Architecture Design

1. Introduc(k)tion

This document contains the basic architecture design of the Chrome extension that has been developed during the context project for ‘Tools for Software Engineering’. The architecture of the system is explained in terms of different views that can be used to look at it (e.g. ‘Subsystem decomposition’). In addition to this explanation, a UML-diagram is included in order to visualise the explained concepts and the relations between them.
The goal of this document is to give the reader a general overview / idea of what the architecture looks like (internally) and to provide arguments to the design choices that have been made.
At the end of the document a ‘Glossary’ is provided for the technical jargon that has been used throughout this document. These words can be recognised by a ‘*’.

1.1 Design Goals

Throughout the development of the architecture of the system, the designers tried to interleave software design principles as much as possible with their ideas on creating the architecture. Therefore, design goals were made and established and these are the following:

- <u>Availability</u>
  Availability is important from the moment that the product has been delivered, as the Chrome extension must be able to start whenever the user wants so and data has to be retrieved whenever the extension is supposed to, in order to give the user tips on how to improve his/her pull request reviewing behaviour.
  This design goal has been achieved by creating a stable Chrome extension. The extension will always be available.when the extension is activated. Its only dependency is the database to which it is supposed to log, as it sends data to it, but this is an external dependency.

- <u>Configurability</u> (or <u>Flexibility</u>)
  The system has to take user preferences into account with respect to, for example, privacy and decision-making in when to turn the extension on or off. Because of this, the system has to be configurable (and flexible); the behaviour of the system should be able to be adapted easily, based on the user’s preferences. By creating a flexible structure within the architecture, this goal can be achieved.
  And the extension does have a flexible structure. In the front end, there is an Options page in which the user can indicate his/her preferences. The front end is linked to the backend and whenever an option has been switched, the backend will be notified and will take this into account.

- <u>Extensibility</u>
  The system can easily be extended, especially for use on other websites. We have created such a structure that it’s just a matter of reconfiguring the semantic elements and it all works.

- Manageability
  The Chrome extension has a lot of different kinds of data that is put into the system. From this data, new information is deduced. Therefore all these different types of data and information that are put into the system and are deduced should be manageable. Within the architecture, there should be a way of managing it in a ‘smart’ way.
  The extension does this by separating the different kinds of data and sending these with an ID number[1] to a database. Because of the distinction and a well-designed architecture (which will be explained in *Software Architecture Views*) on how to deal with the data, data management is nicely organised.

- Performance
  Because of the fact that a lot of data is retrieved and deduced, the system can take a lot of memory within the browser. Therefore, performance has to be taken into account within the architecture, in order to soften the amount of possible latency that will be caused.
  The extension does this by a large set of efficient and small functions. Besides, the front end and backend interact easily, so there is no delay caused there. Also by using a database adapter, the extension interacts smoothly with any database, so performance delay is minimised here as well.

- Reliability
  The system works with a database in which data is stored. It has to retrieve data from it and therefore a connection is needed between the two. Besides, front end and backend have to be connected in some way. The extension uses the Chrome storage* for this. User preferences are stored and loaded from it. Therefore, the extension is as reliable as the Chrome storage.

- Scalability
  The target group of the product is relatively quite a large group; in fact, it could be thousands to millions of users. So the product should be able to be distributed over all of these users, without affecting the quality of service of the extension (or at least as little as possible). By taking this fact into account within the architecture, on the server side extra resources are needed. The extension takes this into account by giving IDs to users and communicating this to the database. Also, the database uses a RESTful API*, which also helps on scalability. Therefore, the extension is scalable and scalability is only influenced by the server side.

- Privacy
  A lot of data flows through the system, from input to output. This data is in general quite sensitive for the user, as his full (or partially, depending on the preferences) behaviour is being tracked and saved to a database while reviewing a pull request.
  The extension achieves this by providing privacy options, which allow the user to disable the logging of specific data.

[1] For a full overview of the semantic database elements, check the following document: https://github.com/thervh70/ContextProject_RDD/blob/master/doc/SemanticDatabaseElements.pdf

2. Software Architecture Views

This section describes the architecture of the system by explaining several views on how the system works. First of all the system is decomposed into smaller 'subsystems' and the relations between these subsystems are explained. These subsystems and relations are visualised by a UML-diagram. Secondly, the mapping from hardware to software and vice versa is explained. Thirdly the persistency of the data management is explained and finally concurrency within the system will be explained.

2.1 Subsystem Decomposition

Before going into the details, a general overview will be provided in terms of a UML-diagram. Then each of the subsystems and their relations will be described.

UML-diagram
The overall idea can be generalised by the following UML-diagram:
https://drive.google.com/open?id=0B2Oj2T9nWhD7XzVlY3NlYXFRRzQ

The subsystems and their dependencies
As shown in the UML-diagram, there are seven subsystems within the system: the '*MainController*', '*Status*', '*DatabaseAdapter*', the '*ContentController*', '*SelectionBehaviour*', '*EventBinding*' and '*Options*'.

Let's start with explaining why there are two Controllers. Certain actions can only be performed in either the content script of the extension or the background page of the extension.
The MainController is active in the background page of the extension, which is always accessible.
The ContentController is active in the content script, which has access to the DOM* and is only loaded when a Pull Request is open.

MainController
The MainController has to make sure that the ContentController is activated on pages where a Pull Request is open. To this end the Chrome API* is called and for all tabs (including tabs that are later created by the user) it is checked whether they contain a valid Pull Request URL by a utility class URLHandler. In that case, a message will be sent back to the ContentController, saying that it should hook event handlers to the DOM.
When the MainController gets a message from the content script with data that should be logged, it will log this to its current DatabaseAdapter (see next page).
The MainController will set the Status (see ahead) according to the current state.

Status
The Status class is a singleton* that can be used to propagate the current state to the rest of the system (both front and backend). Whenever the status is set to a new *StatusCode* (which is an enumeration of "running", "standby", "error", "off"), this class also informs other parts of the system that the status has been changed, so that the browserAction icon and the popup can be updated.

The Status class has a dependency on the Chrome API. It sends a message to the browserAction icon and the popup whenever the status is changed.
Due to its Singleton* design the Status class will not be replicated, this means the whole system can only have a single Status in total. It would be wrong if the system had 2 Statuses. This can be solved with the design pattern we are using.

DatabaseAdapter
The extension has to track data, which is then stored in a database. In order to do so, an adapter has been designed so that the extension can interact with the database, independent of the interface that is used on the database side. By doing this, the 'Adapter' design pattern* is applied. There are currently two DatabaseAdapters. In the final version, we will use the Adapter that logs to the global database (*RESTApiDatabaseAdapter*), but for debugging we use an adapter that just logs the data to the Chrome console (*ConsoleLogDatabaseAdapter*).
The RESTApiDatabaseAdapter thus has a dependency to the database. The rest of the system then uses the DatabaseAdapter, because now there is an interface that is expected within the rest of the system. Because of this, the rest of the system can send data from the database relatively easily.
The RESTApiDatabaseAdapter also has a dependency to a utility class URLHandler. This class helps with formatting URLs and fetching things like repository name and pull request number from URLs.

ContentController
The Controller has a matrix of which ElementSelectionBehaviours (selects HTML elements) and which ElementEventBindings (defines the type of event) are matched. Not all Elements and Events should be matched. For example, scrolling over the "merge"-button has no use. This filtering is done by the DoNotShowOptions class (see subsystem Options).
The ContentController provides the ElementEventBindings with a DatabaseAdapter, but not any of the ones mentioned before. This MessageSendDatabaseAdapter sends a message to the background page so that it can be posted to the real database. We stepped away from directly logging to the database because of security issues.

ElementEventBinding
There are a few different ElementEventBindings, all with a different type of event (EventID). For example, there is one for click events, mouse events, key press events, etc. Every ElementEventBinding is instantiated by the ElementEventBindingFactory, which generates GenericElementEventBindings. These contain ElementEventBindingData objects that specify the different properties of the GenericElementEventBinding. The ContentController is the main user of this Factory, which creates for every ElementSelectionBehaviour exactly one ElementEventBinding.

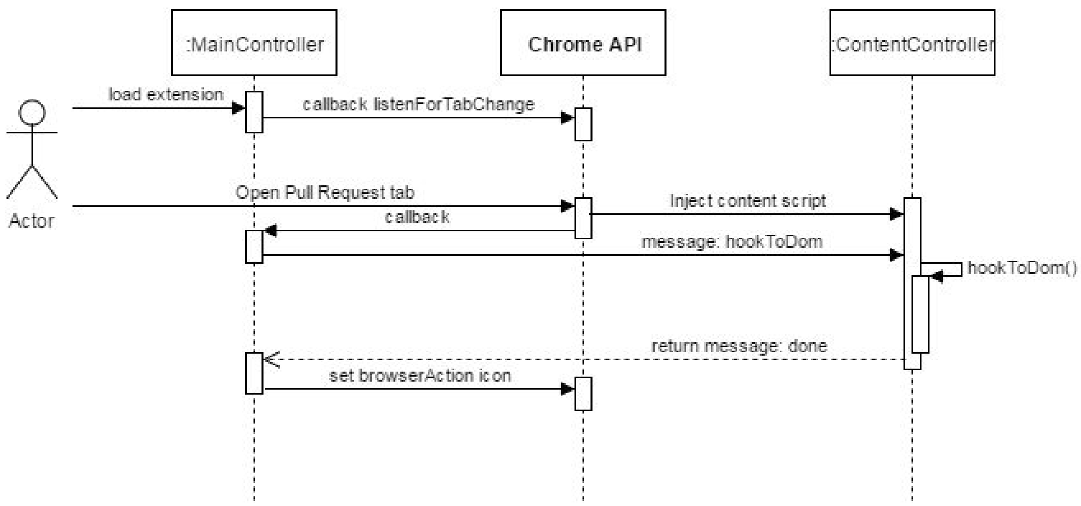
ElementSelectionBehaviour
There are a lot of a lot of different ElementSelectionBehaviours, each for a different type of element (ElementID). Just as with the ElementEventBindings, the ElementSelectionBehaviours are created by a ElementSelectionBehaviourFactory. These create GenericElementSelectionBehaviours, which contain different ElementSelectionBehaviourData objects.The ContentController is the main user of this Factory, which creates every ElementSelectionBehaviour exactly once.

EventTracker
To be able to log different raw data events, EventTrackers are created. Every type of raw data event has its own Tracker. This Tracker logs an Event to the database. This Event implements its corresponding EventObject interface and the Tracker uses the proper DatabaseAdaptable (which uses the Strategy Pattern) to log this Event to the database.

Options
The user should be able to indicate its preferences and these, of course, have to be taken into account. Therefore the subsystem ‘Options’ is designed so that preferences data can be derived. The ContentController uses the Options subsystem so that it can make decisions based on it. These decisions are about mouse tracking, keyboard tracking and other data.
User preferences are saved and loaded in the browser via the Chrome storage*. In the singleton Options class, these preferences are retrieved from the storage. Because of this, no extra save file, database or something similar is needed.
We are using the Singleton* pattern to assure that there is only a single, globally accessible version of the Options class. This way we can know that there will be only a single variant of the options in the backend and that one closely listens to the Chrome storage.
The Options class also makes sure that the default options can be set back by the user. The default options are also stored in a part of the Chrome storage so that the front end can easily reload them.
The OptionsObserver is used to inform the rest of the backend that there was a change after a user submitted new preferences. Classes such as Status will be linked to the Options class via this Observer pattern* in order to know when Octopeer is turned off. We use this Observer pattern for extendibility. When another class needs to know about any change in options, it can implement OptionsObservable and added to the OptionsObserver
There is also a class DoNotWatchOptions in this subsystem. This class decides which elements-event combinations should not be tracked. This is based on some heuristics (e.g. scrolling over a merge button has no use) and on the Options set by the user.

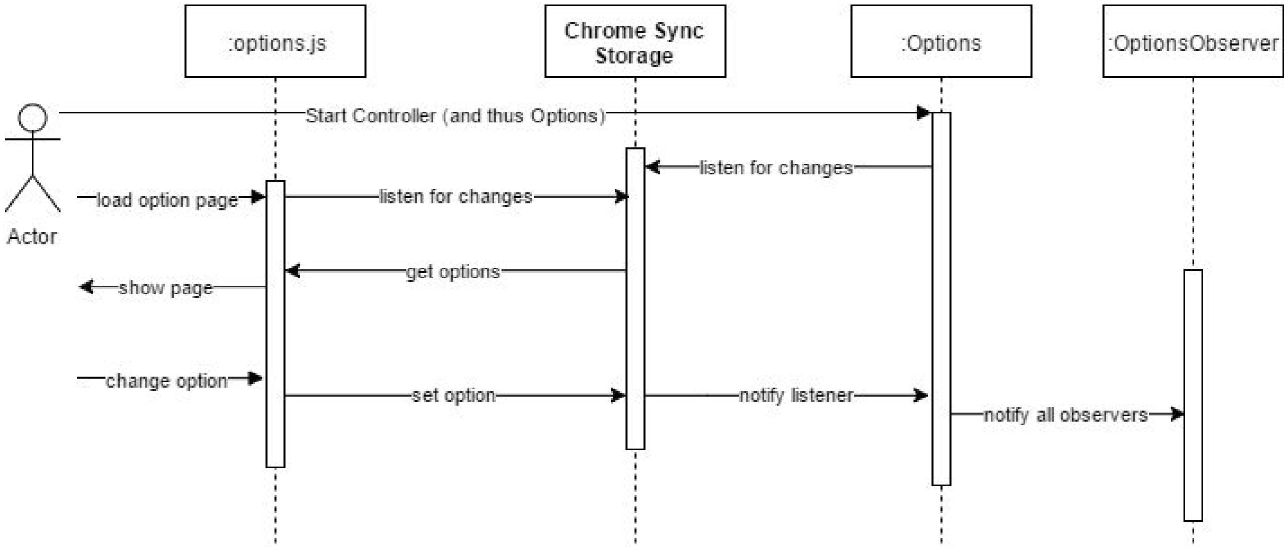
2.1.1 Sequence diagram for MainController and ContentController

Below you find a Sequence diagram of how the MainController and the ContentController interact when the user loads the extension.



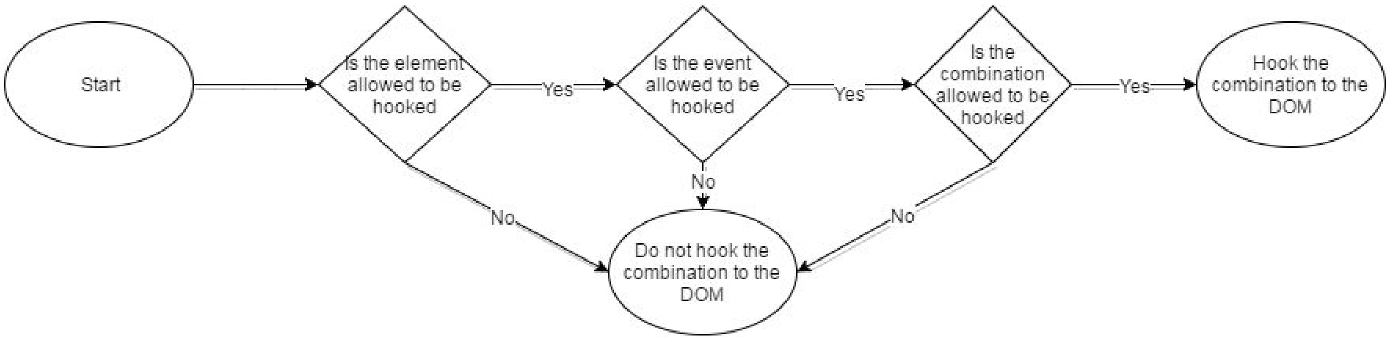
2.1.2 Sequence diagram for Options (front and back end)

Below you find a Sequence diagram of how the Options are used in both the front and the backend. It is useful to mention that the MainController is the most important OptionsObserver, making sure that all open pull request tabs will re-initialize when options have been changed.



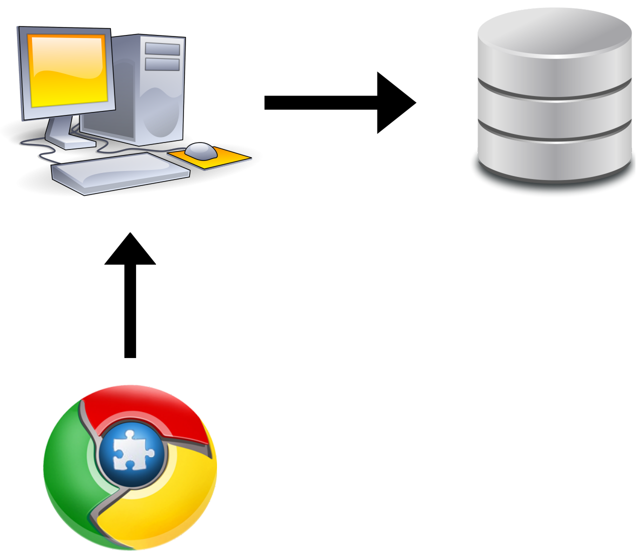
2.1.3 Flowchart Element-Event Combination (Pre) Binding

Below you can find a Flowchart in which the process of element-event combination (pre) binding is visualised. It shows the process of hooking a combination to the DOM.



2.2 Hardware/Software Mapping

The hardware/software mapping within the system is fairly simple, as there are only two factors that interact with each other in this relation. There is a database and a computer on which the Chrome extension is active. This interaction can be summarised with the following sketch:



The Chrome extension runs on the computer and uses data that is send from the database.
The extension runs within the browser Google Chrome on Windows (7 or higher), Mac OS X (10.8 and higher) and Linux. It requires a reasonable connection speed; at least 1 MB/s down- and upload speed. Besides, we require the extension to use a maximum of 100MB of free RAM, when active.

2.3 Persistent Data Management

The data that is used throughout the program can be split into two classes: the user preferences data and the 'raw' data that is stored in the database.
- User preferences data is stored within the Local Storage and this data can be retrieved by the system with the *Options* subsystem.
- 'Raw' data that is created when a user is reviewing pull requests is stored within a database that runs on a server and this data can be retrieved by a RESTful API*.

Chrome also has an 'Incognito mode', in which stored browsing data is reduced to a minimum. Incognito mode by default disables the extensions that are installed in the browser, so our product is, in this mode, with the default options, not able to collect any data or function. Chrome by default disables the extensions that are installed in the browser in incognito mode, so our product is, in this mode, with the default options, not able to collect any data.

2.4 Concurrency

In general, there is little concurrency within the system, as data is retrieved from the DOM tree* and added to it. Therefore only the DOM tree is the ‘main resource’ of the system.
Besides, for the data within the system, because we’re using a RESTful API, the asynchronous callbacks that are performed are independent of each other and therefore shouldn’t cause any concurrent processes.

3. Glossary

In this section, the technical terms that are used within this document are explained. The technical terms themselves have been sorted alphabetically within the table below.

Technical Term	Description
Adapter Pattern	A design pattern that is used throughout software engineering in order to make interfaces from two different systems compatible with each other.
API	Application Programming Interface, a term that indicates that an interface for which a set of definitions are defined so that they can be used for communication between systems.
Chrome storage	An API provided by Google for saving data of Chrome extensions in the browser. The storage can be used to store, retrieve and track changes to user data.
DOM tree	Document Object Model tree, in which data is structured in terms of HTML objects.
Observer Pattern	A design pattern that is used throughout software engineering so that the object that changes state does not have to know all other objects that want to know about a state change while writing the code. Objects that want to “observe” this object, can register as an observer. The observed object then only has to iterate over its list of observers and call the callback functions that were provided.
RESTful API	An API that adheres to the principles of REST does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service.
Singleton Pattern	A design pattern that ensures a class has only one instance and provides a global point of access to it.
Strategy Pattern	A design pattern that is used throughout software engineering. It defines a family of algorithms, encapsulates each one and makes them interchangeable.