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 visualize 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:

- Availability
  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 behavior.

- Configurability (or Flexibility)
  The system has to take user preferences into account with respect to, for example, privacy, security and decision making in when to turn the extension on or off. Because of this, the system has to be configurable (and flexible); the behavior of the system should be 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.

- 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 and producing useful tips with respect to pull request behavior.

- 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.

- 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. This connection should be reliable, so some kind of an ‘adapter’ should be needed for providing this reliable connection.

- 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 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

- Securability
  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) behavior is being traced and saved to a database while reviewing a pull request. Therefore securability has to be taken into account within the architecture so that the data that is being used is safe and can’t be retrieved by outsiders.

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 visualized 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 generalized by the following UML-diagram:
https://drive.google.com/open?id=0B2Oj2T9nWhD7aTZ0SXJQVFgyWlE

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 not-yet-existing tabs) it is checked whether they contain a valid Pull Request URL. 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. 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 (for example) the browserAction icon can be updated.
The Status class has no dependencies on itself because we implemented the informing of the status via the Observer design pattern*. (is done in sprint 4)

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.

ContentController
The Controller maintains 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 matrix should also be processed by the Options class, because the user should be able to disable certain types of logging.
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.

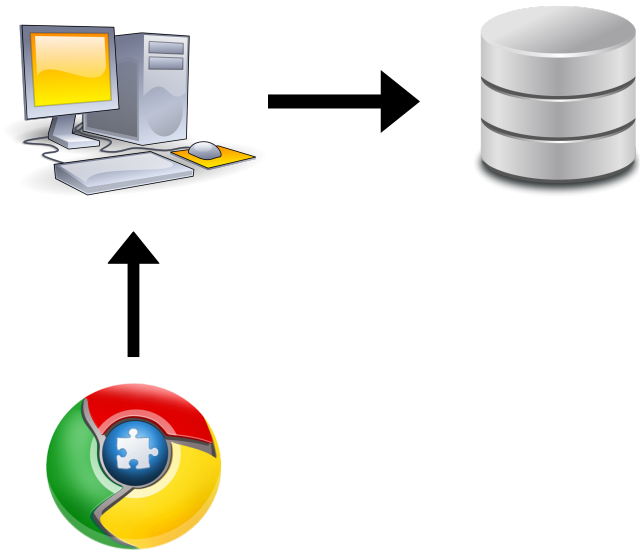
ElementEventBinding
There are a few different ElementEventBindings, all with a different type of event (EventID). For example, there is one for click events, mouse events, key press events, etc. They use ElementSelectionBehaviours to determine which elements to bind the event handlers to. Also, they make sure each event is only attached to the DOM once.

ElementSelectionBehavior
There are a lot of different ElementSelectionBehaviours, each for a different type of element (ElementID). They have no dependencies except for jQuery, as they have to return a jQuery object containing the HTML elements that should be tracked.

Settings → 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 Options so that it can make decisions based on it. These decisions are about privacy (what to track) and security (what data to obfuscate).
User preferences are saved and loaded in the browser via the chrome storage*. In the Options class, these preferences are retrieved from the storage. Because of this, no extra save file, database or something similar is needed.

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 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, the extension requires to use around 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. 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.

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 are 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.
Strategy Pattern	A design pattern that is used throughout software engineering and defines a family of algorithms, encapsulates each one, and makes them interchangeable.