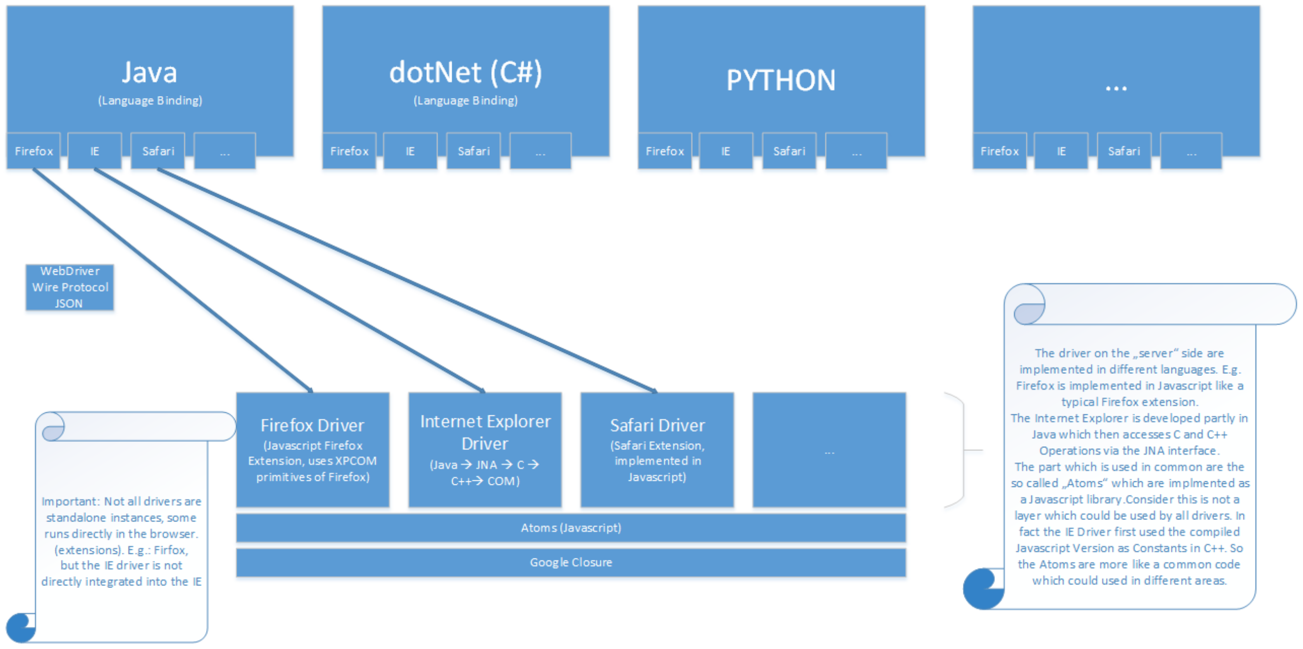
Software Architecture

Lab 1
Selenium

Part 1

Rough Sketch

In order to get an idea how selenium is structured, the following sketch gives a nice overview. Here we can identify several language bindings which are used by different clients. Each language binding has the ability to establish a connection to the browser drivers (e.g. IEDriver, FirefoxDriver, …). These browser drivers act as a server in order to retrieve some information about the web site or to manipulate the input. The important point by using a client server architecture is, that a language independent protocol is introduced for establishing loose coupling. Therefore there is no need to implement each driver in each language. The common layer *Atoms* together with the *Google-Closure*-Layer provide functionalities for all drivers. i.e. The so called Atoms is a javascript library, providing functions which can be executed within all browsers initiated by the respective browser driver.


Java
(Language Binding)
dotNet (C#)
(Language Binding)
PYTHON
...
Firefox
IE
Safari
...
WebDriver Wire Protocol JSON
Firefox Driver (Javascript Firefox Extension, uses XPCOM primitives of Firefox)
Internet Explorer Driver (Java → JNA → C → C++→ COM)
Safari Driver (Safari Extension, implemented in Javascript)
...
Atoms (Javascript)
Google Closure
Important: Not all drivers are standalone instances, some runs directly in the browser. (extensions). E.g.: Firfox, but the IE driver is not directly integrated into the IE
The driver on the „server" side are implemented in different languages. E.g. Firefox is implemented in Javascript like a typical Firefox extension. The Internet Explorer is developed partly in Java which then accesses C and C++ Operations via the JNA interface. The part which is used in common are the so called „Atoms" which are implmented as a Javascript library.Consider this is not a layer which could be used by all drivers. In fact the IE Driver first used the compiled Javascript Version as Constants in C++. So the Atoms are more like a common code which could used in different areas.

Qualities & Tactics

Usability

The quality usability describes how easy it is to use an object, in our case Selenium. There are a vast of different quality attributes regarding usability:

- How easy it is to write my first browser automation with Selenium?
- How easy it is to maintain my code when the codebase is getting bigger?
- Are utility functions which are helping me to reach my goal available?

- How many lines do I need to do certain things?
- How detailed is the documentation?
- Are there active support forums?
- Is there a real customer support?

Some points mentioned above are not part of this elaboration about the software architecture such as support and the quality of the documentation but during our research we used them a lot to gather information and knowledge. Hence we can say that Selenium has a very active community organized in different forums and the documentation of the project is in the most times very useful and has high quality.

Language bindings

The nature of Selenium is that their **users are software developer**. This means all potential users have some programming knowledge but generally you can not assume that all of them have the same skill level or know the same languages. To tackle this problem the contributor to Selenium decide to **offer language bindings** to their browser automation in **all main languages from a web technology point of view**. The available language bindings are written in Java, JavaScript, Perl, PHP, Ruby, C# and Python.

Concise API

Besides the language bindings in different languages the selenium team took also many decisions in the codebase to improve the usability. To have an example we take a look at very basic code snippet from the official documentation:

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;

public class Example  {
    public static void main(String[] args) {
        // Create a new instance of the html unit driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.
        WebDriver driver = new HtmlUnitDriver();

        // And now use this to visit Google
        driver.get("http://www.google.com");

        // Find the text input element by its name
        WebElement element = driver.findElement(By.name("q"));

        // Enter something to search for
        element.sendKeys("Cheese!");
        // Now submit the form. WebDriver will find the form for us from the element
        element.submit();

        // Check the title of the page
        System.out.println("Page title is: " + driver.getTitle());

        driver.quit();
    }
}
```

If the JAR-files are in place this is a fully working Java example and basically opens the URL “http://www.google.com”, writes in the field “q” the word “Cheese!” and submits the selected field. As we can see every line is very concise and therefore very intuitive and easily read. To achieve this the selenium team tries to name all functions and objects as concise as possible. In addition they try to help the developer during coding by placing not to many functions and **only relevant functions** in the classes **which the user uses to interact with the API**. Hence if the users is pressing Ctrl+Space on a object in his favourite IDE the IDE displays only relevant functions and hopefully from the names of the functions he roughly knows what they do. One central class which facilitates this is the class By. It basically offers all different types of possibilities to get a *WebElement*. For example to select by ID, class or XPath. When you are calling a function on the class By it just **delegates** the call to the respective function in the class Webdriver. This function could have been called previously and directly by the user in the class Webdriver but using this class By, the code is more readable and in the class By are only functions to select a *WebElement*. Therefore working in an IDE the IDE just suggest functions to select a *WebElement* and not all the other functions available in the WebDriver class.

Consistent API implementation

Another important chosen tactic is a **consistent API** and implementation **across all different browsers**. When the user uses the API across different browsers the functions should return the values in the same format and the algorithms to retrieve the value should concern all browser-specific quirks. For example the API assures that CSS colours are returned always in the same format. The problem is here that some browser for the color black would return the value “#000” where other browser would return “#000000”, “rgb(0,0,0)” or even “black”.

Pattern support

Selenium tries also to **sustain the user** when he is applying established patterns to his code base. For example the class PageFactory sustains the software developer when he wants to use the page object pattern. With the **PageFactory** he just needs to define a class for each web page or module of a web page and within the class, members for each element on the page or module. Afterwards the class can be passed to the PageFactory which instantiates the class members regarding their name with WebElements from the page or module. The newly created object can then be used for further processing or asserting and the user doesn't need to search for every single element on the page or module.

Help and support functions

To further improve the usability Selenium provides **many utility and help functions**. For instance in the Java language binding there is a whole package with support classes to make the use of the API as easy as possible. Also when it comes to find the executable file of a browser **Selenium tries to find the browser** on the default location for the actual operating system. Selenium tries also to make the **exceptions** from the webdriver as verbose and informative as possible. This is actually helpful when user are contacting the developers for help because then a lot of information is already available in the exception message and it is more unlikely that the users forgets to include something.

Doesn't promise something which can't be kept

Finally Selenium doesn't try to make promises which can't be hold. For instance with the todays technologies it is **not possible** to precisely **define when a page has finished loading**. Some user may say the page is loaded after all pending AJAX calls are finished, some may say after the whole DOM is available. Another problem is to define the term when it comes to AJAX calls which are scheduled every second or pages with an infinite scroll. When is the page loading finished on these pages? For all kind of these problems Selenium pushes the responsibility to the user and the user has to provide his own synchronisation. This maybe sound **inconvenient** and maybe some user claim that this is a usability flaw. But as mentioned it is not possible to tackle this problem in a clear way and therefore **Selenium doesn't try to do it**.

Modifiability

Selenium is a very big open-source software project. The most special thing about the project is that there are used over 5 different programming languages. As mentioned in the last chapter this is necessary to provide a good usability of the project and of course this introduces a lot of complexity in the project. Another important point about the project is that the goal is to sustain as many browsers as possible. This basically means that with language bindings available in x different programming languages it should be possible to steer y different browser. It is very obvious that this **could led to a complexity explosion towards x times y**. To avoid this problem the whole project should be very fast and easy to modify.

Thin language bindings

One tactical design decision was to keep the language bindings as small as possible and to move as **much code as possible in the different webdrivers**. Hence the language bindings easier to maintain because they are small and in addition there is less code written in different languages. They also decided to introduce uniform communication protocol between the language bindings and the different webdrivers. This communication protocol is realized with HTTP and JSON and so very similiar to a RESTful communication. The **uniform protocol allows to reuse a lot of code** and therefore also helps to keep the language bindings thin.

Atoms

Browser automation atoms are JavaScript code blocks which are **used across all driver implementations**. Instead of implementing in every single driver the same functionality the code is written at one central location. During the build this so called atom is then copied as compiled fragment into the webdrivers. The benefit of this tactic is that a bug in the functionality of a driver could be found in one of these atoms. Solving the bug would then **solve the problem in all driver implementations** rather than in only one webdriver implementation. In addition with this approach it is much more easier to test functionality of the webdrivers because all atoms related test can be tested separately.

Cross-language build system

As there are so many different languages and also cross-languages dependencies the build is very complex. To solve this problem Selenium has a **cross-language build system with rake**. This build system allows to build **each system separately with cross-platform dependencies** such as browser automation atoms. This powerful build systems cuts the development time drastically and hence supports the overall modifiability of the project.

In-browser development

One of the most important parts of Selenium are the automation atoms. They are shared over all driver implementation and therefore it is very important to have as few as possible bugs in the atoms. A bug in an automation atom would cause a misbehaviour of all webdrivers which are using the atom. To make the development of the automation atoms as easy as possible the **test suite is built to use directly in the browser.** This is much more comfortable then running the test in a CLI-interface and it is very easy to set breakpoints.

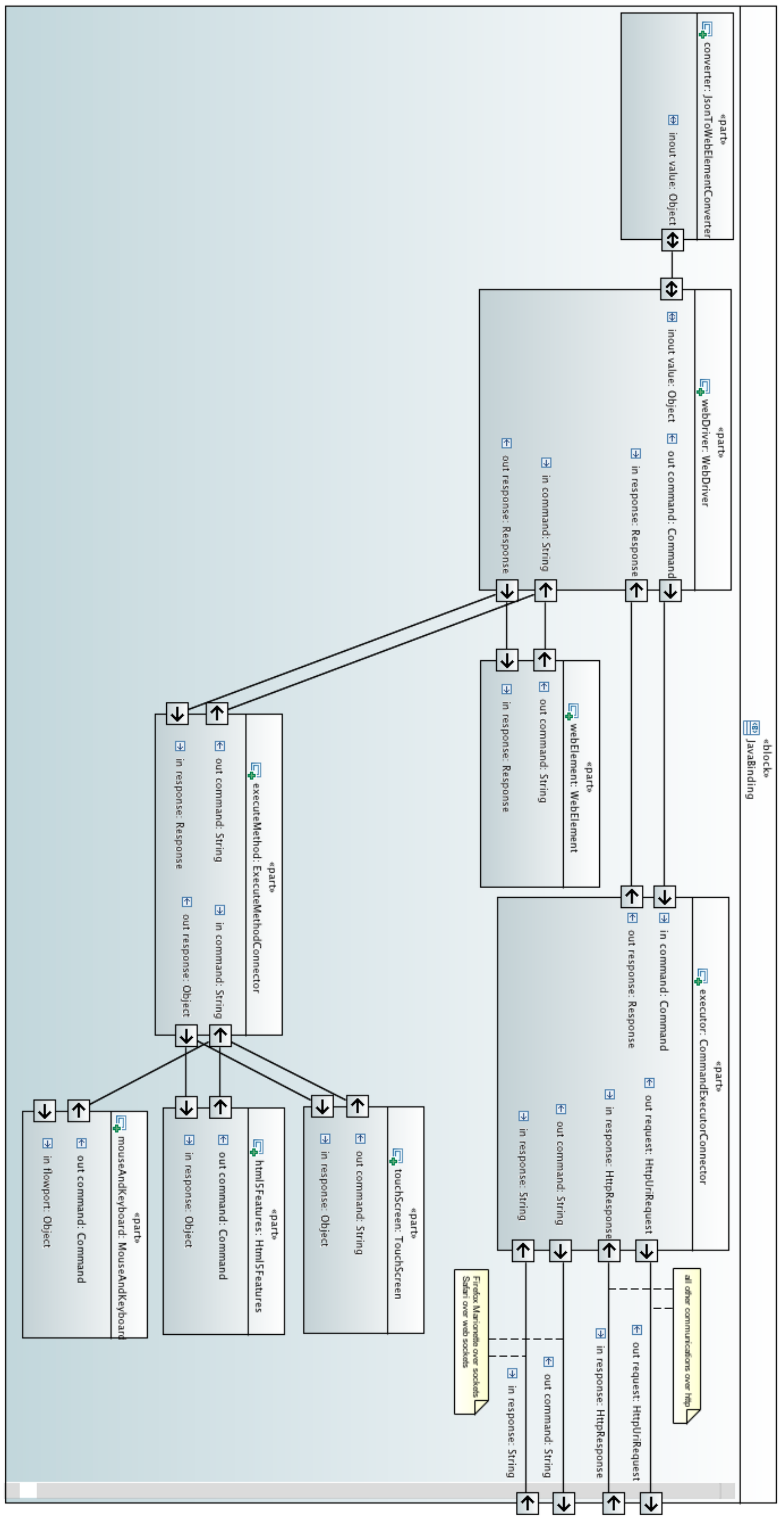
Continuous integration

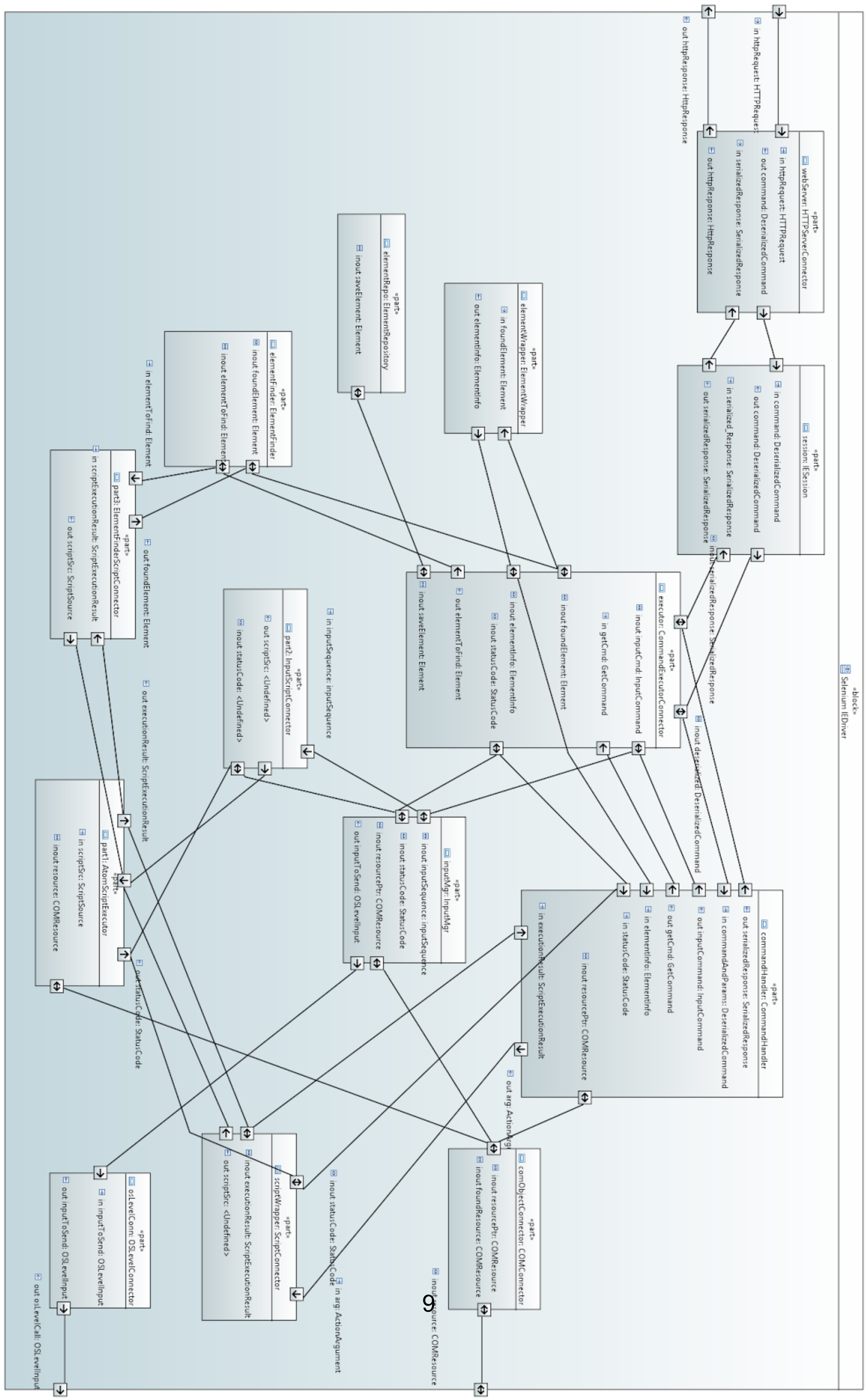
Another point to consider is that not all browser can run on every operating system. Therefore for **most of the developers it is not possible to run the integration tests** for the projects on their systems. To solve this issue the **Selenium project provides a service** hosted by Sauce Labs to run the integrations tests. This lowers the barrier for new contributor and it also speeds up the development time hence improves the modifiability of the project.

Scope of the Model

Considering the diagram in the section “Rough Sketch” we could see the high number of language bindings and browser drivers. In order to describe Selenium Web API appropriately it was decided to focus on one specific browser driver and one language binding implementation. Therefore we chose Java as the binding and the “IEDriver”-implementation as the concrete driver. In contrast to the selected language binding the Internet Explorer Driver was implemented in C++. Hence we used Eclipse and Microsoft Visual Studio 2013 as environment for structural and behavioural analysis of the code. Although it was quite hard to read C++ code, we thought it reflects the components and connectors quite well by preferring deeper instead of broader analysis of n * m language bindings and drivers implementations.

component & connector View




«block»
Selenium IEDriver
in httpRequest: HTTPRequest
out httpResponse: HttpResponse
«part»
webServer: HTTPServerConnector
in httpRequest: HTTPRequest
out command: DeserializedCommand
in serializedResponse: SerializedResponse
out httpResponse: HttpResponse
«part»
session: IESession
in command: DeserializedCommand
out command: DeserializedCommand
in serialized_Response: SerializedResponse
out serializedResponse: SerializedResponse
inout serializedResponse: SerializedResponse
«part»
elementWrapper: ElementWrapper
in foundElement: Element
out elementInfo: ElementInfo
«part»
elementRepo: ElementRepository
inout saveElement: Element
«part»
elementFinder: ElementFinder
inout foundElement: Element
inout elementToFind: Element
in elementToFind: Element
«part»
part3: ElementFinderScriptConnector
in scriptExecutionResult: ScriptExecutionResult
out scriptSrc: ScriptSource
out foundElement: Element
«part»
executor: CommandExecutorConnector
inout inputCmd: InputCommand
in getCmd: GetCommand
inout foundElement: Element
inout elementInfo: ElementInfo
inout statusCode: StatusCode
out elementToFind: Element
inout saveElement: Element
inout deserialized: DeserializedCommand
«part»
part2: InputScriptConnector
out scriptSrc: <Undefined>
inout statusCode: <Undefined>
in inputSequence: inputSequence
out executionResult: ScriptExecutionResult
«part»
part1: AtomScriptExecutor
in scriptSrc: ScriptSource
inout resource: COMResource
out statusCode: StatusCode
«part»
inputMgr: InputMgr
inout inputSequence: inputSequence
inout statusCode: StatusCode
inout resourcePtr: COMResource
out inputToSend: OSLevelInput
«part»
commandHandler: CommandHandler
out serializedResponse: SerializedResponse
in commandAndParams: DeserializedCommand
out inputCommand: InputCommand
out getCmd: GetCommand
in elementInfo: ElementInfo
in statusCode: StatusCode
inout resourcePtr: COMResource
in executionResult: ScriptExecutionResult
out arg: ActionArg
«part»
scriptWrapper: ScriptConnector
inout executionResult: ScriptExecutionResult
out scriptSrc: <Undefined>
inout statusCode: StatusCode
in arg: ActionArgument
«part»
osLevelConn: OSLevelConnector
in inputToSend: OSLevelInput
out inputToSend: OSLevelInput
out osLevelCall: OSLevelInput
«part»
comObjectConnector: COMConnector
inout resourcePtr: COMResource
inout foundResource: COMResource
inout resource: COMResource

Supporting Connectors

Modifiability

As it was already explained, Selenium uses a “REST”-ish interface which controls the communication flow between the language binding and the different browser drivers. Therefore we would point out the ***HTTPServerConnector*** at the browser drivers’ side and the ***CommandExecutorConnector*** at the language binding side. These two connectors facilitate the loose coupling of the two stated entities. Suppose that this would not be the case, tight coupling would the developers force to consider n * m implementations for Selenium WebAPI and the complexity would appear on all locations and language bindings. Due to this failure of encapsulating complexity in only a few locations (e.g. at browser drivers’ side) the quality of modifiability would be ruined. Therefore a language independent HTTP and JSON data exchange using the ***HTTPServerConnector*** and the ***CommandExecutorConnector*** facilitates modifiability.

At the language binding side the *CommandExecutorConnector* **encapsulates** all the functionality which is needed to communicate in the right way with all different browsers. For example the Safari exposes instead of an HTTP endpoint a websocket to communicate. In this case the connector is responsible to translate the command into a string representation and send it over the websocket connection instead of sending a *HTTPUriRequest*. As we can see the component WebDriver doesn’t care with which browser the communication is established. Therefore when we want to change how the language binding communicates to a specific browser we only have to look at the connector. This encapsulation of course increases the modifiability because specific code changes need to be applied only at one location.

After a successful communication with the browser over the corresponding connectors the webdriver receives from his connector a response object. By using the ***JsonToWebElementConverter*** he translates every JSON object with the form {"ELEMENT": id} into a *WebElement*. Here we can see that Selenium doesn’t distinguish between different HTML elements and instantiate everything as a *WebElement.* This conversion and the convention to mark *WebElements* with “ELEMENT” is very generic approach and helps to reduce number of code lines. With less code lines normally when it comes to modify something there are also less places where to modify and hence improves the modifiability.

Finally the importance of the use of **atoms** should be emphasized by considering different connectors which enables access to the execution of atoms in the browser. Of course, as it was already discussed, atoms support the modifiability throughout the application. By providing atom connectors like ***InputScriptConnector***, ***ElementFinderConnector*** or simply the ***ScriptConnector***, the execution of different commands (e.g. key input, element finding etc.) is mainly done within the javascript execution environment of the browser. This ensures that a large part of the complexity could also be encapsulated in one place, which facilitates the reuse of this fundamental building blocks, called **atoms**.

Usability

Usability is a very important non functional property of the Selenium project. The *HTTPServerConnector* and the *CommandExecutorConnector* play again a central role to enable this property. The connectors are mainly responsible for the communication between the language binding and the driver. Without these clear separation it would be very difficult for the developers to maintain so many language bindings and browser drivers at one time. Hence they also play an **important role regarding the usability** of the project.

Another connector is the ***ExecuteMethodConnector***. He is responsible that all components are calling the webdriver in a clear way. These components encapsulate the logic to call commands which steer different interfaces of the browser such as mouse, keyboard, touchscreen or Html5 features like local storage. Normally these function could also lie in the webdriver implementation but with this **separation** the user can execute for instance a mouse click very verbose by calling *webdriver.getMouse().Click();* which of course enhances the usability of Selenium.

Role of the connectors:

HTTPServerConnector

The *HTTPServerConnector* acts as a **converter** by parsing the HTTP request with JSON content. Finally the role of being a **facilitator** is also fulfilled by this connector, due to providing a link between the language binding and the browser driver. By assigning the incoming requests to command types the HTTPServerConnector also prepares the **coordination** task for deciding which command is handled by which command handler.

InputScriptConnector, ElementFinderConnector and ScriptConnector

These three connectors are quite similar, because each of them establishes a link to the execution of javascript atoms. Because of the already mentioned advantage of using atoms, namely the reusability of code throughout the project, the script connectors become important parts of the browser driver. By giving access to this central defined code, we could speak about **facilitators**.

CommandExecutorConnector (IEDriver)

Different roles are combined here. First, several links are provided to different key components like the InputManager or ElementFinder (**Fascilitator**). Secondly, incoming commands are handed over to the specific command handlers, therefore the CommandExecutorConnector coordinates which handler is the right one for a concrete task (**Coordinator**).

CommandExecutorConnector (Language Binding)

The roles of the *CommandExecuteConnector* are **communicator**, **coordinator** and **converter**. It implements different communication mechanism (communicator), coordinates the communication with timeouts and asynchronous calls (coordinator) and converts the commands in the right format for the communication (converter).

JsonToWebElementConverter

The role of this connector is the conversion from JSON to the corresponding *WebElements* of a response. Hence he has only one role, **conversion**.

ExecuteMethodConnector

The *ExecuteMethodConnector* has the main roles **communicator** and **coordinator** as he facilitates the communication between the component *WebDriver* and the different interfaces of the browser.

Model-to-Implementation Information

Selenium IEDriver

The component & connectors’ diagram was realized in a bottom-up way. After having a look on the general architecture and the overall structure we decided to analyze the code base of the IEDriver C++ project. Therefore the implemented components and connectors were built upon classes and program language level interfaces. In order to stay on the same abstraction level and to provide a certain amount added value in the model we did not only just map the classes to components but rather performed some aggregation. E.g. we introduced the **“HTTPServerConnector”** as a bundle of a couple of classes, which take care for the request and response parsing as well as the initialization of the session and the forwarding of JSON-object commands. The following c++ classes, located in the namespace “webdriver-server”, “IEDriver” and “IEDriverServer”, in the implement this behaviour:

- IEServer.cpp
- Server.cc
- WebDriver.cpp
- IEDriverServer.cpp

Code snippets of server.cc show for example the dispatching of command method signature as well as the binding of URLs to Commands, which are main connector tasks of a typical **coordinator** (defines the used commandtype, which is the foundation for deciding which command handler should act):

```
void Server::PopulateCommandRepository() {
...
                      this->AddCommand("/session/:sessionid/touch/click",       "POST",
webdriver::CommandType::TouchClick);
                           this->AddCommand("/session/:sessionid/keys",         "POST",
webdriver::CommandType::SendKeysToActiveElement);
...
}

std::string Server::DispatchCommand(const std::string& uri,
                                     const std::string& http_verb,
                                     const std::string& command_body) {
...
bool session_is_valid = session_handle->ExecuteCommand(
          serialized_command,
          &serialized_response);
...
}
```

Secondly the different script connectors come into play: *InputScriptConnector, ElementFinderConnector* and *ScriptConnector*

The model-to-implementation mapping for these connectors could be described by having a look into the corresponding c++ classes.

- ElementFinder.cpp
- ComandHandler.cpp
- InputManager.cpp

These classes make use of a handle named *Script* which is used similarly in each implementation. The following code snippet shows how *Script* is used to execute a mouse button down command:

```
std::wstring script_source = L"(function() { return function(){" +
                                atoms::asString(atoms::INPUTS) +
                                                                    L";  return
webdriver.atoms.inputs.mouseButtonDown(arguments[0]);" +
                                L"};})();";
CComPtr<IHTMLDocument2> doc;
browser_wrapper->GetDocument(&doc);
Script script_wrapper(doc, script_source, 1);
script_wrapper.AddArgument(this->mouse_state_);
int status_code = script_wrapper.Execute();
```

As we can see in the given code, an additional handle *CComPtr* is used within *Script* which provides access to the com object interface of the IE Browser. So the task of *Script* is to perform the execution of javascript code by making use of the specific IE interface.
In order to model the different connectors *InputScriptConnector, ElementFinderConnector* and *ScriptConnector* we just made a difference for each of them by considering in which context *Script* is used. Of course we could have modelled only one connector, but we decided to identify different ones, because there were different kind of data flows processed. (e.g. *InputScriptConnector* just performs write operations on the current browser page, the *ElementFinderConnector retrieves elements and the ScriptConnector works as a connector for command handler specific queries.*)

Language binding

With the Java language we proceeded the same way as with the IEDriver. After drawing the class diagram we discovered that the **main functionality** is exposed by the **interface *WebDriver***. Hence we aggregated all implementing interfaces and utility classes in the component *WebDriver* where the most important are:

- *RemoteWebDriver*
- *FirefoxDriver*
- *SafariDriver*
- *InternetExplorerDriver*
- *ChromeDriver*
- *HtmlUnitDriver*

The main implementation of the interface *WebDriver* can be found in the class ***RemoteWebDriver***. Most of the other mentioned class **inherit** from the *RemoteWebDriver* and add browser-specific capabilities, properties and behaviour. The exception is the *HtmlUnitDriver* which directly implements the *WebDriver* interface because it doesn't have to talk with the HtmlUnit browser over a remote connection. In the code snippet you can see a part of the execute function of the *RemoteWebDriver*. The code snippet below shows how a command in the *RemoteWebDriver* is passed over to the *CommandExecutorConnector* and in addition it shows the conversion of the response value to *WebElements* with the *JsonToWebElementConverter*.

```
log(sessionId, command.getName(), command, When.BEFORE);
        response = executor.execute(command);
        log(sessionId, command.getName(), command, When.AFTER);

        if (response == null) {
        return null;
        }

        // Unwrap the response value by converting any JSON objects of the form
        // {"ELEMENT": id} to RemoteWebElements.
        Object value = converter.apply(response.getValue());
        response.setValue(value);
```

In the code snippet we have seen the link between the *WebDriver* component and the first class connector *CommandExecutorConnector*. The *CommandExecutorConnector* is responsible for the communication between the language binding and the different web browsers and aggregates many classes and interfaces. The most important ones are:

- *CommandExecutor*
- *HttpCommandExecutor*
- *JsonHttpCommandCodec*

As the name suggests the class *JsonHttpCommandCodec* is responsible to decode and encode the commands sent between the driver and the language binding. The send mechanism is exposed by the *CommandExecutor* interface and the main implementation of this interface is located in the *HttpCommandExecutor*. The following code snippet from the HttpCommandExecutor shows how the command is encoded and then send over the function *fallBackExecute* to the browser. Afterwards the program follows all redirects on the page if there are any and returns the response.

```
HttpRequest request = commandCodec.encode(command);

…

log(LogType.PROFILER, new HttpProfilerLogEntry(command.getName(), true));
HttpResponse response = fallBackExecute(context, httpMethod);
log(LogType.PROFILER, new HttpProfilerLogEntry(command.getName(), false));

response = followRedirects(client, context, response, /* redirect count */0);

return createResponse(response, context);
```

Complementary View

For the complementary view we used an interaction view. This view describes how the different components and connectors interact with each other and how the different browsers can be managed with just one single implementation.
For this we used the component & connector view as a basis. It gives an overview which components must interact in some way and on which basis this is achieved. The first diagram shows the IE Selenium Driver and it's different interacting components. The main connector that provides the handling between the single components, is like in the component & connector view of the Selenium IE Driver, the *CommandExecutor*.
On the other side is the java language binding. This part of Selenium was also one main part that we had to analyze. As we have seen in the component & connector view the main connector that is the same as it is in the IE driver package, was the *CommandExecutor*.
To link those two together the *HTTPServerConnector* in the IE driver (the other drivers are pretty similar to this one) takes the HTTP-requests, processes them and sends a certain code back to the language binding. To show this relationship between those two the following code snippet of the server.cc is used:

```
int Server::OnNewHttpRequest(struct mg_connection* conn) {
  mg_context* context = mg_get_context(conn);
  Server* current_server = reinterpret_cast<Server*>(mg_get_user_data(context));
  mg_request_info* request_info = mg_get_request_info(conn);
  int handler_result_code = current_server->ProcessRequest(conn, request_info);
  return handler_result_code;
}
```

At the current server the new process request received and at the next step the request is further processed and after it finishes the result_code is sent back.
The steps in between are shown as an example in the next code snippet. It is also from the server.cc class, but it shows a different part of the component. Also the session comes into play and is a part of the process chain.
server.cc:

```
std::string Server::DispatchCommand(const std::string& uri,
                                    const std::string& http_verb,
                                    const std::string& command_body) {
        …
        bool session_is_valid = session_handle->ExecuteCommand(
                serialized_command,
                &serialized_response);
        ...
        return serialized_response;
}
```

IESession.cpp:

```
bool IESession::ExecuteCommand(const std::string& serialized_command,
                               std::string* serialized_response) {
::SendMessage(this->executor_window_handle_,
              WD_SET_COMMAND,
              NULL,
              reinterpret_cast<LPARAM>(serialized_command.c_str()));
  ::PostMessage(this->executor_window_handle_,
              WD_EXEC_COMMAND,
              NULL,
              NULL);

                                int                     response_length                    =
static_cast<int>(::SendMessage(this->executor_window_handle_,
                                WD_GET_RESPONSE_LENGTH,
                                NULL,
                                NULL));
  LOG(TRACE) << "Beginning wait for response length to be not zero";
  while (response_length == 0) {
    // Sleep a short time to prevent thread starvation on single-core machines.
    ::Sleep(10);
```

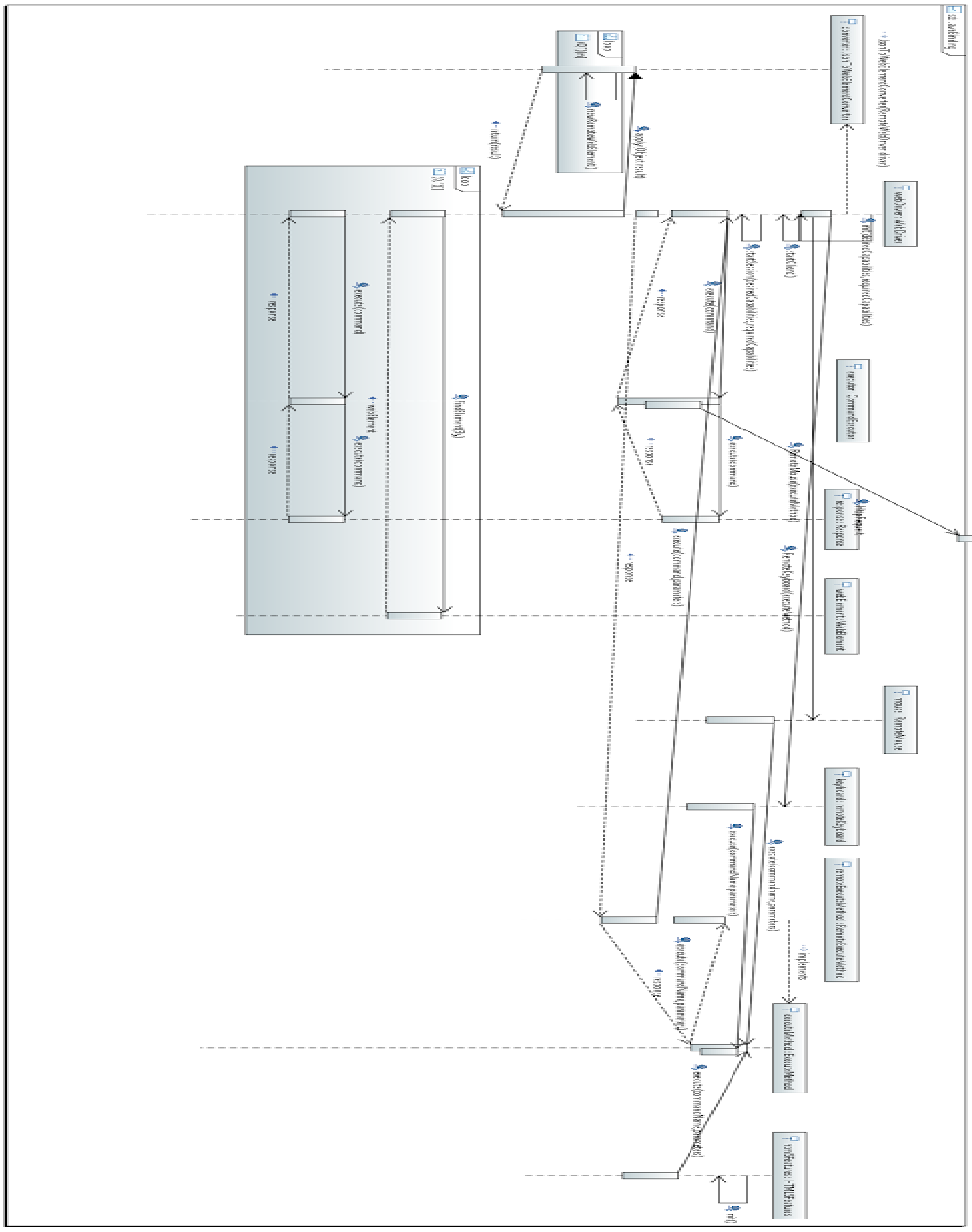
```
   response_length = static_cast<int>(::SendMessage(this->executor_window_handle_,
                                          WD_GET_RESPONSE_LENGTH,
                                          NULL,
                                          NULL));
 }
std::vector<char> response_buffer(response_length + 1);
 ::SendMessage(this->executor_window_handle_,
               WD_GET_RESPONSE,
               NULL,
               reinterpret_cast<LPARAM>(&response_buffer[0]));
 *serialized_response = &response_buffer[0];
 bool session_is_valid = ::SendMessage(this->executor_window_handle_,
                                       WD_IS_SESSION_VALID,
                                       NULL,
                                       NULL) != 0;
 return session_is_valid;
}
```

This short code snippet shows how the server knows which session is online and on which browser it is running. One thing not shown in this code is the method call to look up all sessions that are running, but it is clear how this is done within the code. There are some more classes involved in the part of executing a command after it is received, but it would be too big to outline every single class in this document.
As we said before the second part was to analyze the language bindings in the code. The classes that are most interesting here are:

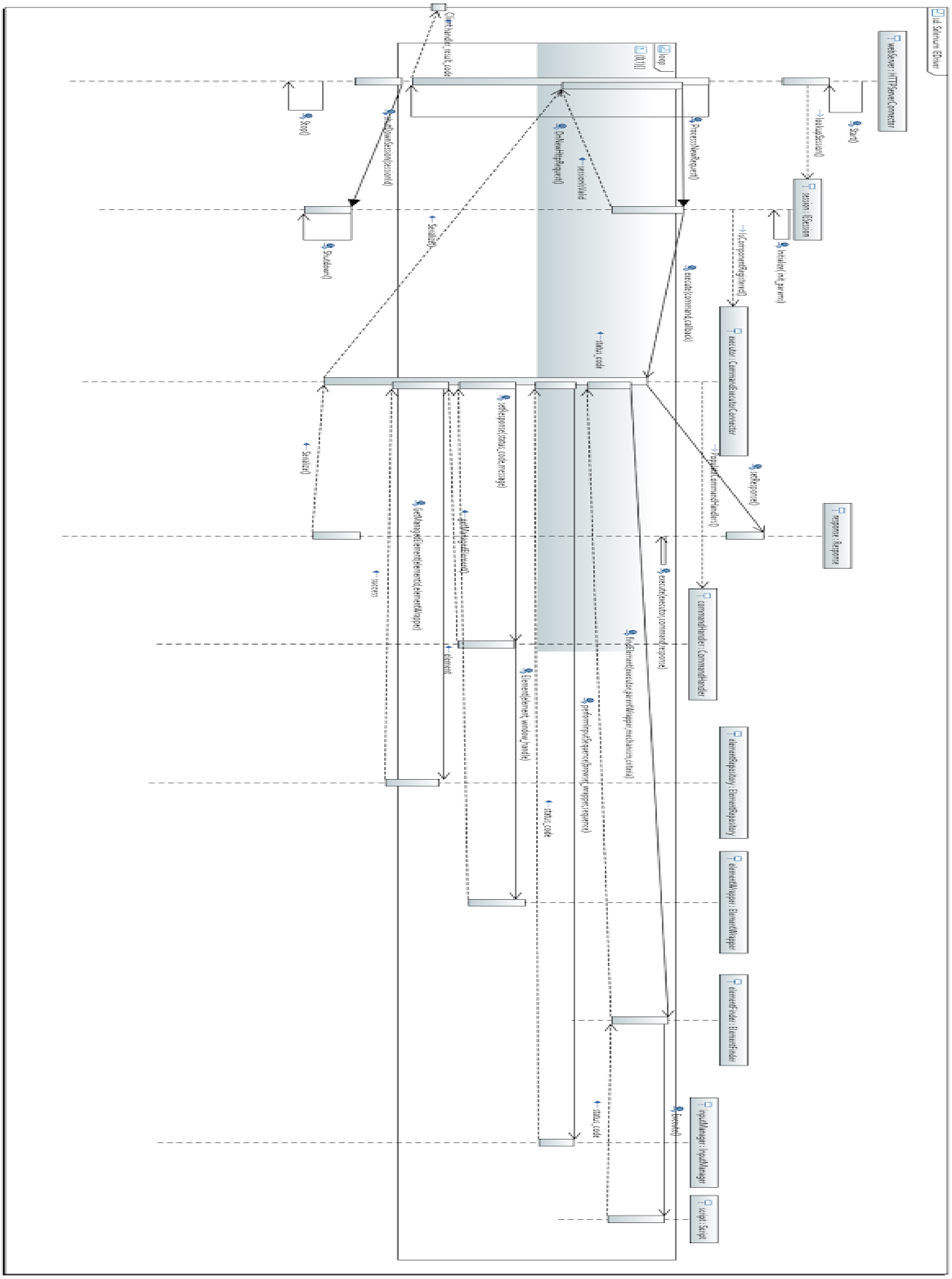
- webdriver
- commandExecutor

These two are the main actors and do the processing between the commands from the Mouse, Keyboard and many other things, and the HTTPExecutor. The webdriver class is like we said before just an interface that is used by the remoteWebDriver class. This class is then implemented by all the specific browsers and is easier to be accessed and edited. The commandExecutor is the interface between the language binding and the special drivers and communicates with both sides.

Complementary View



Java language Binding


sd Selenium IEDriver
webServer : HTTPServerConnector
session : IESession
executor : CommandExecutorConnector
response : Response
commandHandler : CommandHandler
elementRepository : ElementRepository
elementWrapper : ElementWrapper
elementFinder : ElementFinder
inputManager : InputManager
script : Script
loop
Start()
lookupSession()
ProcessNewRequest()
Initialize(init_params)
IsComponentRegistered()
execute(command,callback)
setResponse()
RequestCommandHandlers()
execute(executor,command,response)
status_code
Serialize()
Stop()
Shutdown()
ShutdownSession(sessionId)
OnNewHttpRequest()
sessionIsValid
setResponse(status_code,message)
getManagedElement(elementId,elementWrapper)
success
element
Element(element, window_handle)
status_code
Execute()

IE Selenium driver

IE Selenium driver

The approach for this view was to analyze the component & connector view and to take the code basis for checking which different methods and parts are interacting between the components. In the interaction view the same basic components are used, but there are some parts that are shown more in detail like the response.cpp class. This class is for handling the responses that are sent back and forth between the session, the executor and the handler classes. It is analyzing the different commands that are sent via the executor from the session to the handler and sets a specific response for every single command.
But the first part of the interaction view is the webserver that is called upon a new HTTP-request. Before it can receive any HTTP- requests, the server has to be initialized and running. After the request is received the server processes this request through various stages. It is processed to the session, that was looked up before by the server. If the command before is invalid or points to a wrong url, the serialized response that is sent back is put together by hand.
In the specific IESession the command is executed and is evaluated too. This is happening in the other specific sessions too like firefox session or something else. The session is initialized by the webserver when the lookup is done. The specific session has than to check whether the command executor is running and if the component is registered there. Only if this is positive, commands can be sent and executed. For this five actions has to be taken.

Sending a command consists of five actions:

1. Setting the command to be executed
2. Executing the command
3. Waiting for the response to be populated
4. Retrieving the response
5. Retrieving whether the command sent caused the session to be ready for shutdown

This part is taken from the code as a comment from a developer. It describes exactly what has to be done to reliably execute a command and get the right response for the client back.
As long as there are commands to be executed the session takes them and tries them to execute. This can result in many different outcomes. After finishing with all commands those results are sent back to the client in a serialized way. At the end of the program, a shutdown sequence is initialized by the server to shutdown the sessions, and afterwards itself.

But let’s look at the executing of one single command. A really important aspect in that case is the command executor. This class is exactly drafted for executing and handling commands and therefore is the main part in this action. It has to get the response for the right command and the right handling. This means, it has to call the command handler class and also the response class in order to produce a valuable outcome. This has to happen for every single command that is in line for execution.
Also the element that has to accessed by the command has to be found by the command executor in order to get the command at the right element. If for example a click has to be made, it has to be evaluated if the click can even be done on the element. If this part is not clickable an error code is sent back and the execution is aborted.
The command handler class is not really one simple class, but many classes that abstract the IECommandHandler class for the IE driver. This means that there is for every single command a specified command handler that has to be called by the executor. To simplify this we used one overall class to show how this is done. All in all the interaction view is a simplified version of how the real access is done and how to single modules are interacting between each other.

Java language binding

The second part of the component & connector view is about the java language binding and how this helps to develop a well maintainable system and also a useable system that is not to hard to understand, if someone would like to adapt one certain aspect.
We used mainly the same components for the complementary view, we just simplified some interactions. This helps to understand how the language binding is achieved. One exception is the response class, that is also introduced into this view.
The main component of the view, that is in addition the main component for the whole project is the webdriver, that is implemented by every single webdriver like firefox, IE or safari driver. At the beginning it has to be initialized, the special options here are the desired capabilities and the required capabilities. These two options are never the same for different browsers, so they have to be specified for every single one. Also for every single implementation the desired capabilities could be something else, so for every run a user uses Selenium for, he is enabled to change them as he likes.
After the initialization has finished the client and the session have to be started, the session also with the capabilities that were specified before. The next step is to execute the commands, that are then sent to the client and are performed. There are many options for how a single command could look like and this is shown with the view with the loop and the execute method calls.

Selenium is able to execute more or less every single input that is available on a device. This can cover a touch, a mouse click or just a keyboard button. It could also be a command that is done by a web element. For example the remoteMouse and remoteKeyboard classes were introduced, so that a user of Selenium could easily click the mouse or the keyboard.

Model Consistency Arguments

In the two views many similar components can be found. As we can see in the component & connector the view, the webdriver class is the main part that has to be called. Also in the interaction view the webdriver is the most important class that we looked into. In both views the command is passed on to the next class/component. This is done in the interaction view via method calls.
This is also done in the IE Selenium Driver, there we used the commandExecutor in both views. In the component & connector view this class is not a single class like in the interaction view, but a combined component out of different classes. In the interaction view this is split up into more than one class, this reflects with the response class and the commandExecutor.
The component & connector view was the basis of the complementary view and every change was first applied to the component & connector view and was afterwards reflected to the complementary view. This methodical implementation of changes ensures that the views stay consistent. It also helped to clarify the views, and to bring out the important components, that are also important for the complementary view.
Both the complementary view and the component & connector view use the same names for components and connectors. This approach ensures that these are used consistently and no naming confusion occurs. The only difference between the two views is, that in the interaction view some components are split up into several parts that are needed like that for the interactions. One important class that we drawed just in the interaction view is the response class. This is because we aggregated this class into the commandExecutorConnector in the component & connector view.

Found Architectural Styles

Object-oriented

When the Selenium started with the Webdriver project they decided to build an **object-oriented API** for their users. This object-oriented API is of course a big benefit for the users and enhanced the usability of the project as explained in the previous chapters. Before the Webdriver project was released the users had to use a dictionary-based API where you have to split up your command always into three parts. For instance if you wanted to type “something” into the query field “search” you had to send the command “type name=search something” over the API.
Not only in the language bindings an object-oriented style is used also all the driver implementations are using this style. This is not a big surprise because most of the big modern software projects are using this style to reach their quality goals.

REST

Another very important decision was the to choose REST with JSON as communication style between the language bindings and the various driver implementations. There were also several **other possibilities** which could have been used as communication channel but all of them had some serious caveats. For instance for a communication over raw sockets there wouldn’t have been many libraries available to sustain a custom protocol. Additionally this approach would have restricted the communication to be line-based and therefore sending images such as screenshot would have brought bigger problems.
To don’t have these restrictions Selenium team choose HTTP as transport mechanism for their protocol. In this next iteration of the select process they had again some possibilities to choose from. A possible protocol could have been SOAP, which would had exposed a single end-point for communication. The original protocol of Selenium used such a single end-point communication to receive the column-based query strings which also worked pretty well. Nevertheless the Selenium team ruled SOAP out, it didn’t felt right to them and they had the vision of being able to connect to a remote WebDriver instance in a browser to view the state of the server. Tu fulfill these requirements they choose REST with JSON but they call their approach **“REST-ish”** because they are breaking several rules of a true REST communication. For instance in their communication it makes so far no sense to cache requests and they communication between the language bindings and webdriver implementations is not stateless.

Overview of information sources

https://code.google.com/p/selenium/
https://code.google.com/p/selenium/wiki
https://code.google.com/p/selenium/source/checkout
https://github.com/SeleniumHQ/selenium
http://www.seleniumhq.org/
http://www.praveenjana.com/selenium-webdriver-architecture/
http://www.guru99.com/introduction-webdriver-comparison-selenium-rc.html
http://docs.seleniumhq.org/docs/03_webdriver.jsp
http://en.wikipedia.org/wiki/Google_Closure_Tools
https://saucelabs.com/

videos:
https://www.youtube.com/watch?v=MwUHEdAL1Ts
https://www.youtube.com/watch?v=lfIzRHNXQhM
https://www.youtube.com/watch?v=IheoX9bgwNg

groups/forums:
https://groups.google.com/forum/#!forum/selenium-users
https://groups.google.com/forum/#!forum/selenium-developers

Software Architecture

Lab 1
Selenium

Part 2 - What did we learn?

Modelling

For all of us it was the first try to model such a big software project. We spend many hours with just reading and digging around in the code base. One thing which it made very complex was of course the many different languages used throughout the project. In the whole project there were used over six different languages and of course nobody of us knew all of them well. This was then also one of our main problems because if you know a language just superficial you have to spend even more time to just understand maybe some easy code snippet. Additionally it was complex for us to figure out which code belongs to which part of the project. Selenium uses one single repository where all different drivers, language bindings and even the Selenium RC are stored.
After this hard start we somehow understood how all the pieces worked together. At this point we decided to focus on the Java language binding and on the Internet Explorer driver implementation and started to draw a class diagram of these two system components. For this task the tool support was very good and all of us knew the class diagram from other courses so we hadn't had big difficulties to carry out this task.
After we had drawn a class diagram we tried to find the right level of abstraction. For us this task was not very easy. The biggest problem there was that we never had drawn a component & connector view before. Therefore we had the problem to assign the classes to the right components and to find a good level of abstraction. For instance at the language binding we introduced the *CommandExecutorConnector*. But as we understood the definition of first class connectors from the lecture a huge percentage of the classes were part of this connector and basically there weren't many classes left to form other components or connectors. After some discussion we insisted on this huge connector in the language binding because in our opinion a language binding is nothing else than a big connector which allows the user to connect to some system, in our case the different webdrivers.
Similar problems we encountered when we tried to model the driver implementation for the Internet Explorer. There the problem was more about to assign the classes to the right components. For instance we never exactly knew if we should form a new component or if the class belongs to an already existing component.

Correctness

As it is our first component & connector view and also the first time that we tried to model something we don't have written on our own it is very difficult to say how good the outcome is. As said before we spent many hours just reading the code line by line and at the beginning we were very overtaxed with the big repository.

At the point where we decided to start drawing a normal class diagram we actually started to do something and additionally it helped us a lot to understand the application more deeply. As we already mentioned we based then our component & connector view mainly on these two class diagrams. Because of that fact we think that the created model is to a great extent correct. Of course we have to mention that that we only modelled the overall picture of the application because we focused on one language binding and one driver implementation Therefore are maybe some important and interesting driver-specific implementation details missing.
Another point which gives us confidence that the model is correct is that we had some contact with one of the contributors. He gave us additional information to the project and feedback to our current solution which clarified many uncertainty about the project and therefore gave us a better understanding of the code base.

Chapter in the book “The Architecture of Open Source Applications“

We started our analysis of the software project by reading the corresponding chapter in the book “The Architecture of Open Source Applications”. It was a good starting point for us but let also many questions open. Basically already there we had a big problem to actually understand which component is running where. Some parts of the chapter were also not important for this report such as the history part and sections about the Selenium RC. All in all the whole chapter mixes the names Selenium, Selenium RC, Selenium Core, Selenese and Webdriver very often and so if you read it the first time it is very confusing.
Reading through the chapter again at some later point or after the whole analysis many parts of the chapter make more sense and actually some chapters were also very valuable to carry out this report.
We also didn’t found any differences or contradictions between the information we gathered from the chapter in the book and the model we derived from the various other artifacts. Of course we could have overlooked something but in general we can see that the chapter from the book “The Architecture of Open Source Applications” was very valuable and aligns with the architecture derived from the artifacts.

Groupwork

At the beginning all of us read many general articles and tutorials about Selenium. We shared the articles and tried to have the same knowledge level. Afterwards we started to dive into the code. There we splitted us up into two teams where one team tried to understand the various language bindings and the other team the driver implementations. The teams were formed as following:

- Simon and Valdemar: Java language binding
- Thomas, Christoph and Balint: Internet Explorer driver

After every session we tried again to keep us up to date and to share the knowledge. This pair-reading approached worked very well because it was very important for us to discuss during reading code. Most of the time also one of our group members was more experienced as the other ones and therefore helped us to keep the speed. Nevertheless the reading part was a very time consuming task.
After both of the groups were comfortable with their respective code base each group started with the component & connector view. Additionally the group with language binding started to think about the which complementary view we should choose.
During all this tasks we always extended and discussed together the qualities and tactics of the project. Therefore we could start at same time to elaborate this part of the report.
At the end we could work very individually and we were able to save some time because group work wasn't needed anymore. To still have a report with a high quality we gave us mutual feedback and tried to be very critic.

Communication with the developers/maintainers

Selenium has organized their project on google code and they use mainly a google forum to discuss topics around Selenium. After we have spent some hours for our report we decided to post our current solutions in the developer forum and asked for feedback, information and insights. After some days we were very happy that one of the contributor of the project replied us very detailed. He extended the list of applied tactics and design decisions to improve the qualities usability and modifiability. Besides the new tactics and design decision the reply was for us very helpful to get a deeper understanding of the application and the purpose of some design decisions.
So far we don't know how good the outcome of our work is but we have planned to share the report with the Selenium team. We don't know if they are using some architectural models to onboard new developers or to discuss about the current architecture and of course we would be very happy if they can reuse something from this report.

Lecture

One of our main problems was regarding the first class connectors. We didn't knew if we should or could model components and connectors inside of first class connector

and if yes how. Maybe this is forbidden in a normal component & connector view but in the case of the *CommandExecutorConnector* the first class connector already wrap many classes and so there weren't many classes left to form new connectors and component.
In addition it wasn't easy for us to find other first class connectors besides the *CommandExecutorConnector* and the *HTTPServerConnector.* Maybe some more code example during the lecture which are showing connectors would help to tackle this problem.
Another problem we had was actually the modelling part. We didn't found many good resources about the semantics and syntax of the component & connector view with sysUML and all definitions we found were very long and detailed. Therefore we choose the approach to be as consistent as possible with the models discussed during the lecture. This of course is not a very scientific way of doing things and here maybe some good resources would help and clarify a lot. Additionally it would be useful to extend this part in the lecture and to provide more slides on this topic.
To further improve the elaboration of the reports it should be considered to introduce a feedback session some weeks before the deadline. For this feedback session the students can prepare questions and the professor could review the current models. We think this would be very valuable and improve the outcome and therefore also provide better learnings for the students.
What we think helped us was the lecture about the qualities and the tactics. The slides have a very concise and clear structure and gave a perfect overview over the topic. During the report we easily identified some qualities of the project and also found many tactics which are trying to enhance these qualities.