Architecture Design

1. Introduc(k)tion

This document *contains* the basic architecture design of the chrome extension 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'), including a UML-diagram in order to make the explained concepts and the architecture itself concrete.
The *goal* of this document is to give the reader a general overview / idea of what the looks like (internally) and to provide arguments to the design choices that have been made.
At the end of the document a *'Glossary'* has been provided so that the (difficult) technical jargon that has been used throughout this document, for explaining technical concepts, can be looked up by the reader. 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
  The system has been build in such a way that each week there is a working version of the product, so that it can be shown to the product owner in terms of a demo. By doing this, feedback can be retrieved in an early stage. Therefore availability of the system throughout the development is an important goal.
  Besides, availability is also 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 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=0B2Oj2T9nWhD7YkxtTGg5Q1QydU0

The subsystems and their dependencies

As shown in the UML-diagram, there are four subsystems within the system: the ‘*DatabaseAdapter*’, the ‘*Tracker*’, the *´Controller´* and *´Settings´*.

DatabaseAdapter

The extension has to retrieve data that is 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.
The DatabaseAdapter 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.

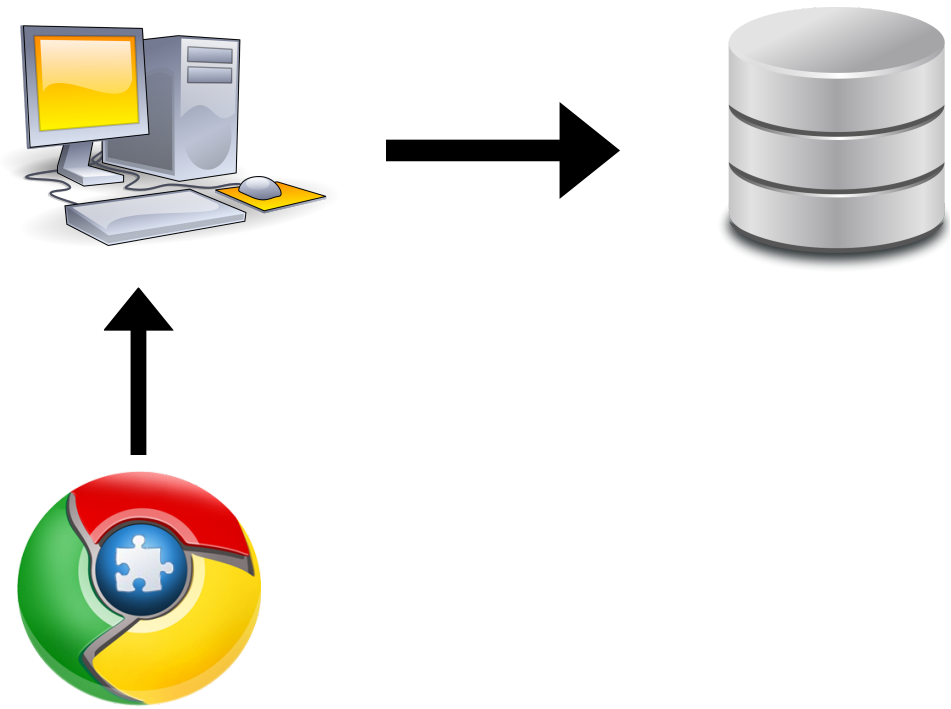
Tracker

As seen in the previous subsystem, data can be send. The data of course also has to actually be gathered. This is done by the Tracker. Because there are a lot of different types of data, the Tracker needs to have a lot of different possible behaviors. In order to facilitate this, the ‘Strategy’ design pattern* is applied; the different kinds of behavior are interchangeable by creating new Tracker instances with the desired behavior. Because of this, we favor composition over behavior, as it’s more easy, flexible and maintainable.
The Tracker thus has a dependency to the DatabaseAdapter, as that’s the source where data is send to. Besides, it has a dependency to its two encapsulated behavior types: selection and element event binding. The Tracker thus uses these in order to get a behavior that decides what elements it will get and what to add event handlers to. The other subsystems, that have not yet been mentioned, will use the Tracker.

Controller
As seen in the previous subsystem, different types of data can be tracked. Because there are a lot of different types of data that can be tracked, there are a lot of Trackers and each of these trackers has to be controlled. This is done by the Controller. The Controller maintains a list of all Trackers that are used within the extension. By starting the controller, the system is started and Trackers are created. The Controller is the AI* of the system and therefore makes all logical decisions, thus hints for improving pull request review behavior are derived here.
The Controller thus has a dependency to the Tracker, as it uses Trackers. The Controller also uses the last subsystem of the four in order to take user preferences into account, as the settings are derived there.

Settings
The user should be able to indicate its preferences and these, of course, have to be taken into account. Therefore the subsystem ‘Settings’ is designed, so that preferences data can be derived.
The Controller uses Settings so that it can make decisions based on it. These decisions vary from when to create what Tracker and when to derive what conclusion.

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:



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 Webstorage and this data can be retrieved by the system with the *Settings* 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*.

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 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.

Adapter Pattern with	- A design pattern that is used throughout software engineering in order to make interfaces from two different systems compatible each other.
AI	- Artificial Intelligence, a term that indicates that a computer can make reasonable decisions, from a human perspective.
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.
DOM tree	- Document Object Model tree, in which data is structured in terms of HTML tags.
Strategy Pattern them	- A design pattern that is used throughout software engineering and defines a family of algorithms, encapsulates each one, and makes interchangeable.