurium Core is that a user can use **UI templates** to represent many identical UI elements or dynamic sizes of different UI elements at runtime. This is extremely useful to test a dynamic web such as a data grid.

One typical data grid example is as follows:

```
ui.Table(uid: "table", clocator: [:]){
   InputBox(uid: "row: 1, column: 1", clocator: [:])
   Selector(uid: "row: *, column: 2", clocator: [:])
   UrlLink(uid: "row: 3, column: *", clocator: [:])
   TextBox(uid: "all", clocator: [:])
}
```

Data Driven Testing is another important feature of Tellurium Core where a user can define data format in an expressive way. In the data file, a user specifies which test to run, the input parameters, and expected results.

Tellurium automatically binds the input data to variables defined in the test script and runs the tests specified in the input file. The test results are recorded by a test listener and the output is in different formats. For example, an XML file.

The Tellurium Core is written in Groovy and Java. The test cases can be written in Java, Groovy, or pure DSL. A user does not need to know Groovy before using it because the UI module definition and actions on UIs are written in DSLs and the rest may be written in Java syntax.

Tellurium Engine

Up to Tellurium 0.6.0, Tellurium still leveraged Selenium core as the test driving engine. Selenium has a rich set of APIs to act on individual UI elements, and Tellurium can use them directly. Tellurium embedded its own small engine in the Selenium core to support the following enhancements:

- jQuery selector support by adding a new jQuery selector locate strategy so that the Selenium core can handle the jQuery selectors passed in from Tellurium core.
- jQuery selector caching to increase the reuse of previously located DOM references and reduce the UI element locating time.
- Bulk data retrieval. For example, the following Tellurium method obtains all data in a table with just one method call, thus improving speed dramatically:

```
String getAllTableCellText(String uid)
```

- New APIs for partial matching of attributes, CSS query, and more.

Using Selenium core as the test driving engine is a quick solution for Tellurium, but it suffers some drawbacks such as low efficiency because Selenium core does not have built-in support for UI modules.

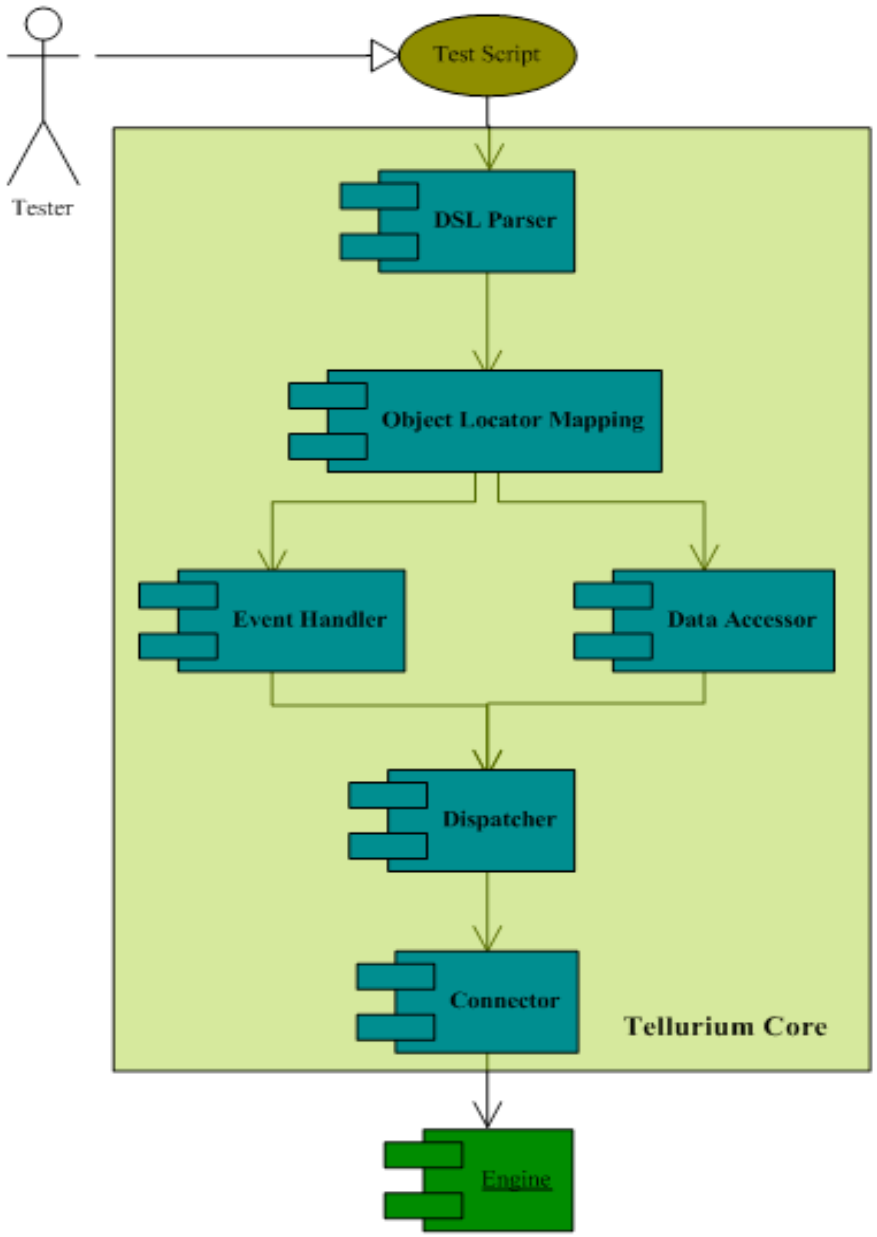
Tellurium is developing a new engine to gradually replace the Selenium core in order to achieve the following goals:

1. Command bundle to reduce round-trip time between Tellurium core and the Tellurium Engine
2. UI module caching at the Engine side to reuse the discovered locator in the UI module no matter if the locator is XPath or a jQuery selector
3. Group locating at the Engine side to exploit more information about the UI elements in the UI module to help locate UI elements
4. Exception hierarchy so that the Engine returns meaningful error codes back to the Tellurium core
5. Ajax response notification

4 Tellurium Architecture

The Tellurium framework architecture is shown in Figure 4-1:

Figure 4-1. Tellurium Framework Architecture



The DSL parser consists of the DSL Object Parser, Object Builders, and the Object Registry.

Using Groovy builder pattern, UI objects are defined expressively and in a nested fashion. The DSL object parser parses the DSL object definition recursively and uses object builders to build the objects on the fly. An object builder registry is designed to hold all predefined UI object builders in the Tellurium framework, and the DSL object parser looks at the builder registry to find the appropriate builders.

Since the registry is a hash map, you can override a builder with a new one using the same UI name. Users can also add their customer builders into the builder registry. The DSL object definition always comes first with a container type object. An object registry (a hash map) is used to store all top level UI Objects. As a result, for each DSL object definition, the top object IDs must be unique in the DslContext. The object registry is used by the framework to search for objects by their IDs and fetch objects for different actions.

The Object Locator Mapping (OLM) is the core of the Tellurium framework and it includes UI ID mapping, XPath builder, jQuery selector builder, and Group Locating.

The UI ID supports nested objects. For example, "menu.wiki" stands for a URL Link "wiki" inside a container called "menu".

The UI ID also supports one-dimensional and two-dimensional indices for tables and lists. For example, `"main.table[2][3]"` stands for the UI object of the 2nd row and the 3rd column of a table inside the container "main".

XPath builder builds the XPath from the composite locator. For example, a set of attributes. Starting with version 0.6.0, Tellurium supports jQuery selectors to address the problem of poor performance of XPath in Internet Explorer.

jQuery selector builders are used to automatically generate jQuery selectors instead of XPath with the following advantages:

- Provides faster performance in IE
- Leverages the power of jQuery to retrieve bulk data from the web by testing with one method call
- Provides new features using jQuery attribute selectors

The Group Locating Concept (GLC) exploits the group information inside a collection of UI objects to assist in finding the locator of the UI objects collection.

The Eventhandler handles all events such as "click", "type", "select", etc.

The Data Accessor fetches data or UI status from the DOM. The dispatcher delegates

all calls it receives from the Eventhandler and the data accessor attached to the connector is also connected to the Tellurium engine.

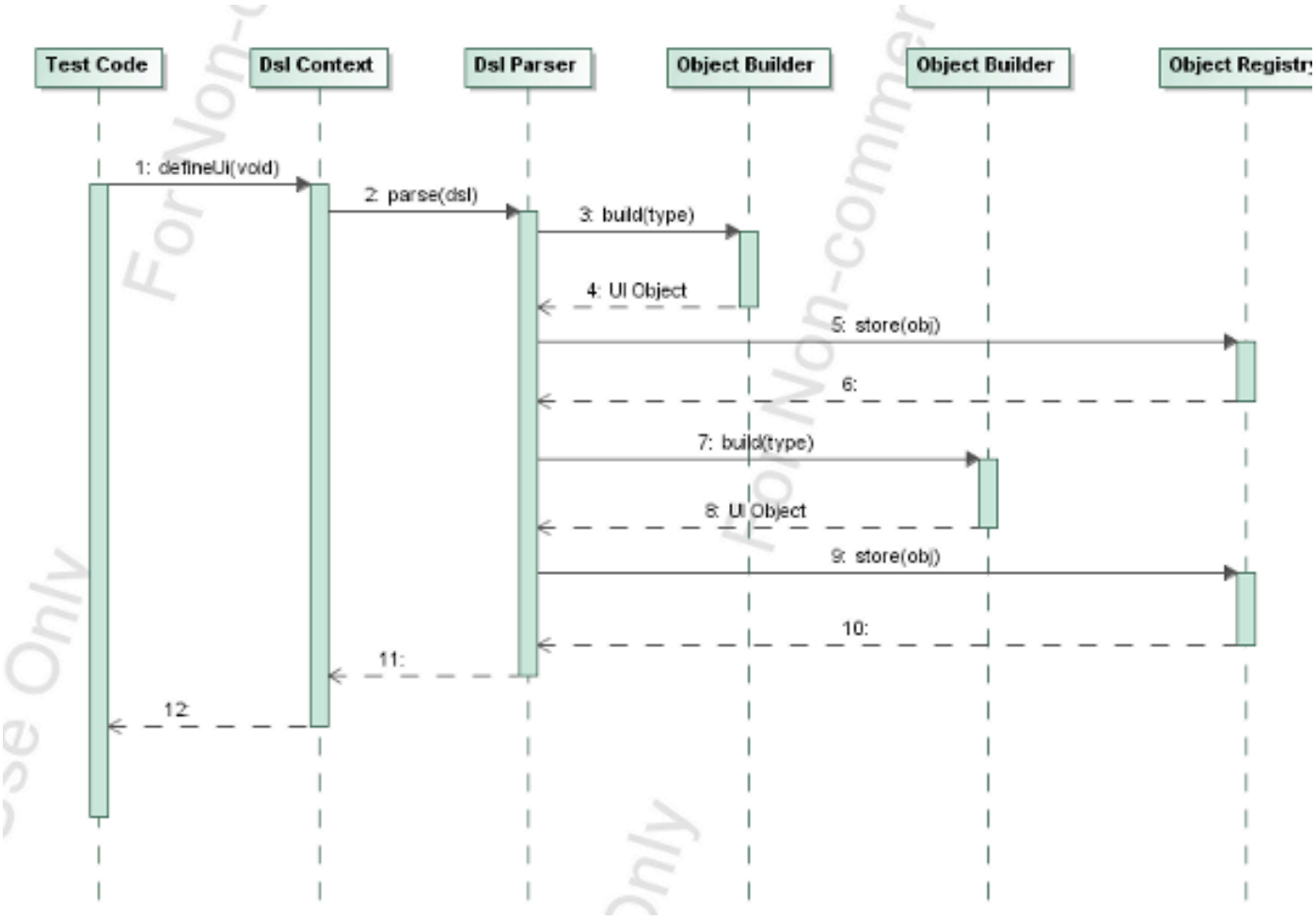
The dispatcher is designed to decouple the rest of the Tellurium framework from the base test driving engine so that it can be switched to a different test driving engine by simply changing the dispatcher logic.

How Tellurium Works

Basically, there are two parts for the Tellurium framework. The first part defines UI objects and the second part works on the UI objects such as firing events and obtaining data or status from the DOM.

The `defineUI` operation is shown in Figure 4-2:

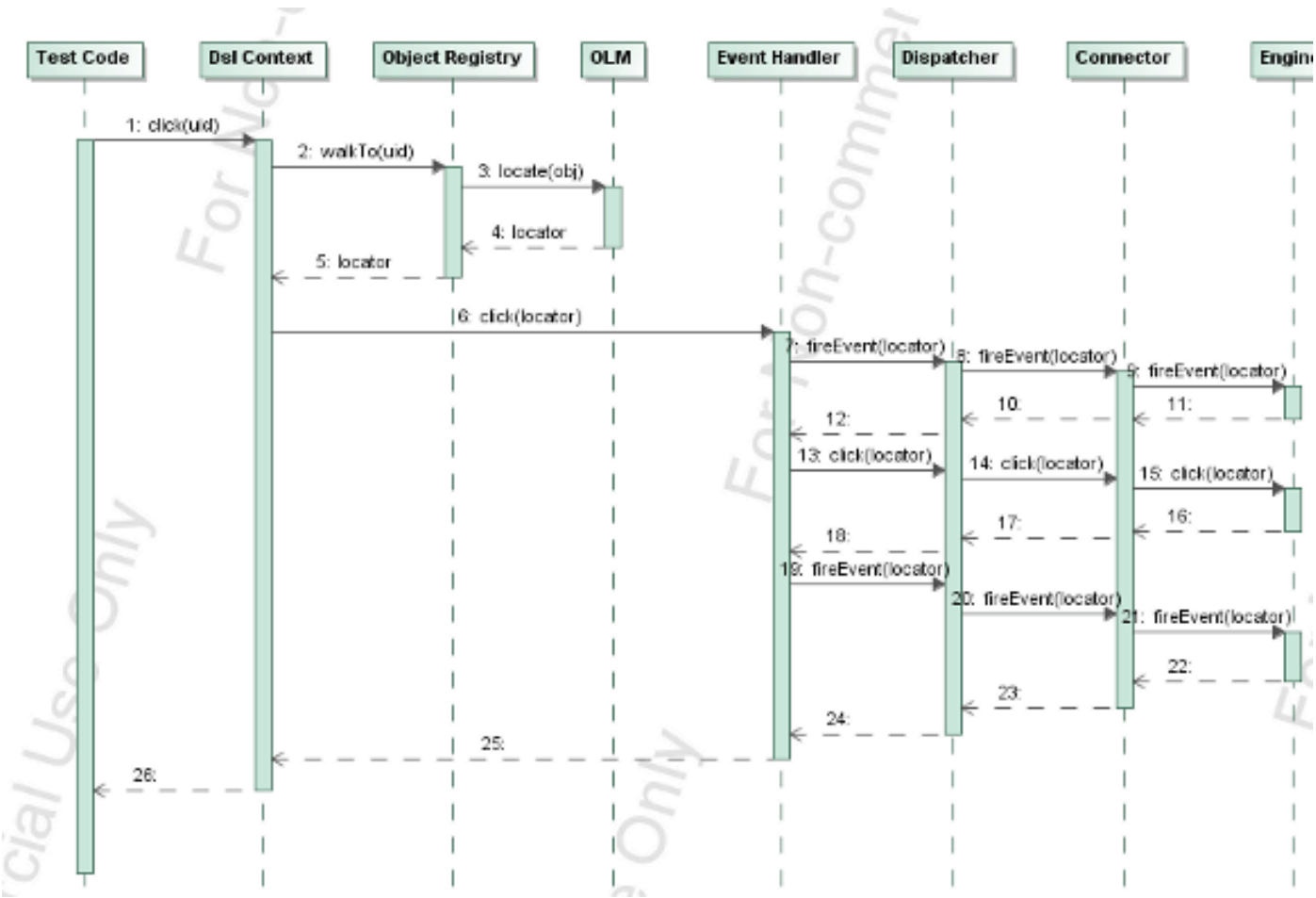
Figure 4-2. `defineUI` Operation Sequence Diagram



When the Test code calls `defineUI()`, the DslContext calls the Dsl Object Parser to parse the UI definition. The Parser looks at each node and calls the appropriate builders to build UI objects. The top level object is stored in the UI Object registry so that the UI object is searched for by uid.

The processing of actions such as clicking on an UI object is illustrated in Figure 4-3:

Figure 4-3. Action Processing Sequence Diagram



The action processing includes the following two parts:

1. The `DslContext` creates a `WorkflowContext` so that meta data such as the relative locator inside it is passed.

2. Then, start looking at the UI object registry by calling the walkTo(uid) method.

   Note: Remember, the UI object registry holds all the top level UI objects.

If the top level UI object is found, the user can recursively call the `walkTo(uid)` method on the next UI object until the UI Object matching the uid is found.

During the `walkTo` method calls, the user starts to aggregate relative XPaths or jQuery selector into the reference locator. Then it is passed on to the next UI object. In this way, the runtime locator is built.

If the UI Object is found, the action is called.

For example, "click" on the UI object and the call is passed on to the EventHandler. Additional JavaScript events may be fired before and/or after the click action as shown in Figure 4-2.

The Action and JavaScript events are passed all the way down from the dispatcher and connector to the Tellurium Engine, which is embedded in the Selenium server at the current stage.

5 Tellurium APIs

Tellurium APIs include all methods defined in DslContext. This chapter provides tables showing the various API methods and the action/result of each method when used:

- DSL Methods
- Data Access Methods
- Test Support DSLs

DSL Methods

This section contains a list of all available Tellurium methods that can be used as DSLs in DslContext and their corresponding DSL syntax.

> **Note** the id here refers to the UiID in the format of "issueSearch.issueType" and the time units are all in milliseconds, if not specified.

Be aware, the user can only apply the methods to the Ui Object if it has these methods previously defined.

METHOD	ACTION/RESULT
`mouseOver id:`	Simulates a user hovering a mouse over the specified element.
`mouseOut id:`	Simulates a user moving the mouse pointer away from the specified element.
`mouseDown id:`	Simulates a user pressing the mouse button (without releasing it yet) on the specified element.
`click id:`	Clicks on a link, button, checkbox or radio button. If the click action causes a new page to load (like a link usually does), call waitForPageToLoad.
`doubleClick id:`	Double clicks on a link, button, checkbox or radio button. If the double click action causes a new page to load (like a link usually does), call waitForPageToLoad.
`clickAt id, coordination:`	Clicks on a link, button, checkbox or radio

METHOD	ACTION/RESULT
	button. If the click action causes a new page to load (like a link usually does), call waitForPageToLoad.
check id:	Clicks on a link, button, checkbox or radio button. If the click action causes a new page to load (like a link usually does), call waitForPageToLoad.
uncheck id:	Uncheck a toggle-button (checkbox/radio).
keyType id, value:	Simulates keystroke events on the specified element, as though the user typed the value key-by-key.
type id, input:	Sets the value of an input field, as though typed it in.
typeAndReturn id, input:	Sets the value of an input field followed by <Return> key.
clearText id:	Resets input field to an empty value.
select id, optionLocator:	Select an option from a drop-down using an option locator.
selectByLabel id, optionLocator:	Select an option from a drop-down using an option label.
selectByValue id, optionLocator:	Select an option from a drop-down using an option value.
addSelectionByLabel id, optionLocator:	Add a selection to the set of selected options in a multi-select element using an option label.
addSelectionByValue id, optionLocator:	Add a selection to the set of selected options in a multi-select element using an option value.
removeSelectionByLabel id, optionLocator:	Remove a selection from the set of selected options in a multi-select element using an option label.
removeSelectionByValue id, optionLocator:	Remove a selection from the set of selected options in a multi-select element using an option value.
removeAllSelections id:	Unselects all of the selected options in a multi-

METHOD	ACTION/RESULT
	select element.
pause time:	Suspend the current thread for a specified milliseconds.
submit id, attribute:	Submit the specified form. This is particularly useful for forms without submit buttons. For example: single-input "Search" forms.
openWindow UID, url:	Opens a popup window. (If a window with that specific ID is not already open). After opening the window, select it using the selectWindow command.
selectWindow UID:	Selects a popup window; once a popup window has been selected, all commands go to that window. To select the main window again, use null as the target.
closeWindow UID:	Close the popup window.
electMainWindows:	Select the original window. For example: the Main window.
selectFrame frameName:	Selects a frame within the current window.
selectParentFrameFrom frameName:	Select the parent frame from the frame identified by the "frameName".
selectTopFrameFrom:	Select the main frame from the frame identified by the "frameName".
waitForPopUp UID, timeout:	Waits for a popup window to appear and load up.
waitForPageToLoad timeout:	Waits for a new page to load.
waitForFrameToLoad frameAdddress, timeout:	Waits for a new frame to load.
runScript script:	Creates a new "script" tag in the body of the current test window, and adds the specified text into the body of the command. Scripts run this way can often be debugged more easily than scripts executed using Selenium's "getEval" command.

METHOD	ACTION/RESULT
	Beware that JS exceptions thrown in these script tags are not managed by Selenium, so the user should wrap the script in try/catch blocks if there is any chance that the script will throw an exception.
captureScreenshot filename:	Captures a PNG screenshot to the specified file.
chooseCancelOnNextConfirmation:	By default, Selenium's overridden window.confirm() function returns true, as if the user had manually clicked OK. After running this command, the next call to confirm() returns false, as if the user had clicked Cancel. Selenium then resumes using the default behavior for future confirmations, automatically returning true (OK) unless/until the user explicitly calls this command for each confirmation.
chooseOkOnNextConfirmation:	Undo the effect of calling chooseCancelOnNextConfirmation. Note: Selenium's overridden window.confirm() function normally automatically returns true, as if the user had manually clicked OK. The user does not need to use this command unless for some reason there is a need to change prior to the next confirmation. After any confirmation, Selenium resumes using the default behavior for future confirmations, automatically returning true (OK) unless/until the user explicitly calls chooseCancelOnNextConfirmation for each confirmation.
answerOnNextPrompt(String answer)):	Instructs Selenium to return the specified answer string in response to the next JavaScript prompt window.prompt()
goBack:	Simulates the user clicking the "back" button on their browser.
refresh:	Simulates the user clicking the "Refresh"

METHOD	ACTION/RESULT
	button on their browser.
dragAndDrop(uid, movementsString):	Drags an element a certain distance and then drops it.
dragAndDropTo(sourceUid, targetUid):	Drags an element and drops it on another element.

Data Access Methods

In addition to the DSL methods, Tellurium provides Selenium-compatible data access methods so that a user can get data or the status of UIs from the web.

METHOD	ACTION/RESULT
`String getConsoleInput():`	Gets input from Console.
`String[] getSelectOptions(id):`	Gets all option labels in the specified select drop-down.
`String[] getSelectedLabels(id):`	Gets all selected labels in the specified select drop-down.
`String getSelectedLabel(id):`	Gets a single selected label in the specified select drop-down.
`String[] getSelectedValues(id):`	Gets all selected values in the specified select drop-down.
`String getSelectedValue(id):`	Gets a single selected value in the specified select drop-down.
`String[] getSelectedIndexes(id):`	Gets all selected indexes in the specified select drop-down.
`String getSelectedIndex(id):`	Gets a single selected index in the specified select drop-down.
`String[] getSelectedIds(id):`	Gets option element ID for selected option in the specified select element.
`String getSelectedId(id):`	Gets a single element ID for selected option in the specified select element.
`boolean isSomethingSelected(id):`	Determines whether some option in a drop-down menu is selected.
`String waitForText(id, timeout):`	Waits for a text event.
`int getTableHeaderColumnNum(id):`	Gets the column header count of a table
`int getTableMaxRowNum(id):`	Gets the maximum row count of a table
`int getTableMaxColumnNum(id):`	Gets the maximum column count of a table
`int getTableFootColumnNum(id):`	Gets the maximum foot column count of a standard table
`int getTableMaxTbodyNum(id):`	Gets the maximum tbody count of a standard table
`int getTableMaxRowNumForTbody(id, index):`	Gets the maximum row number of the index-th tbody of a standard table

METHOD	ACTION/RESULT
`int getTableMaxColumnNumForTbody(id, index):`	Gets the maximum column number of the index-th tbody of a standard table
`int getListSize(id):`	Gets the item count of a list
`getUiElement(id):`	Gets the UIObject of an element.
`boolean isElementPresent(id):`	Verifies that the specified element is somewhere on the page.
`boolean isVisible(id):`	Determines if the specified element is visible. An element can be rendered invisible by setting the CSS "visibility" property to "hidden", or the "display" property to "none", either for the element itself or one if its ancestors. This method will fail if the element is not present.
`boolean isChecked(id):`	Gets whether a toggle-button (checkbox/radio) is checked. Fails if the specified element doesn't exist or isn't a toggle-button.
`boolean isDisabled(id):`	Determines if an element is disabled or not.
`boolean isEnabled(id):`	Determines if an element is enabled or not.
`boolean waitForElementPresent(id, timeout):`	Wait for the Ui object to be present.
`boolean waitForElementPresent(id, timeout, step):`	Wait for the Ui object to be present and check the status by step.
`boolean waitForCondition(script, timeout):`	Runs the specified JavaScript snippet repeatedly until it evaluates to "true". The snippet may have multiple lines, but only the result of the last line will be considered.
`String getText(id):`	Gets the text of an element. This works for any element that contains text. This command uses either the textContent (Mozilla-like browsers) or the innerText (IE-like browsers) of the element, which is the rendered text shown to the user.
`String getValue(id):`	Gets the (whitespace-trimmed) value of an input field (or anything else with a value parameter). For checkbox/radio elements, the value will

METHOD	ACTION/RESULT
	be "on" or "off" depending on whether the element is checked or not.
`String getLink(id):`	Get the href of an element.
`String getImageSource(id):`	Get the image source element.
`String getImageAlt(id):`	Get the image alternative text of an element.
`String getImageTitle(id):`	Get the image title of an element.
`getAttribute(id, attribute):`	Get an attribute of an element.
`getParentAttribute(id, attribute):`	Get an attribute of the parent of an element.
`String getBodyText():`	Gets the entire text of the page.
`boolean isTextPresent(pattern):`	Verifies that the specified text pattern appears somewhere on the rendered page shown to the user.
`boolean isEditable(id):`	Determines whether the specified input element is editable, ie hasn't been disabled. This method will fail if the specified element isn't an input element.
`String getHtmlSource():`	Returns the entire HTML source between the opening and closing "html" tags.
`String getExpression(expression):`	Returns the specified expression.
`getXpathCount(xpath):`	Returns the number of nodes that match the specified xpath, eg. "//table" would give the number of tables.
`String getCookie():`	Return all cookies of the current page under test.
`boolean isAlertPresent():`	Has an alert occurred?
`boolean isPromptPresent():`	Has a prompt occurred?
`boolean isConfirmationPresent():`	Has confirm() been called?
`String getAlert():`	Retrieves the message of a JavaScript alert generated during the previous action, or fail if there were no alerts.
`String getConfirmation():`	Retrieves the message of a JavaScript confirmation dialog generated during the previous action.
`String getPrompt():`	Retrieves the message of a JavaScript question prompt dialog generated during the

METHOD	ACTION/RESULT
	previous action.
`String getLocation():`	Gets the absolute URL of the current page.
`String getTitle():`	Gets the title of the current page.
`String[] getAllButtons():`	Returns the IDs of all buttons on the page.
`String[] getAllLinks():`	Returns the IDs of all links on the page.
`String[] getAllFields():`	Returns the IDs of all input fields on the page.
`String[] getAllWindowIds():`	Returns the IDs of all windows that the browser knows about.
`String[] getAllWindowNames():`	Returns the names of all windows that the browser knows about.
`String[] getAllWindowTitles():`	Returns the titles of all windows that the browser knows about.

Test Support DSLs

Tellurium defines a set of DSLs to support Tellurium tests. The most often used ones include:

openUrl(String url): establish a new connection to Selenium server for the given url. The DSL format is:

```
openUrl url
```

Example:

```
openUrl "http://code.google.com/p/aost/"
```

connectUrl(String url): use existing connect for the given url. The DSL format is:

```
connectUrl url
```

Example:

```
connectUrl "http://code.google.com/p/aost/"
```

Appendix A Tellurium Examples

Custom UI Object

Tellurium supports custom UI objects defined by users. For most UI objects, they must extend the "UiObject" class and then define actions or methods they support.

Container type UI objects, for example, hold other UI objects and extend the Container class. See Tellurium Table and List Objects for more details.

In the tellurium-junit-java project, the custom UI object "SelectMenu" is defined as follows:

```
class SelectMenu extends UiObject{
    public static final String TAG = "div"

    String header = null

    //map to hold the alias name for the menu item in the format of
    //"alias name" : "menu item"
    Map<String, String> aliasMap

    def click(Closure c){
        c(null)
    }

    def mouseOver(Closure c){
        c(null)
    }

    def mouseOut(Closure c){
        c(null)
    }

    public void addTitle(String header){
        this.header = header
    }
```

```
    public void addMenuItems(Map<String, String> menuItems){
        aliasMap = menuItems
    }

    ......

}
```

For each UI object, you must define the builder so that Tellurium knows how to construct the UI object when it parses the UI modules. For example, we define the builder for the "SelectMenu?" object as follows:

```
class SelectMenuBuilder extends UiObjectBuilder{
    static final String ITEMS = "items"
    static final String TITLE = "title"

    public build(Map map, Closure c) {
        def df = [:]
        df.put(TAG, SelectMenu.TAG)
        SelectMenu menu = this.internBuild(new SelectMenu(), map, df)
        Map<String, String> items = map.get(ITEMS)
        if(items != null && items.size() > 0){
           menu.addMenuItems(items)
        }

        menu.addTitle(map.get(TITLE))

        return menu
    }
}
```

You may wonder how to hook the custom objects into Tellurium core so that it can recognize the new type. The answer is simple, you just add the UI object name and its builder class name to Tellurium configuration file TelluriumConfig?.groovy. Update the following section:

```
    uiobject{
        builder{
           SelectMenu="org.tellurium.builder.SelectMenuBuilder"
```

```
        }
    }
```

UI Module

Take the Tellurium issue page as an example, the UI module can be defined as:

```
public class TelluriumIssueModule extends DslContext {


  public void defineUi() {


    //define UI module of a form include issue type selector and issue
search
    ui.Form(uid: "issueSearch", clocator: [action: "list", method: "get"],
group: "true") {
      Selector(uid: "issueType", clocator: [name: "can", id: "can"])
      TextBox(uid: "searchLabel", clocator: [tag: "span", text: "*for"])
      InputBox(uid: "searchBox", clocator: [type: "text", name: "q"])
      SubmitButton(uid: "searchButton", clocator: [value: "Search"])
    }


    ui.Table(uid: "issueResult", clocator: [id: "resultstable", class:
"results"],
             group: "true")
{
      TextBox(uid: "header: 1", clocator: [:])
      UrlLink(uid: "header: 2", clocator: [text: "*ID"])
      UrlLink(uid: "header: 3", clocator: [text: "*Type"])
      UrlLink(uid: "header: 4", clocator: [text: "*Status"])
      UrlLink(uid: "header: 5", clocator: [text: "*Priority"])
      UrlLink(uid: "header: 6", clocator: [text: "*Milestone"])
      UrlLink(uid: "header: 7", clocator: [text: "*Owner"])
      UrlLink(uid: "header: 9", clocator: [text: "*Summary + Labels"])
      UrlLink(uid: "header: 10", clocator: [text: "*..."])


      //define table elements
      //for the border column
      TextBox(uid: "row: *, column: 1", clocator: [:])
```

```
        TextBox(uid: "row: *, column: 8", clocator: [:])
        TextBox(uid: "row: *, column: 10", clocator: [:])
        //For the rest, just UrlLink
        UrlLink(uid: "all", clocator: [:])
      }

    ui.Table(uid: "issueResultWithCache", cacheable: "true",
noCacheForChildren: "false",
             clocator: [id: "resultstable", class: "results"], group:
"true") {
        //define table header
        //for the border column
        TextBox(uid: "header: 1", clocator: [:])
        UrlLink(uid: "header: 2", clocator: [text: "*ID"])
        UrlLink(uid: "header: 3", clocator: [text: "*Type"])
        UrlLink(uid: "header: 4", clocator: [text: "*Status"])
        UrlLink(uid: "header: 5", clocator: [text: "*Priority"])
        UrlLink(uid: "header: 6", clocator: [text: "*Milestone"])
        UrlLink(uid: "header: 7", clocator: [text: "*Owner"])
        UrlLink(uid: "header: 9", clocator: [text: "*Summary + Labels"])
        UrlLink(uid: "header: 10", clocator: [text: "*..."])

        //define table elements
        //for the border column
        TextBox(uid: "row: *, column: 1", clocator: [:])
        TextBox(uid: "row: *, column: 8", clocator: [:])
        TextBox(uid: "row: *, column: 10", clocator: [:])
        //For the rest, just UrlLink
        UrlLink(uid: "all", clocator: [:])
      }

    }

    public List<String> getDataForColumn(int column){
          int nrow = getTableMaxRowNum("issueResult")
          if(nrow > 20) nrow = 20
          List<String> lst = new ArrayList<String>()
```

```
        for(int i=1; i<nrow; i++){
            lst.add(getText("issueResult[${i}][${column}]"))
        }

        return lst
    }

    public List<String> getDataForColumnWithCache(int column){
        int nrow = getTableMaxRowNum("issueResultWithCache")
        if(nrow > 20) nrow = 20
        List<String> lst = new ArrayList<String>()
        for(int i=1; i<nrow; i++){
            lst.add(getText("issueResultWithCache[${i}][${column}]"))
        }

        return lst
    }

    public String[] getAllText(){
      return getAllTableCellText("issueResult")
    }

    public int getTableCellCount(){
      String sel = getSelector("issueResult")
      sel = sel + " > tbody > tr > td"

      return getJQuerySelectorCount(sel)
    }

    public String[] getTableCSS(String name){
      String[] result = getCSS("issueResult.header[1]", name)

      return result
    }

    public String[] getIsssueTypes() {
```

```
    return getSelectOptions("issueSearch.issueType")
  }

  public void selectIssueType(String type) {
    selectByLabel "issueSearch.issueType", type
  }

  public void searchIssue(String issue) {
    keyType "issueSearch.searchBox", issue
    click "issueSearch.searchButton"
    waitForPageToLoad 30000
  }

  public boolean checkamICacheable(String uid){
    UiObject obj = getUiElement(uid)
    boolean cacheable = obj.amICacheable()
    return cacheable
  }
}
```

Groovy Test Case

```
public class TelluriumIssueGroovyTestCase extends GroovyTestCase{

  private TelluriumIssueModule tisp;

  public void setUp() {
    tisp = new TelluriumIssueModule();
    tisp.defineUi();
  }

  public void tearDown() {
  }

  public void testDumpWithXPath() {
```

```
    tisp.disableJQuerySelector();
    tisp.dump("issueSearch");
    tisp.dump("issueSearch.searchButton");
    tisp.dump("issueResult");
  }

  public void testDumpWithJQuerySelector() {
    tisp.useJQuerySelector();
    tisp.exploreSelectorCache = false;
    tisp.dump("issueSearch");
    tisp.dump("issueSearch.searchButton");
    tisp.dump("issueResult");
  }

  public void testDumpWithJQuerySelectorCacheEnabled() {
    tisp.useJQuerySelector();
    tisp.exploreSelectorCache = true;
    tisp.dump("issueSearch");
    tisp.dump("issueSearch.searchButton");
    tisp.dump("issueResult");
  }
}
```

Java Test Case

```
public class TelluriumIssueTestCase extends TelluriumJavaTestCase {
    private static TelluriumIssueModule tisp;

    @BeforeClass
    public static void initUi() {
        tisp = new TelluriumIssueModule();
        tisp.defineUi();
        tisp.useJQuerySelector();
        tisp.enableSelectorCache();
        tisp.setCacheMaxSize(30);
    }

    @Before
    public void setUp(){
        connectUrl("http://code.google.com/p/aost/issues/list");
    }

    @Test
    public void testSearchIssueTypes(){
        String[] ists = tisp.getIsssueTypes();
        tisp.selectIssueType(ists[2]);
        tisp.searchIssue("Alter");
    }

    @Test
    public void testAllIssues(){
        String[] details = tisp.getAllText();
        assertNotNull(details);
        for(String content: details){
            System.out.println(content);
        }
    }
```

```
    @Test
    public void testGetCellCount(){
        int count = tisp.getTableCellCount();
        assertTrue(count > 0);
        System.out.println("Cell size: " + count);
        String[] details = tisp.getAllText();
        assertNotNull(details);
        assertEquals(details.length, count);
    }

    @Test
    public void testCSS(){
        tisp.disableSelectorCache();
        String[] css = tisp.getTableCSS("font-size");
        assertNotNull(css);
    }

    @Test
    public void checkCacheable(){
        boolean result = tisp.checkamICacheable("issueResult[1][1]");
        assertTrue(result);
        result = tisp.checkamICacheable("issueResult");
        assertTrue(result);
    }

    @Test
    public void testGetDataForColumn(){
        long beforeTime = System.currentTimeMillis();
        tisp.getDataForColumn(3);
        long endTime = System.currentTimeMillis();
        System.out.println("Time noCacheForChildren " + (endTime-
beforeTime) + "ms");
        beforeTime = System.currentTimeMillis();
        tisp.getDataForColumnWithCache(3);
        endTime = System.currentTimeMillis();
        System.out.println("Time with all cache " + (endTime-beforeTime) +
"ms");
```

```
    }

    @Test
    public void testCachePolicy(){
        tisp.setCacheMaxSize(2);
        tisp.useDiscardNewCachePolicy();
        tisp.searchIssue("Alter");
        tisp.getTableCSS("font-size");
        tisp.getIsssueTypes();
        tisp.searchIssue("Alter");
        tisp.setCacheMaxSize(30);
    }

    @After
    public void showCacheUsage(){
        int size = tisp.getCacheSize();
        int maxSize = tisp.getCacheMaxSize();
        System.out.println("Cache Size: " + size + ", Cache Max Size: " +
maxSize);
        System.out.println("Cache Usage: ");
        Map<String, Long> usages = tisp.getCacheUsage();
        Set<String> keys = usages.keySet();
        for(String key: keys){
            System.out.println("UID: " + key + ", Count: " +
usages.get(key));
        }
    }
}
```

Data Driven Testing

Tellurium uses the Issue page as the data driven testing example. We define tests to search issues assigned to a Tellurium team member and use the input file to define which team members we want the result for.

We first define a TelluriumIssuesModule? class that extends TelluriumDataDrivenModule? class and includes a method "defineModule". In the "defineModule" method, we define UI modules, input data format, and different tests. The UI modules are the same as defined before for the Tellurium issue page.

The input data format is defined as:

```
fs.FieldSet(name: "OpenIssuesPage") {
   Test(value: "OpenTelluriumIssuesPage")
}


fs.FieldSet(name: "IssueForOwner", description: "Data format for test
SearchIssueForOwner") {
   Test(value: "SearchIssueForOwner")
   Field(name: "issueType", description: "Issue Type")
   Field(name: "owner", description: "Owner")
}
```

As shown, there are two different input data formats. The "Test" field defines the test name and the "Field" field defines the input data name and description. For example, the input data for the test "SearchIssueForOwner?" has two input parameters "issueType" and "owner".

The tests are defined use "defineTest". One of the test "SearchIssueForOwner?" is defined as follows:

```
defineTest("SearchIssueForOwner") {
    String issueType = bind("IssueForOwner.issueType")
    String issueOwner = bind("IssueForOwner.owner")
    int headernum = getCachedVariable("headernum")
    int expectedHeaderNum = getTableHeaderNum()
    compareResult(expectedHeaderNum, headernum)

    List<String> headernames = getCachedVariable("headernames")
    String[] issueTypes = getCachedVariable("issuetypes")
```

```
            String issueTypeLabel = getIssueTypeLabel(issueTypes,
issueType)
            checkResult(issueTypeLabel) {
                assertTrue(issueTypeLabel != null)
            }
            //select issue type
            if (issueTypeLabel != null) {
                selectIssueType(issueTypeLabel)
            }
            //search for all owners
            if ("all".equalsIgnoreCase(issueOwner.trim())) {
                searchForAllIssues()
            } else {
                searchIssue("owner:" + issueOwner)
            }
            ......
        }

    }
```

Use "bind" to tie the variable to input data field. For example, the variable "issueType" is bound to "IssueForOwner?.issueType". For example, field "issueType" of the input Fieldset "IssueForOwner?". "getCachedVariable" is used to get variables passed from previous tests and "compareResult" is used to compare the actual result with the expected result.

The input file format looks like:

```
OpenTelluriumIssuesPage
## Test Name | Issue Type | Owner
SearchIssueForOwner | Open | all
SearchIssueForOwner | All | matt.senter
SearchIssueForOwner | Open | John.Jian.Fang
SearchIssueForOwner | All |  vivekmongolu
SearchIssueForOwner | All |  haroonzone
```

The actual test class is simple:

```
class TelluriumIssuesDataDrivenTest extends TelluriumDataDrivenTest{
```

```
    public void testDataDriven() {

        includeModule org.tellurium.module.TelluriumIssuesModule.class

        //load file
        loadData
"src/test/resources/org/tellurium/data/TelluriumIssuesInput.txt"

        //read each line and run the test script until the end of the file
        stepToEnd()

        //close file
        closeData()

    }
}
```

We first define which Data Driven Module we want to load and then read input data file. After that, we read the input data file line by line and execute the appropriate tests defined in the input file. Finally, close the data file.

The test result looks as follows:

```
<TestResults>
  <Total>6</Total>
  <Succeeded>6</Succeeded>
  <Failed>0</Failed>
  <Test name='OpenTelluriumIssuesPage'>
    <Step>1</Step>
    <Passed>true</Passed>
    <Input>
      <test>OpenTelluriumIssuesPage</test>
    </Input>
    <Assertion Value='10' Passed='true' />
    <Status>PROCEEDED</Status>
    <Runtime>2.579049</Runtime>
  </Test>
  <Test name='SearchIssueForOwner'>
    <Step>2</Step>
```

```
    <Passed>true</Passed>
    <Input>
      <test>SearchIssueForOwner</test>
      <issueType>Open</issueType>
      <owner>all</owner>
    </Input>
    <Assertion Expected='10' Actual='10' Passed='true' />
    <Assertion Value=' Open Issues' Passed='true' />
    <Status>PROCEEDED</Status>
    <Runtime>4.118923</Runtime>
    <Message>Found 10  Open Issues for owner all</Message>
    <Message>Issue: Better way to wait or pause during testing</Message>
    <Message>Issue: Add support for JQuery selector</Message>
    <Message>Issue: Export Tellurium to Ruby</Message>
    <Message>Issue: Add check Alter function to Tellurium</Message>
    <Message>Issue: Firefox plugin to automatically generate UI module for
users</Message>
    <Message>Issue: Create a prototype for container-like Dojo
widgets</Message>
    <Message>Issue: Need to create Wiki page to explain how to setup Maven
and
             use Maven to build multiple projects</Message>
    <Message>Issue: Configure IntelliJ to properly load one maven sub-
project and not
             look for an IntelliJ project dependency</Message>
    <Message>Issue: Support nested properties for Tellurium</Message>
    <Message>Issue: update versions for extensioin dojo-widget and TrUMP
projects</Message>
  </Test>
  <Test name='SearchIssueForOwner'>
    <Step>3</Step>
    <Passed>true</Passed>
    <Input>
      <test>SearchIssueForOwner</test>
      <issueType>All</issueType>
      <owner>matt.senter</owner>
    </Input>
    <Assertion Expected='10' Actual='10' Passed='true' />
```

```
    <Assertion Value=' All Issues' Passed='true' />
    <Status>PROCEEDED</Status>
    <Runtime>4.023694</Runtime>
    <Message>Found 7  All Issues for owner matt.senter</Message>
    <Message>Issue: Add Maven 2 support</Message>
    <Message>Issue: Add Maven support for the new code base with multiple
             sub-projects</Message>
    <Message>Issue: Cannot find symbol class Selenium</Message>
    <Message>Issue: Need to create Wiki page to explain how to setup Maven
and
             use Maven to build multiple projects</Message>
    <Message>Issue: Need to add tar.gz file as artifact in Maven</Message>
    <Message>Issue: Configure IntelliJ to properly load one maven sub-
project and
             not look for an IntelliJ project dependency</Message>
    <Message>Issue: update versions for extensioin dojo-widget and TrUMP
             projects</Message>
  </Test>
  <Test name='SearchIssueForOwner'>
    <Step>4</Step>
    <Passed>true</Passed>
    <Input>
      <test>SearchIssueForOwner</test>
      <issueType>Open</issueType>
      <owner>John.Jian.Fang</owner>
    </Input>
    <Assertion Expected='10' Actual='10' Passed='true' />
    <Assertion Value=' Open Issues' Passed='true' />
    <Status>PROCEEDED</Status>
    <Runtime>3.542759</Runtime>
    <Message>Found 5  Open Issues for owner John.Jian.Fang</Message>
    <Message>Issue: Better way to wait or pause during testing</Message>
    <Message>Issue: Export Tellurium to Ruby</Message>
    <Message>Issue: Add check Alter function to Tellurium</Message>
    <Message>Issue: Create a prototype for container-like Dojo
widgets</Message>
    <Message>Issue: Support nested properties for Tellurium</Message>
```

```
  </Test>
  <Test name='SearchIssueForOwner'>
    <Step>5</Step>
    <Passed>true</Passed>
    <Input>
      <test>SearchIssueForOwner</test>
      <issueType>All</issueType>
      <owner>vivekmongolu</owner>
    </Input>
    <Assertion Expected='10' Actual='10' Passed='true' />
    <Assertion Value=' All Issues' Passed='true' />
    <Status>PROCEEDED</Status>
    <Runtime>3.521415</Runtime>
    <Message>Found 5  All Issues for owner vivekmongolu</Message>
    <Message>Issue: Create more examples to demonstrate the usage of
different UI
             objects</Message>
    <Message>Issue: DslScriptEngine causes NullPointerException when
attempt to
             invoke server.runSeleniumServer() method</Message>
    <Message>Issue: No such property: connector for class:
DslTest</Message>
    <Message>Issue: Firefox plugin to automatically generate UI module for
users</Message>
    <Message>Issue: Setup Firefox plugin sub-project</Message>
  </Test>
  <Test name='SearchIssueForOwner'>
    <Step>6</Step>
    <Passed>true</Passed>
    <Input>
      <test>SearchIssueForOwner</test>
      <issueType>All</issueType>
      <owner>haroonzone</owner>
    </Input>
    <Assertion Expected='10' Actual='10' Passed='true' />
    <Assertion Value=' All Issues' Passed='true' />
    <Status>PROCEEDED</Status>
```

```
    <Runtime>3.475594</Runtime>
    <Message>Found 5  All Issues for owner haroonzone</Message>
    <Message>Issue: Solve HTTPS certificate problem</Message>
    <Message>Issue: Refactor Tellurium project web site examples using
TestNG and
             put them to the tellurium-testng-java sub-project</Message>
    <Message>Issue: Create a new wiki page on how to create custom
tellurium project
             for NetBeans</Message>
    <Message>Issue: Create a new wiki page on how to create custom
tellurium project
             for IntelliJ</Message>
    <Message>Issue: Table operations for large size one seems to be very
slow</Message>
  </Test>
</TestResults>
```

DSL Test Script

In Tellurium, you can create test scripts in pure DSL. Take TelluriumPage?.dsl as an example:

```
//define Tellurium project menu
ui.Container(uid: "menu", clocator: [tag: "table", id: "mt", trailer:
"/tbody/tr/th"],
             group: "true")
{
    //since the actual text is  Project Home, we can use partial match
here.
    //Note "*" stands for partial match
    UrlLink(uid: "project_home", clocator: [text: "*Home"])
    UrlLink(uid: "downloads", clocator: [text: "Downloads"])
    UrlLink(uid: "wiki", clocator: [text: "Wiki"])
    UrlLink(uid: "issues", clocator: [text: "Issues"])
    UrlLink(uid: "source", clocator: [text: "Source"])
}

//define the Tellurium project search module, which includes an input box
//and two search buttons
```

```
ui.Form(uid: "search", clocator: [:], group: "true"){
    InputBox(uid: "searchbox", clocator: [name: "q"])
    SubmitButton(uid: "search_project_button", clocator: [value: "Search
Projects"])
    SubmitButton(uid: "search_web_button", clocator: [value: "Search the
Web"])
}

openUrl "http://code.google.com/p/aost/"
click "menu.project_home"
waitForPageToLoad 30000
click "menu.downloads"
waitForPageToLoad 30000
click "menu.wiki"
waitForPageToLoad 30000
click "menu.issues"
waitForPageToLoad 30000

openUrl "http://code.google.com/p/aost/"
type "search.searchbox", "Tellurium Selenium groovy"
click "search.search_project_button"
waitForPageToLoad 30000

type "search.searchbox", "tellurium selenium dsl groovy"
click "search.search_web_button"
waitForPageToLoad 30000
```

To run the DSL script, you should first compile the code with ant script. In project root, run the following ant task:

```
ant compile-test
```

Then, use the rundsl.sh to run the DSL script:

```
./rundsl.sh src/test/resources/org/tellurium/dsl/TelluriumPage.dsl
```

For Windows, use the rundsl.bat script.

Appendix B Tellurium Frequently Asked Questions (FAQs)

When Did Tellurium Start?

Tellurium was over one year old in June 2009 if we count the date from the day it became an open source project. But actually, Tellurium had been through two phases of prototyping before that. The first prototype was created in 2007 to test our company's Dojo web applications, which was basically a Java framework based on Spring XML wiring and no UI modules. You have to use factories to create all UI objects.

As a result, it was not convenient to use. The second prototype was created in early 2008 to improve the usability of the first prototype. The UI module was introduced in the second prototype. Both prototypes had been used for a few internal projects before it was re-written in Groovy and became an open source project in June 2008. Notice that prototype framework is called AOST and it was officially renamed to the Tellurium Automated Testing framework (Tellurium) in July 2008 when it moved out of the prototyping phase and became a team project.

What Are the Main Differences Between Tellurium and Selenium?

Tellurium was created when I was a Selenium user and tried to address some of the shortcomings of the Selenium framework such as verbosity and fragility to changes. Selenium is a great web testing framework and up to 0.6.0, Tellurium uses Selenium core as the test driving engine. From Tellurium 0.7.0, we will gradually replace the Selenium core with our own Engine.

Although Tellurium was born from Selenium, there are some fundamental differences between Tellurium and Selenium, which mainly come from the fact that Tellurium is a UI module-based testing framework. For example, Tellurium focuses on a set of UI elements instead of individual ones. The UI module represents a composite UI object in the format of nested basic UI elements. For example, the download search module in Tellurium project site is defined as follows:

```
ui.Form(uid: "downloadSearch", clocator: [action: "list", method: "get"],
group: "true") {
   Selector(uid: "downloadType", clocator: [name: "can", id: "can"])
   InputBox(uid: "searchBox", clocator: [name: "q"])
   SubmitButton(uid: "searchButton", clocator: [value: "Search"])
}
```

With the UI module, Tellurium automatically generates runtime locators for you and there is no need to define XPaths or other types of locators by yourself. Tellurium is robust, expressive, flexible, and reusable.

Do I Need to Know Groovy Before I Use Tellurium?

Tellurium Core is implemented in Groovy and Java to achieve expressiveness. But that does not mean you have to be familiar with Groovy before you start to use Tellurium. Tellurium creates DSL expressions for UI module, actions, and testing. Use a Groovy class to implement the UI module by extending the DslContext Groovy class. Then the user can write the rest using Java syntax. The test cases can be created in Java, Groovy, or Dsl scripts. However, we do encourage geting familiar with Groovy to leverage its meta programming features.

To create a Tellurium project, install a Groovy plugin for your IDE. There are Groovy plugins for commonly used IDEs such as Eclipse, Netbeans, and IntelliJ. Refer to the following WIKI pages on how to set up Groovy and use Tellurium in different IDEs,

http://code.google.com/p/aost/wiki/TelluriumReferenceProjectEclipseSetup

http://code.google.com/p/aost/wiki/TelluriumReferenceProjectNetBeansSetup

http://code.google.com/p/aost/wiki/TelluriumReferenceProjectIntelliJSetup

What Unit Test and Functional Test Frameworks Does Tellurium Support?

Tellurium supports both JUnit and TestNG frameworks. Extend TelluriumJavaTestCase for JUnit and TelluriumTestNGTestCase for TestNG. For more details, please check the following WIKI pages:

http://code.google.com/p/aost/wiki/BasicExample

http://code.google.com/p/aost/wiki/Introduction

Tellurium also provides data driven testing. Data Driven Testing is a different way to write tests. For example, test data are separated from the test scripts and the test flow is not controlled by the test scripts, but by the input file instead. In the input file, users can specify which test to run, what the input parameters are, and what the expected results are. More details can be found from "Tellurium Data Driven Testing" WIKI page,

http://code.google.com/p/aost/wiki/DataDrivenTesting

Does Tellurium Provide Any Tools to Automatically Generate UI Modules?

Tellurium UI Model Plugin (TrUMP) is a Firefox Plugin used to automatically generate UI modules simply by clicking on the web page under testing. A user can download it from the Tellurium download page at:

http://code.google.com/p/aost/downloads/list

or from Firefox Addons site at:

https://addons.mozilla.org/en-US/firefox/addon/11035

The detailed user guide for TrUMP 0.1.0 is at:

http://code.google.com/p/aost/wiki/TrUMP

To understand more about how TrUMP works, please read "How does TrUMP work?" at:

http://code.google.com/p/aost/wiki/HowTrUMPWorks

What Build System Does Tellurium Use?

Tellurium supports both Ant and Maven build systems. The ant build scripts are provided in Tellurium core and Tellurium reference projects. For Maven, please check out the Tellurium Maven guide at:

http://code.google.com/p/aost/wiki/MavenHowTo

What is the Best Way to Create a Tellurium Project?

Tellurium provides two reference projects for JUnit and TestNG project, respectively. Use one of them as a template project. Please see the reference project guide at:

http://code.google.com/p/aost/wiki/ReferenceProjectGuide

However, the best and easiest way to create a Tellurium project is to use Tellurium Maven archetypes. Tellurium provides two Maven archetypes. For example, tellurium-junit-archetype and tellurium-testng-archetype for Tellurium JUnit test project and Tellurium TestNG test project, respectively.

As a result, you can create a Tellurium project using one Maven command. For a Tellurium JUnit project, use:

```
mvn archetype:create -DgroupId?=your_group_id -
DartifactId?=your_artifact_id \
-DarchetypeArtifactId?=tellurium-junit-archetype -
DarchetypeGroupId?=tellurium\ -DarchetypeVersion?=0.7.0-SNAPSHOT \ -
DarchetypeRepository?=http://kungfuters.org/nexus/content/repositories/snapshots
```

and for a Tellurium TestNG project, use

```
mvn archetype:create -DgroupId?=your_group_id -
DartifactId?=your_artifact_id \
-DarchetypeArtifactId?=tellurium-testng-archetype -
DarchetypeGroupId?=tellurium \ -DarchetypeVersion?=0.7.0-SNAPSHOT \ -
DarchetypeRepository?=http://kungfuters.org/nexus/content/repositories/snapshots
```

For more details, please read "Tellurium Maven archetypes",

http://code.google.com/p/aost/wiki/TelluriumMavenArchetypes

Where Can I Find API Documents for Tellurium?

The user guide for Tellurium DSLs, other APIs, and default UI objects could be found at:

http://code.google.com/p/aost/wiki/UserGuide

Is There a Tellurium Tutorial Available?

Tellurium provides very detailed tutorials including basic examples, advanced examples, data driven testing examples, and Dsl script examples. We also provide Tellurium Tutorial Series. Please use Tellurium tutorial WIKI page as your starting point,

http://code.google.com/p/aost/wiki/Tutorial

We also provide a quick start, "Ten Minutes To Tellurium", at

http://code.google.com/p/aost/wiki/TenMinutesToTellurium

Tellurium Dependencies

Tellurium is built on top of Selenium at the current stage and it uses Selenium 1.0.1 and Selenium Grid tool 1.0.2. Tellurium 0.6.0 was tested with Grooy 1.6.0 and Maven 2.0.9.

You can go to Tellurium core and run the following Maven command to check the dependencies:

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
--
[INFO] Building Tellurium Core
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
--
[INFO] [dependency:tree]
[INFO] tellurium:tellurium-core:jar:0.6.0
[INFO] +- junit:junit:jar:4.4:compile
[INFO] +- org.testng:testng:jar:jdk15:5.8:compile
[INFO] +- caja:json_simple:jar:r1:compile
[INFO] +- org.stringtree:stringtree-json:jar:2.0.10:compile
[INFO] +- org.seleniumhq.selenium.server:selenium-server:jar:1.0.1-te:compile
[INFO] +- org.seleniumhq.selenium.client-drivers:selenium-java-client-
driver:jar:1.0.1:compile
[INFO] +- org.openqa.selenium.grid:selenium-grid-tools:jar:1.0.2:compile
[INFO] +- org.codehaus.groovy:groovy-all:jar:1.6.0:compile
[INFO] |  +- org.apache.ant:ant:jar:1.7.1:compile
[INFO] |  |  \- org.apache.ant:ant-launcher:jar:1.7.1:compile
[INFO] |  \- jline:jline:jar:0.9.94:compile
[INFO] +- org.codehaus.groovy.maven.runtime:gmaven-runtime-1.6:jar:1.0-rc-
5:compile
[INFO] |  +- org.slf4j:slf4j-api:jar:1.5.6:compile
[INFO] |  +- org.codehaus.groovy.maven.feature:gmaven-feature-
support:jar:1.0-rc-5:compile
[INFO] |  |  \- org.codehaus.groovy.maven.feature:gmaven-feature-api:jar:1.0-
rc-5:compile
[INFO] |  \- org.codehaus.groovy.maven.runtime:gmaven-runtime-
support:jar:1.0-rc-5:compile
[INFO] |     +- org.codehaus.groovy.maven.runtime:gmaven-runtime-api:jar:1.0-
rc-5:compile
[INFO] |     +- org.codehaus.groovy.maven:gmaven-common:jar:1.0-rc-5:compile
[INFO] |     +- org.codehaus.plexus:plexus-utils:jar:1.5.5:compile
[INFO] |     \- com.thoughtworks.qdox:qdox:jar:1.8:compile
[INFO] \- bouncycastle:bcprov-jdk15:jar:140:compile
[INFO] ------------------------------------------------------------------------
--
```

But be aware that some of the dependencies are required ONLY for Maven itself, for example, `gmaven-runtime`, `bouncycastle`, and `plexus`.

What Is the `ui.` in UI Module?

Very often, you will see the ui. symbol when you define Tellurium UI modules. For instance, look at the following GoogleSearchModule UI module:

```
ui.Container(uid: "GoogleSearchModule", clocator: [tag: "td"], group:
"true"){
   InputBox(uid: "Input", clocator: [title: "Google Search"])
   SubmitButton(uid: "Search", clocator: [name: "btnG", value: "Google
Search"])
   SubmitButton(uid: "Imfeelinglucky", clocator: [value: "I'm Feeling
Lucky"])
}
```

If you have read the Tellurium core code, you will find the following line in the `BaseDslContext` class,

```
UiDslParser ui = new UiDslParser()
```

The ui is actually an instance of UiDslParser. On the above UI module, call the method "Container" on UiDslParser with a map of attributes plus a Closure with the following nested code.

```
{
   InputBox(uid: "Input", clocator: [title: "Google Search"])
   SubmitButton(uid: "Search", clocator: [name: "btnG", value: "Google
Search"])
   SubmitButton(uid: "Imfeelinglucky", clocator: [value: "I'm Feeling
Lucky"])
}
```

Look at what the UiDslParser actually does from the source code:

```
class UiDslParser extends BuilderSupport{

   def registry = [:]

   def UiObjectBuilderRegistry builderRegistry = new
UiObjectBuilderRegistry()

   protected Object createNode(Object name) {
   }

   ....
}
```

The UiDslParser extends the Groovy `BuilderSupport` class and works as a parser for what ever you passed in starting from `Container(uid: "GoogleSearchModule", clocator: [tag: "td"], group: "true")` in the above example.

You may notice that the `BuilderSupport` class needs to handle couple call back methods such as:

```
protected Object createNode(Object name)

protected Object createNode(Object name, Object value)

protected Object createNode(Object name, Map map)

protected Object createNode(Object name, Map map, Object value)

protected void nodeCompleted(Object parent, Object node)

protected void setParent(Object parent, Object child)
```

If you are familiar with XML parser, you will see that this is really similar to the XML PUSH style parser. Define call back methods and the parser will parse the message to the end automatically.

The above callback methods are doing the similar thing. For example, to create a UI object when it sees the name like "Container", "InputBox", and "SubmitButton". The different createNode methods are used for different use cases.

Basically, what the UiDslParser does is to get the object name such as "Container" and then look at the UI builder registry to find the builder for that object, then use the builder to build that UI object. The UI builder registry is a hash map and you can find the Container builder by the object name "Container".

Also the UiDslParser will keep the parse results. For example, UI objects in a registry so that you can refer to them by UID such as "GoogleSearchModule.Search", The object hierarchy is handled by the setParent method.

How Do I Add My Own UI Object to Tellurium?

First, create your UI object groovy class by extending class UiObject or Container if it is a container type object. Then, create your UI object builder by extending class UiObjectBuilder. Finally, register your ui builder for your ui object by call method in class TelluriumFramework:

```
public void registerBuilder(String uiObjectName, UiObjectBuilder builder)
```

You can also register your builder in class UiObjectBuilderRegistry if you work on Tellurium source code directly.

From Tellurium 0.4.0, a global configuration file TelluriumConfig.groovy is used to customize Tellurium. You can also define your own UI object in this file as follows,

```
uiobject{
    builder{
       Icon="org.tellurium.builder.IconBuilder"

    }
```

```
}
```

That is to say, create the UI object and its builder and then in the configuration file specify the UI object name and its builder full class name. **Note**: this feature is included in Tellurium 0.5.0. Please check the SVN trunk for details.

How to Build Tellurium from Source

If you want to build Tellurium from source, you can check out the trunk code using the subversion command:

```
svn checkout http://aost.googlecode.com/svn/trunk/ tellurium
```

or using the Maven (`mvn`) command:

```
mvn scm:checkout -
DconnectionUrl=scm:svn:http://aost.googlecode.com/svn/trunk -
DcheckoutDirectory=tellurium
```

Be aware that the Maven command calls the subversion client to do the job and you must have the client installed in your system.

To build the whole project, use:

```
mvn clean install
```

and Maven compiles source code and resources, compiles test code and test resources, runs all tests, and then installs all artifacts to your local repository under YOUR_HOME/.m2/repository.

Sometimes, tests may break and if you still want to proceed, please use the ignore flag:

```
mvn clean install -Dmaven.test.failure.ignore=true
```

To build an individual project, go to that project directory and run the same command as above.

To run the tests, use the command:

```
mvn test
```

The sub-projects under the tools directory include Tellurium Maven archetypes and TrUMP code, you may not really want to build them by yourself. For TrUMP, the artifacts include a .xpi file.

The assembly project just creates a set of tar files and you may not need to build it either.

Tellurium also provides ant build scripts. You may need to change some of the settings in the build.properties file so that it matches your environment. For example, the

settings for javahome and javac.compiler. Then in the project root directory, run command:

```
ant clean
```

to clean up old build and run:

```
ant dist
```

to generate a new artifact, which can be found in the dist directory.

Run the following to compile test code:

```
ant compile-test
```

What is the Issue with Selenium XPath Expressions and Why is There a Need to Create a UI Module?

The problem is not in XPath itself, but the way you use it. If the following XPath locator is:

```
"//div/table[@id='something']/div[2]/div[3]/div[1]/div[6]"
```

then the problem is easily seen. It is not robust. Along the path

```
div -> table -> div -> div ->div -> div
```

if anything is changed there, your XPath is no longer valid. For example, if you add additional UI elements and the new XPath was changed to:

```
"//div[2]/table[@id='something']/div[3]/div[3]/div[1]/div[6]"
```

you would have to keep updating the XPath. For Tellurium, it focuses on element attributes, not the XPath, and it can be adaptive to the changes to some degree.

More importantly, Tellurium uses the group locating concept to use information from a group of UI elements to locate them in the DOM. In most cases, the group of elements are enough to decide their locations in the DOM, that is to say, your UI element's location does not depend on any parent or grandparent elements.

For instance, in the example above, if you use the group locating concept to find locators for the following part of UI elements directly:

```
"div[3]/div[1]/div[6]"
```

then they do not depend on the portion certainly.

```
"div[2]/table[@id='something']/div[3]"
```

1) The UI elements can address any changes in the portion of :

```
"div[2]/table[@id='something']/div[3]"
```

Note: In Tellurium, the user will not use a locator in the format of:

```
"div[3]/div[1]/div[6]"
```

directly.

2) The syntax of:

```
selenium.type("//input[@title='Google Search']", input)
selenium.click("//input[@name='btnG' and @type='submit']")

...

selenium.type("//input[@title='Google Search']", input)
selenium.click("//input[@name='btnG' and @type='submit']")

...

selenium.type("//input[@title='Google Search']", input)
selenium.click("//input[@name='btnG' and @type='submit']")

...
```

everywhere is really ugly to users. Especially if someone needs to take over your code. In Tellurium, the UiID is used and it is very clear to users what you are acting upon.

```
click "google_start_page.googlesearch"
```

3) The test script created by Selenium IDE is a mess of actions, not modularized. Other people may take quite some time to figure out what the script actually does. And it is quite difficult to refactor and reuse them. Even the UI is not changed, there are data dependence there and for most cases, you simply cannot just "record and replay" in practical tests.

In Tellurium, once you defined the UI module, for example, the Google search module, you can always reuse them and write as many test cases as possible.

4) Selenium is cool and the idea is brilliant. But it is really for low level testing only, focusing on one element at a time and it does not have the whole UI module in mind. That is why another tier on top of it is needed so that you can have a UI module-oriented testing script and not the locator-oriented one. Tellurium is one of the frameworks designed for this purpose.

5) As mentioned in 4), Selenium is quite a low level process and it is really difficult to handle more complicated UI components like a data grid. Tellurium can handle them easily. Please see the test scripts for the Tellurium project web site.

Appendix C Support

If you have any questions or problems with using Tellurium, please join our Tellurium user group and then post them at this site. You will receive a response shortly.

Appendix D Resources

Click the following links or copy and paste the URL address to connect to the Tellurium Resources information:

- Tellurium Project
  http://code.google.com/p/aost/wiki/UserGuideResources#Tellurium_Project
- Users' Experiences
  http://code.google.com/p/aost/wiki/UserGuideResources#Users'_Experiences
- Interviews
  http://code.google.com/p/aost/wiki/UserGuideResources#Interviews
- Presentations and Videos
  http://code.google.com/p/aost/wiki/UserGuideResources#Presentations_and_Videos
- IDEs
  http://code.google.com/p/aost/wiki/UserGuideResources#IDEs
- Build
  http://code.google.com/p/aost/wiki/UserGuideResources#Build
- Related
  http://code.google.com/p/aost/wiki/UserGuideResources#Related