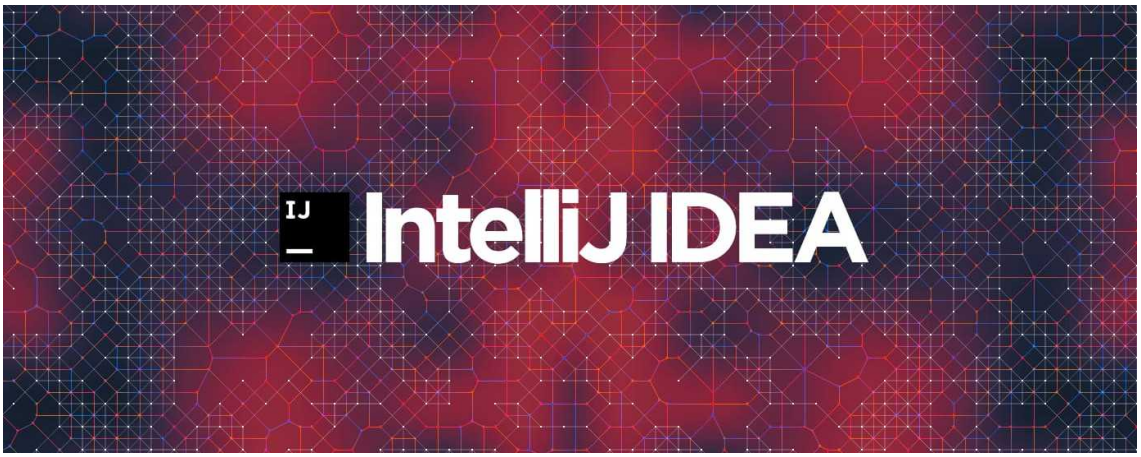
Tracking Changes in Links to Code

The Development of an IntelliJ plugin



Authors:

Ceren Ugurlu (4851609)
Irem Ugurlu (4851625)
Tommaso Brandirali (4792890)
Tudor Popovici (4812379)

Date: 21 June 2020
Course: Q4 Software Project (CSE2000)
Project Coach: Prof. dr. Arie van Deursen
Teaching Assistant: Cédric Willekens
Client: Vladimir Kovalenko representing JetBrains

TU Delft
Department of Computer Science

JET BRAINS

Preface

This report is written by a group of four computer science students at Delft University of Technology. With this report, we conclude the final quarter software project, a second-year course which is part of the Bachelor program of Computer Science and Engineering. The project was completed from 20 April to 28 June 2020 for the company JetBrains. As the final result of the project, an IntelliJ IDEA plugin which fixes the problem of broken links to code in Markdown files was created.

While writing this report we assumed the reader to have basic knowledge of the IntelliJ IDEA development environment. Additionally, readers are also assumed to have a basic understanding of how the version control system Git works.

Readers particularly interested in the solution approach to the problem of obsolete links to code can find all relevant information in Chapter 2. Readers more interested in the list of features of the plugin included in the final release can find this in Chapter 6.

We would like to thank lots of people that supported us over the course of this project. First of all, we would like to thank JetBrains for the giving us the opportunity to work on this interesting and challenging project. Furthermore, we would like to thank Vladimir Kovalenko, our primary contact at JetBrains, who supported us with useful feedback and made sure that the project is continued in the right direction. Secondly, we would like to show appreciation for the guidance and the feedback received from our coach Prof. dr. Arie van Deursen. Finally, we would like to thank Cédric Willekens, our teaching assistant, for his help and advice.

Delft, 21 June 2020
Irem Ugurlu
Ceren Ugurlu
Tudor Popovici
Tommaso Brandirali

Summary

A common problem in team software development is that of out of date links within documentation files. Documentation is needed to share knowledge about the inner functioning of different modules between developers and teams. Such documentation often contains links to resources both within and outside the source code of the software. These links orient the reader towards the parts of code that are being mentioned in the documentation. These links, however, can easily become obsolete when the resources they refer to change, and maintaining them up to date is a necessary, although non productive and time consuming task.

IntelliJ IDEA is an open source IDE, developed and maintained by JetBrains. It is mainly oriented at languages running in the Java Virtual Machine, such as Java, Scala, and Kotlin. The IntelliJ IDEA source code is the base for a family of IDEs adapted for specific languages and environments, most of which are maintained by JetBrains as paid products. This family is commonly referred to as the IntelliJ Platform. IntelliJ IDEA and all of its descendant IDEs support the on-demand installation of plugins to extend the features of the IDE.

This report aims to describe the research, design, implementation and uses of a proof of concept plugin for IntelliJ IDEA. The plugin is meant to solve the aforementioned problem of out of date links, by automating and simplifying the aforementioned task of maintaining links within documentation up to date. In practice, this was achieved by scanning supported files to find such links, tracking them through the version control history of the project, detecting when the resources linked have been modified, and suggesting replacement links to the user based on the new location of the resources.

The design stage of the product's development started from a discussion with the client, JetBrains, about their vision for the product's functionality and intended use. This was followed by a literature research and study, focused on how to solve specific technical challenges related to tracking the same resources throughout the evolution of a software project, especially in the case of single code lines and groups of lines. The research's findings were analyzed for feasibility within the project's scope and resources, and found to be feasible. A project plan was then drafted, detailing the research's findings and defining the requirements for the final product. The development stage then proceeded following the Agile Scrum planning methodology, with weekly meetings and assignment of tasks to the members of the team.

Research was done on the ethical implications of the product, according to the guidelines of Value Sensitive Design. The main stakeholders were found to be first and foremost the client (JetBrains), then the future users of the plugin, and finally the readers of documentation produced with the help of the plugin. The main ethical issues related to the product's development and use were found to be: code quality and readability, automating decisions for the user, and possible privacy risks due to the handling of some login credentials for remote repositories. Design choices were made in order to mitigate these issues.

The final product successfully integrates all of the necessary requirements that had been defined in the initial project plan. It can correctly parse various types of links from all `.md` files within the currently open project and provide automatic maintenance for the identified relative links, as well as links targeting web-hosted repositories that correspond to the open project.

The client had initially expressed an interest in receiving a product that would be portable to environments different from the IntelliJ IDE. While it was not possible to modularize the code of the plugin in order to enable easy portability of the source code to other environments due to shortness of time, during development a considerable effort was made to separate the logic into modular components with well defined interfaces. Furthermore, while it

was not possible to perform unit testing extensively, due to the complexity of the IntelliJ platform dependencies, a suite of integration tests was developed in order to ensure high code quality standards, based on the test framework offered by the platform.

Contents

Chapter 1

Introduction

Links have numerous uncontested benefits, but have a couple of inconveniences: they are rarely maintained and their targets evolve over time [4]. According to the same source, these links often suffer from decaying and lack of bidirectional traceability. Inherently, links to code are not excepted from these aforementioned disadvantages, as software changes having impact on targeting links are inevitable over the course of software project development [5]. Interestingly, in the context of code documentation, solving the problem of keeping links to code in an all-time valid state could deal with the problem of ensuring the documentation is synchronized with the source code it explains, which, according to [6], is "one of the thorniest issues facing the software engineering and technical writing communities today".

JetBrains, one of the world's leading vendors of software development tools, has identified this problem of dead links targeting code and is in need of a viable automated solution for maintaining such links, capable of detecting broken links to code and generate more recent, valid ones. The ideal application should be easy to integrate in all relevant environments that require link tracking, with a primary focus on the IntelliJ IDEA development environment. The implementation of the application in the latter mentioned environment should take the form of a plugin, which maintains the links defined in the markdown files of a project.

This report answers the question of how a plugin that automatically maintains links to code could be implemented in the IntelliJ environment with regard to the technical aspect, and aims to describe how such a proof-of-concept plugin was implemented. This will be done by first performing an analysis of the types of links that require maintenance, followed by an investigation of the solutions in which code targeted by links can be tracked, such that new, equivalent links can be generated in the scope of the IDE. This research will provide insight into what the best design choices are for building such a plugin that guards against link decaying. These design choices will then be used as guidelines for the implementation of the plugin.

This report will be structured as follows. Chapter 2 will describe the general background of the problem of obsolete links to code, providing a thorough analysis of the links that need to be tracked by the plugin and how these links are going to be tracked. A list of functional and non-functional requirements, along with project specifications will be located in Chapter 3. Chapter 4 will look into the design choices and the underlying reasons of opting for such designs in the development of the plugin. The development process, software architecture and the testing methodologies will be discussed in Chapter 5. Future recommendations together with a comparison between the requirements listed in Chapter 3 and the requirements fulfilled at the end of the project will be included in Chapter 6. Chapter 7 lays out the possible ethical issues of the project and the impact of the project on the identified stakeholders. Chapter 8 concludes the final report.

Chapter 2

The Issue of Obsolete Links

This chapter aims to present the issue of unmaintained, obsolete links that reference code and how the problem of automatically maintaining them can be solved. Section 2.1 of this chapter examines what the problem of broken links to code entails and why the maintenance process of these links is worth being automated. Section 2.2 explains the types of links that require to be tracked by the plugin. An analysis of the methods in which the plugin can retrieve the changes that have affected the code referenced by a link is discussed in section 2.3.

2.1 Overview of the Problem

According to [4], out of a study done on almost 26,000 active Git repositories, more than 9,6 million links have been found by scanning the repositories, with approximately 380,000 distinct links. A total of around 14,000 of the scanned projects had at least one link referencing code on the platform *github.com*. Furthermore, a total of 3,500 broken links to code hosted on GitHub have been identified.

With code changing frequently and the links being left behind these changes by the developers, it is of no doubt that all developers could benefit from having an automated tool solving this problem of automatically maintaining links to code. As such, the client of this project is interested in exploring this idea in all relevant environments in which this problem occurs, such as an IDE or code review system. In order to narrow the scope of the project, it has been decided together with the client that a plugin for IntelliJ IDEA, which the developers could run whenever needed to update the links targeting code listed in the markdown files in their projects should be created. The plugin should be able to track links to files, directories and lines, detect the changes that have occurred in these targeted resources and based on these changes propose new links to the user.

2.2 Types of Links to be Tracked

This section defines the types of links that require to be maintained by the plugin. A link to code can be defined in multiple ways, which depends on the environment in which the link appears. Therefore, this section will treat links that appear in the IDE environment and outside of the IDE environment respectively.

2.2.1 Links in the IDE Environment

In the IDE, links can be defined in markdown files and in the documentation comments of various programming languages, such as Javadoc for Java or Docstring comments for Python. The plugin will, however, primarily support links from markdown files, with the option of supporting links from documentation comments.

Links that appear in markdown files follow the formats imposed by the markdown syntax. This syntax allows links to be defined in multiple ways, such as inline-style links or reference-style links [7]. The plugin should support each possible markdown syntax rule of declaring links, by being able to identify links in that format and suggest newer links in the same format.

All of the options of defining a link have something in common: they need to specify what exactly is being referenced by that specific link. This is done via an URL, which can either point to a web-hosted repository of code or be a relative path in the repository containing the markdown file. In the first case, the link is said to be a web-hosted repository link. The latter case describes what is known as a relative link.

Web-hosted Repository Links

As discussed above, web-hosted repository links have an URL which points to a web-hosted repository. The URL to such a repository mainly corresponds to the pattern **https://platform/name/project-name/(blob/tree)/commit/path**, where:

- *platform* corresponds to the name of the platform followed by domain (e.g. github.com);
- *name* correspond to the name of the web-hosted repository owner;
- *project-name* is the name of the project that is linked;
- *(blob/tree)* is a constant, it can either be one of the 2 values: *blob* or *tree*. *Blob* appears in links to files and lines, whereas *tree* appears in links to directories;
- *commit* represents anything that can be mapped to a commit: be it a branch name, a tag name or a commit SHA hash. When a branch/tag name is specified, the link will point to the contents of the path at the latest commit (where HEAD is pointing at) for that branch/tag;
- *path* represents a path to a directory or file. It can also point to a specific line or multiple lines within a file, by appending to the path of the file, for example, #L21 to reference line number 21 or #L21-L25 to reference lines number 21 up to and including 25;

According to [8], when the *commit* part of the URL is a commit SHA hash, the link is called a permalink. This link will always point to the version of the path corresponding to that commit, regardless of how many changes have been applied that would have affected that path since that commit. Even though this means that permalinks cannot theoretically become broken, these links can still benefit from being maintained, as newer links pointing to more recent commits and equivalent paths can be generated. As in the case of the link not being a permalink, e.g. a branch name is provided as the *commit* part, the *path* contents at the HEAD of the branch can change as new commits are made, so the *path* contents might not be the same when someone looks at it later. Because of this reason, non-permalinks also require maintenance.

In the case of all links to web-hosted repositories, the plugin should suggest a new link having an updated path. In the case of permalinks, in addition to the updated path, the new link should point to a newer commit.

Relative Links

With respect to relative links, a link of this type has as an URL a path relative to the location of the markdown file in which that link is listed. This path can reference a file, directory or multiple lines, just like web-hosted repository URLs.

Both relative links and web-hosted repository links share a number of key features. That is due to the fact that once a markdown file is pushed on a web-hosted repository, that file will be converted into a HTML file and the links that were once relative links in that file will be converted as well to web-hosted repository links.

As indicated previously, all web-hosted repository links correspond to the format **https://platform/name/project-name/(blob/tree)/commit/path**. In the case of relative links that are converted to web-hosted repository links, *commit* will have as a value the name of the currently checked out branch or tag in the web-hosted repository. *Path* will have as a value the relative path URL of the relative link.

As a result, relative links always refer to the latest commit in the currently checked out branch. This is due to the fact that these links are equivalent to web-hosted repository links that reference a path at the HEAD commit of a

branch. Because of this reason, relative links are susceptible to change as new commits are made. It is therefore essential that these links are also witihin maintenance scope of the plugin.

2.2.2 Links Outside the IDE Environment

Outside the IDE environment, code can only be referenced via web-hosted repository links. To track changes of links in these other environments, one would have to resort to retrieving the changes that affected the linked files, directories or lines by using the APIs of the platforms that host the code, e.g. GitHub's API for links to code hosted on GitHub.

It seems that tracking changes of links in these environments corresponds to the case in the IDE environment of tracking links to web-hosted repositories that do not correspond to the currently open project. Therefore, by solving this problem in the IDE environment, the problem would also be inherently solved for these other environments.

2.3 Detecting Changes in Links' Targets

Having discussed the types of links that require to be tracked by the plugin, this section will describe the approach of detecting changes in link code targets. First, a description is given of how the history provided by version control systems can be used to tackle the problem of detecting changes in elements referenced by links. After that, methods of tracking changes of files and directories are going to be examined specifically, followed by concrete methods of tracking lines.

2.3.1 Using Version Control Systems' History

The problem of retrieving the changes that have affected a file, directory or multiple lines referenced by a link can be solved using the history that version control systems automatically store for a project. The VCS stores all the changes that affect a certain path over time, as the project evolves and code is changed. As a result, the plugin could make use of this data in tracking a path that a certain link targets.

This will work if the link for which the changes are retrieved is a relative link or a web-hosted repository link that corresponds to the currently open project in the IDE. This is because in these enumerated cases, the plugin is capable of extracting the project history from the local VCS. However, if the link is a web-hosted repository link which does not correspond to the currently open project, the plugin will have to resort to using the API of the platform that hosts the element being referenced.

2.3.2 Tracking of Files and Directories

As previously stated, the plugin can use the history that version control systems automatically store. For files, the plugin is interested in the set of changes that have affected a file path between the version of the project at the time the link was created or the time the link was last updated and the current version of the project. These changes directly result from the history of the VCS and from these the plugin can determine the current status of the file that was originally referenced by the link. The file can appear as having the same path, but having been modified, having another path and having been renamed or having been deleted.

Most of the version control systems work by tracking individual files and not directories as a whole throughout the history of a project. Because of this, the plugin is not capable of retrieving changes of directories in a such direct-fashion as for files. Therefore, for tracking directories, the plugin will proceed in the following manner:

- It will first get from the VCS the directory contents (that is, all the files and sub-directories in that directory) at the moment the link to the directory was created/last updated;

- It will then track, one at a time, each file inside of the directory from the moment the link was created/last updated to the current time;
- If there is at least one file that remained in the current directory, then the directory still exists. However, if all of the files seem to have been moved, the plugin needs to check whether there exists a majority of the files that appear to be moved altogether in another directory. If so, then it can be said that the directory has moved to this new directory;

2.3.3 Tracking of Lines

Lines referenced by links can be tracked by retrieving all the versions of the file containing the lines from the project version at the moment of link creation/last update moment of the link until the current version of the project. These versions of the file can then be passed to a line mapping algorithm, which will compare each old version of the file with the new version of the file and determine the new location of the lines. There are a multitude of existing, language-independent algorithms that tackle this problem [9]–[14].

The goal would be to track lines with the highest accuracy possible, no matter the changes that have been applied to those lines. Therefore, it is the most sensible to analyse the candidate tracking methods by comparing their performances in practice.

Asaduzzaman et al. [12] provide a comparison between LHDiff and other line tracking techniques, such as ldiff, git's blame command, W_BESTI_LINE and sdiff. Their experiment results can be visualized in Table 2.1.

Table 2.1: Performance of different line mapping techniques against the "Reiss" benchmark [12]

Method Name	Correct%	Change%	Spurrious%	Eliminate%	Time [s]
LHDiff	97.0	42.5	0	57.5	4
Git	92.5	0	0	100	0.8
W_BESTI_LINE	96.7	0	29.6	70.5	4.8
diff	92.5	0	0	100	0.2
ldiff [-i 0]	96.4	0	66.7	33.3	25.7
ldiff [-i 1]	96.4	0	25	75	73.9
ldiff [-i 3]	96.2	29.4	70.6	9	118.6
ldiff [-i 5]	96.2	0	29.4	70.6	160.7

From Table 2.1, it can be obtained that LHDiff outperforms the other methods when it comes to the overall correctness of the results, which in the case of LHDiff is 97% when tested against the "Reiss" benchmark. The 3% of incorrect results are split into 42.5% of the algorithm being able to find a new mapping of a line, but the mapping is incorrect and 57.5% of the algorithm detecting the deletion of a line when the line still exists in the file. The algorithm registered no occurrences of detecting a new mapping of a line when the line is deleted. Time-wise, LHDiff took 4 seconds on the tests that were performed.

LHDiff works by performing a diff operation on each pair of versions of a file, detecting the lines that were deleted and added between the two versions. After that, the algorithm proceeds with calculating the hamming distance between the sim-hashes of each deleted line and added line respectively, together with the hamming distance between the context lines of the deleted line and of the added line. A combined score is obtained from these two scores, which is used to create a candidate list of mapped lines for each deleted line. The algorithm then tries for each deleted line to go through the list of that deleted line's candidate lines. For each candidate line, it calculates the Levenshtein distance between the contents of the deleted line and of the candidate line, as well as the Cosine similarity between

the context lines of these aforementioned lines. Finally, the candidate line which obtains the highest score, with this score exceeding a predefined threshold, is considered as the mapped line of the deleted line.

Previously, the best solution to mapping lines has been described as the one having the greatest overall accuracy. Taking into account the overall good performance of the LHDiff algorithm, it can be said that LHDiff would be a suitable line tracking solution.

Chapter 3

Product Requirements

This chapter will introduce both a list of functional and non-functional requirements of the plugin. All of the requirements have been researched for the product with the client as the main stakeholder, evolving the requirements in parallel with restricting the scope of the product research and development. Section 3.1 will list the functional requirements of the plugin, followed by an enumeration of non-functional requirements in section 3.2.

3.1 Functional Requirements

This section will describe the functional requirements of the plugin, using the MoSCoW method. This method defines four classes of requirements based on their implementation's priority: Must Have, for core functionalities that must be included in the product release for the project to be considered successful; Should Have, for important functionalities that are not critical for delivery within the deadline; Could Have, for optional features that can be included if time and resources permit; Would Have, for least critical or low-payback features that will not be included in the release, unless rediscussed at a later time.

1. Must Have
   (a) The plugin must be able to detect links to files, lines and directories in markdown files within the open project.
   (b) The plugin must be able to detect changes in files, lines or directories referenced by a tracked link between commits.
   (c) The plugin must be able to detect changes to files between commits. Possible cases:
      i. File rename.
      ii. File deletion.
      iii. File move.
      iv. File content modification.
   (d) The plugin must be able to detect changes to single or multiple lines within files between commits.
   (e) The user must be able to set a threshold similarity value for lines and files above which the links are automatically updated. If the value is below the threshold, the plugin must inform the user about the broken link and not try to automatically update it.
   (f) The plugin must work on projects using Git as a VCS.
   (g) The plugin must track source code lines independent of the programming language in which these lines are written in.
   (h) The plugin must suggest new link paths for links found to be out of date.
   (i) The plugin must allow the user to apply the proposed changes to the links' paths through the UI.

2. Should Have

   (a) The plugin should track links to web-hosted Git repositories that do not correspond to the local repository on which the plugin is run.

   (b) The plugin should track links to web-hosted repositories, other than Git repositories, that do not correspond to the local repository on which the plugin is run.

   (c) The plugin should be able to track links from Javadoc comments.

3. Could Have

   (a) The plugin could support reading links to code in the project from outside the project itself, like git issue trackers and external documentation.

   (b) The plugin could work on projects using different VCSs, such as SVN and/or Mercurial.

   (c) The plugin could support IDEs different from IntelliJ, such as Eclipse.

   (d) The plugin could support tracking of links from comments of other programming languages than Java.

4. Would Have

   (a) The plugin would have a generic implementation that stores snapshots of the project under which the plugin is run.

Requirements 2.a, 2.b, 3.b have been re-prioritized since the initial project plan to better suit the client's evolving wishes and adapt to the team's needs. Requirements 1.g, 1.h, 1.j have been added since the initial project plan to better describe the product's evolving design.

3.2 Non-functional Requirements

This section will list the non-functional requirements of the plugin, using the MoSCoW method, as it was done for the functional requirements:

1. Must have

   (a) The plugin must have read access to all files in the repository.

   (b) The plugin must store information about the version of the project on which it was last run.

   (c) The plugin must have an implementation compatible with the IntelliJ IDEA platform.

   (d) The plugin must be reasonably usable independently from project size and number of commits.

2. Should Have

   (a) The plugin should use the VCS support functions of the IDE to parse changes.

   (b) The code of the plugin should be extensible to other relevant environments.

   (c) The code of the plugin should be modular. It should contain a core module, which is environmental-agnostic and an environmental-specific module.

Requirement 1.d is purposefully vague as its evaluation will be done by the client once the product is testable. Requirements 2.b and 2.c have been added since the initial project plan to better describe the product's design.

Chapter 4

Product Design

The previous chapter of this report listed the functional and non-functional requirements of the plugin. This chapter will further expand on the design of the plugin, based on the aforementioned requirements. Section 4.1 aims to present the primary goals of the design, followed by a discussion on how these goals are going to be satisfied. In section 4.2, the three main assumptions on which the design is based will be presented. Section 4.3 will examine the concrete design of the plugin, by providing a description of the three main operations that the plugin needs to execute: retrieving the links, tracking and updating them.

4.1 Design Goals and Their Satisfiability

This section will provide a list of the goals that the design of the plugin should satisfy. These goals mainly arise from the requirements listed in Chapter 3:

1. **Extensibility:** The code of the plugin should be easy to integrate into all relevant environments, such as IDEs other than IntelliJ IDEA or even code review systems. It is also desired, but not necessarily required, that the plugin is easily-extendable to work with version control systems other than Git.
2. **Modularity:** The design of the plugin should be as modular as possible. Having such a property would ease the portability of the plugin code to other environments.
3. **Independence:** The plugin should be programming-language independent. It should work when run on links to code written in all programming languages.
4. **Usability:** The plugin should be easy to use and be intuitive.
5. **Scalability:** The plugin should scale up to projects with a large number of links and files.

A good way to cover the extensibility and modularity design goals is by breaking the code of the plugin into two parts: a core module that is system-agnostic, and a module that is specific to the environment in which the plugin will run, namely the IntelliJ IDEA environment. In this way, when the code of the plugin needs to be extended to other environments, the core module can be reused. In addition to the core module, another module needs to be implemented for this other environment, in which the environment-specific logic is added.

The independence requirement is satisfied by using a language-independent line mapping technique, such as LHDiff, for tracking links to lines. This algorithm ensures that lines are mapped irrespective of the programming language in which the source code is written.

Usability greatly depends on the implementation of the user interface of the plugin. Therefore, during the implementation phase, the components of the UI will be specifically tested for user-friendliness. Scalability is also tied to the implementation of the module that tracks changes of links, as that is the main performance bottleneck of the plugin.

4.2 Design Assumptions

The section will provide insight into three main assumptions on which the concrete design of the plugin will be based on. These will also dictate the choices made later during the implementation stage. The list of assumptions is the following:

1. **The component will run before each commit**

   The contexts in which the component would be run is of great importance for the approach to the problem; the group has decided that the most appropriate moment for running the link checks would be before each commit is made, inside of the IDE. This is because it is the earliest moment in which link checks can be done, which could prevent the overhead of doing these at a later stage.

2. **The previous run will leave the links in a valid state**

   Between component runs, there is an assumption made that the previous run will leave the link in a valid state, such that the run after will work with a valid link, seeing whether there have been changes in the elements referenced and updating the link accordingly.

3. **The real reference of the link is in the project snapshot in which the link was created**

   When the component is run for the first time by a user, which links can be declared valid, and which not? The component would essentially need to trace back the snapshot of the project that corresponds to the moment the link was created. That is how to see the real meaning of the link that is currently checked. After that, the history of that/those line(s)/file can be followed commit-by-commit up to the current commit. This way, the component can see how the element being referenced changed and either determine the new location of it or determine whether it has been lost (deleted) along the way. Thus, depending on the current state of the project, the component can check the validity of each link and suggest a new valid link to the user.

4.3 Concrete Design

After discussing the goals of the design and the conceptual design of the plugin, this section will provide a more concrete description of the design of the product. The main functionality of the plugin can be split into three main categories: retrieving the links from the markdown files in the currently open project, tracking the changes that affected these links and updating the links via the user interface. Figure 4.1 presents a diagram describing the interaction between the three operations.

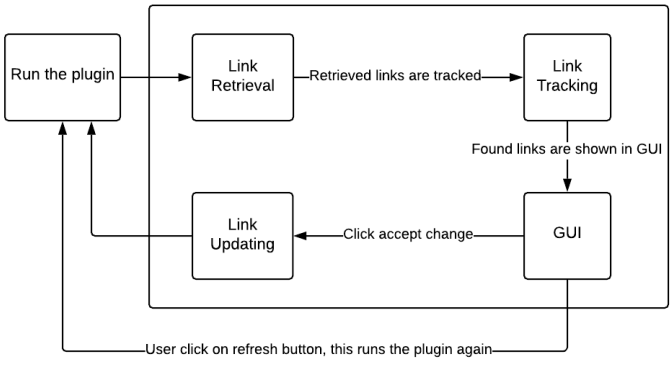


Figure 4.1: Diagram of the plugin describing the operations executed by the plugin

4.3.1 Retrieving Links

As it can be obtained from Figure 4.1, the first task that the plugin executes when it is run is to scan the project for all links listed in markdown files. A stand-alone service is dedicated to this operation, the *LinkRetrieverService*. This service ensures that all links are retrieved from the currently open project, regardless of their definition type in the markdown file.

As part of the link retrieving process, the *LinkRetrieverService* will also extract information about each link, such as the link text, link path or the line number at which each line containing the link appears in the markdown file. A link's path can then be used to classify the link into one out of a number of categories: link to file, link to a directory, link to a line/lines, or a web link. This categorization of links proves to be helpful in the subsequent task performed by the plugin, tracking changes in the code referenced by links.

4.3.2 Tracking Changes in Links

As soon as the task of retrieving links is performed and the links are categorized into the right types, the task of tracking changes in the code referenced by these links can begin. A project service solely-dedicated to this operation is the *ChangeTrackerService*, which will process one link at a time from the list of the links retrieved from the previous task.

Classification of links is done in order to accommodate the fact that each type of link requires substantially different logic of retrieving the changes that affected the element referenced by that link. Due to this reason, it has been decided that the *ChangeTrackerService* will expose a different method for each link type, as part of it's implementation.

4.3.3 Updating Links via the User Interface

Updating the links has been decided to be done manually by the user, directly from the user interface. This design choice gives the flexibility to the user to decide by himself which links require to be updated and which not. Therefore, the plugin's responsibility is to display the links in the UI, such that for each link that requires to be updated, a more recent link is suggested and displayed. A good option would be to display all the links in a UI window, split into three categories:

1. **Changed**: Represents the category of links that require the user's action — the referenced element of these links has either been deleted or moved from the path specified by the link.
2. **Unchanged**: Represents the category of links that have an unchanged path and do not require the user's action.
3. **Invalid**: Represents the category of links that were never valid, e.g. the path that they reference never existed in the history of the project.

Chapter 5

Implementation Process

In this chapter, the aspects of the development process of the plugin are discussed. Also, the choices made in product implementation are explained in more detail. First, the development process, teamwork and communication are examined in Section 5.1. Language and tool choices are explained in Section 5.2. The implementation details of the plugin are discussed in Section 5.3. Lastly, Section 5.4 addresses how testing has been done and code quality has been assured during the implementation phase.

5.1 Development Process and Teamwork

This section will give an overview of the development process of the project team, by taking a look at the software development methodology which the group has followed throughout the project. The section will also continue with examining the communication channels used by the group internally as well as with the stakeholders of the project.

5.1.1 Software Development Methodology

The team has agreed upon using the Scrum framework as the main software development methodology for the development of the plugin. Scrum has been proven to deliver results in a quicker time than other development methodologies, and due to the short time frame of this project, this was the most sensible methodology to opt for.

Furthermore, the project team has decided that sprint iterations are going to be executed on a weekly basis. By having short iteration sprint, results can be seen much faster. It also made the most sense since the group had to give a demo of the product to the client on a weekly basis.

As part of each sprint's iteration, weekly Monday morning meetings dedicated to Sprint Planning have been held. During these meetings, each team member shared information with the other team members about the work done in the previous week. As part of this meeting, the backlog of that week was created, issues were prioritized and each team member was assigned to an equal amount of issues to solve in that week.

GitLab's issue board has been used to keep track of the backlogs throughout the plugin development process. This decision has been made in order to minimize the number of different tools that the team uses, as the code of the plugin is already hosted on GitLab. In the issue board, four different labels have been used to denote the status of the issues in a sprint. Figure 5.1 depicts all four labels used by the team.

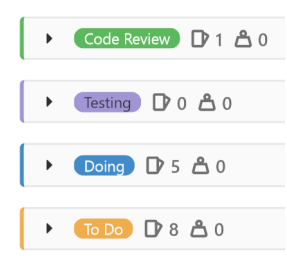


Figure 5.1: Backlog Labels

5.1.2 Communication Between Team Members and Stakeholders

To ensure good communication between all team members, as well as between the project team and the stakeholders of the project, different communication platforms have been used. Firstly, the group communicated internally via a dedicated WhatsApp channel. Communication with the company has been done using a dedicated JetBrain's Slack channel. Lastly, communication with the TU Delft coach and TA has been done using a Mattermost channel.

On a weekly basis, Monday morning meetings were held between the project team and the TU Delft supervisors of the project. A weekly meeting with the client has been held on each Friday afternoon, in which a small demo showcasing the features included in that week's sprint iteration was given. Lastly, internal meetings between the team members were held each Monday, Wednesday and Friday, in order to keep the entire project team up to date with the progress of each member of the team.

5.1.3 Changes in Development Process

During the development process, a few adjustments have been made in order to improve the quality of the deliverable. In the beginning of the project, it has been found that merge requests of new features had no minimum number of approvals required before being merged into main branches. This was immediately seized by one of the team members, upon which a minimum approval of two developers had to be obtained for a merge request to be finally merged.

Another change that occurred in the development of the plugin has been the addition of a static analysis step in the build of the plugin via Kotlin Lint. This decision has been taken as parts of the code have diverged in terms of code style, due to different developers working on them. As a result, adding the linting improved the deliverable source code's quality, by uniformizing the code style throughout the project.

5.2 Programming Language & Development Tools

This section will first introduce the choice of the programming language used for developing the plugin. Following that, the decision of using the IntelliJ IDEA editor for the development process is motivated.

5.2.1 Kotlin

Kotlin is a programming language developed by the client of the project, JetBrains. It is a powerful programming language, designed to fully-interoperate with the Java language and provide solutions to problems that often occur while programming in Java.

The client has explicitly expressed his interest in the plugin being developed in Kotlin. As a consequence, the group has adhered to the requirements received from the client and has written the source code of the plugin in Kotlin.

There was only one initial drawback of using Kotlin: no member of the group had previous experience with this language. However, as the language shares a substantial amount of key features with the Java language, with which each team member had previous experience, this has not caused any delay in the development process. As suggested by the client, each team member followed the Kotlin Koans tutorial and learned the syntax and the rules of the language in the first week of the project. Following that, during the development process, all team members progressively gained experience with the language. As a result, each group member is now comfortable and familiar with the Kotlin language.

5.2.2 IntelliJ IDEA

The group has commonly agreed that each member should use the IntelliJ IDEA development editor in the development process of the plugin. This decision was the most sensible, given that the group is developing a plugin for the exact same environment. As a result, this choice gave the team members the capability of installing and testing the plugin in their own editors, creating a fast feedback loop for the features added within each sprint iteration, as the group members were the first users of the plugin.

5.3 Software Implementation

In this section, the implementation of the plugin is going to be discussed in more detail. As a result, the concrete design presented in Chapter 4 will be elaborated. To make the explanations easier to follow, the components of the plugin will be explained, by grouping them into two categories: backend and frontend respectively.

5.3.1 Backend Implementation

When the user runs the plugin, the plugin action is called, which in turn calls a number of services in the backend of the plugin. The main components of the backend that are called are: the link retrieving task, the task of tracking changes in the links and the task of generating new links.

Link Retrieving Task

The first step in the plugin is the link retrieving task, which is executed by the *LinkRetrieverService* class. The first functionality of this service is to get all markdown files in the project scope. Then, it gets all the links defined in these markdown files by making use of the Program Structure Interface (PSI) Tree, a layer in the IntelliJ Platform that is responsible for parsing files and creating the syntactic and semantic code model.

The *LinkRetrieverService* extracts context information of each link found in the markdown files of the project. For each link, it stores information such as the link's path, which is encapsulated into a *LinkInfo* class.

After the link retrieving process has ended, the path of each identified link is passed into the factory method of the class *LinkFactory*, which, based on the pattern of the path, classifies the link into a number of categories. There are two main categories: web links, which correspond to web-hosted repository links to code and relative links. Each of these categories has various subcategories, such as web links to files, directories, lines or relative links to files, directories or lines respectively.

The result of this whole operation is a list of link objects. This list will then be iterated through link-by-link, where each link will be visited by an implementation of the *ChangeTrackerService* interface, which defines the logic of gathering changes for each of the elements that these links reference.

Change Tracking Task

The second step in the plugin is the tracking of changes in the elements that the links reference. The service responsible for this task is an implementation of the *ChangeTrackerService* interface, named *ChangeTrackerServiceImpl*.

As part of the contract imposed by the implementation of the base interface, this service has to implement a method for each link type. This is due to the fact that each link type requires different logic for retrieving the changes that the link references.

Tracking changes in the elements referenced by links requires data on the history of the currently open project. As a consequence, this task has to retrieve this history in some way. To achieve this, the service makes extensive use

of the class *GitOperationManager*, which handles all the logic of calling Git commands on the project, processing the outputs of these calls and handling errors that might occur. This class in turn makes use of IntelliJ's open source plugin Git4Idea, which facilitates the interaction between the plugin and the Git executor installed on the machine.

The changes that have affected files and directories can be directly retrieved from calls to the *GitOperationManager* class. However, links to lines require one extra layer of processing, which is done by the class *LineTracker*. This class represents an implementation of the LHDiff algorithm, a high-accuracy line-mapping algorithm which was examined in Chapter 2. An important thing to note is that the developers of the team spent a consistent amount of time in stabilising the implementation of the algorithm, such that the accuracy of the line mapping technique is as precise as possible.

Change tracking methods in the *ChangeTrackerServiceImpl* service will share a key, similar feature: they will all have the same return type. This is done by having each method return an implementation of the interface *Change*. There are three implementations of this interface: the *CustomChange* class, which corresponds to changes of links to files and directories, *LineChange* which corresponds to changes of links to a single line and *LinesChange*, which corresponds to changes of links to multiple lines.

A user configurable similarity threshold for files, directories and lines should be taken into consideration by the implementation of retrieving changes in link targets. That is, in the case a resource targeted by a link changes more than a predefined threshold, it should be considered as deleted, and otherwise moved.

At the end of this task, the result will consist of a list of change objects. Each change object will then be paired to the corresponding link object into a list of pairs, which is then passed to the next task.

Link Updating Task

The third and last step in the backend of the plugin is represented by the link updating task. This is done by the class *LinkUpdaterService*, which has the responsibility of generating a newer link, as well as updating a link in the markdown file of provenience with this more recent link.

As dictated by design, this link updating task is triggered explicitly by the user via the UI. Therefore, the service's exposed updating method will be called upon an event in the user interface.

To generate a more recent version of the link, this service accepts as input a pair of a link and a change object. This pair is already computed from the previous tasks and it is persisted in the main-memory of the IDE. Using this pair, the service is capable of generating a new link with the same characteristics, with the exception of a more recent link path, which is extracted from the change object.

As discussed previously, the *LinkRetrieverService* will store context data from each markdown file for each link. As part of the context data, it also stores the PSI element that each link corresponds to. Using this PSI element, the *LinkUpdaterService* knows the exact location in the markdown file at which each link needs to be replaced by the newer link.

5.3.2 Frontend Implementation

There are two different components in the frontend part of the plugin. The first one is the Tool Window, which is the main part of the frontend. The second one is the Settings. This section will introduce both of these components and will go into details about them.

Tool Window

The first and main UI component is the Tool Window, which is shown in Figure 5.2. It has been decided to use a tree structure for this window in order to increase the intuitive usability of the product. The user is thus able to navigate seamlessly through the Tool Window, evaluating each link's changes.

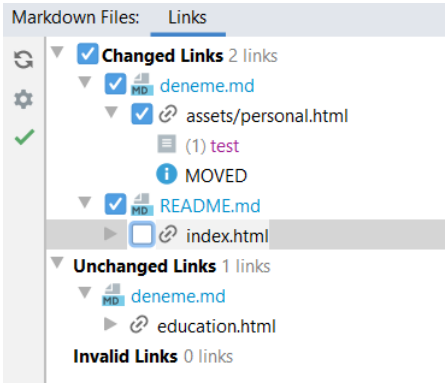


Figure 5.2: Screenshot depicting the Tool Window of the plugin

As depicted in Figure 5.2, the biggest part of this tool is represented by the link tree. Following the design of the UI, this tree has three main leaves: one for changed links, unchanged links and invalid links. For each link, it contains another sub-tree, which shows the link's information.

To perform an update operation, the user has two choices: either press the right click button on a link, and then press on "Accept Change", or by ticking the checkbox of each link. The user is capable of ticking multiple changed link's checkboxes in order to perform a batch update operation on these links. After ticking the checkboxes, the user needs to click on the checkmark on the left part of the Tool Window, which will trigger the update operation.

A button for accessing the settings page of the plugin has been included in the left bar. The plugin action can also be re-triggered by pressing the first button in the left bar.

Settings

The second user interface component is represented by the Settings. The plugin settings have been placed into the default settings of Intellij IDEA. These can be found under Tools/Link Tracking.

There are 2 different pages that constitute the plugin's settings. The first one is a page where the user can set similarity thresholds for tracking links to files, directories and lines. The second one corresponds to a token management page for managing personal access token to private GitHub repositories. The latter is needed in order to track links to web-repositories that do not correspond to the currently open project.

5.4 Testing & Code Quality

This section is about the the ways the plugin has been tested. Code quality assurance methodologies are also examined as part of this section.

5.4.1 Testing Procedures

To guarantee that the plugin's functionalities work as intended, as well as to increase the code quality of the plugin, testing has been performed. As a consequence, testing has been executed on three main levels: unit, integration and system testing.

Unit Testing

Unit testing has been the least utilized mean of testing for the plugin. Most of the functionalities of the plugin have been IntelliJ platform-dependent. As a consequence, these functionalities have been inherently hard to unit test, due to the large amount of mocks that each test case would have required to set up.

However, where the mocks where not needed or hardly needed, unit testing has been performed. Examples of such tests are the tests written for the data classes of the plugin, as well as for the *LinkFactory* class.

Integration Testing

A viable solution of testing has been proven to be by means of integration testing. All of the three main tasks of the plugin, namely the link retrieving, change gathering and link updating parts have been made sure to be tested using automated test cases.

One of the main functionalities of the plugin, the operation of retrieving changes in the links targets, requires executing specific Git calls on a project's commit history. Mocking this would have taken a considerable amount of effort, and as such, the group has tried to use IntelliJ IDEA's testing framework of the Git4Idea plugin, which provided the necessary infrastructure for executing such tests. A substantial amount of time has been dedicated by the team developers into setting up the integration tests infrastructure, as the testing framework of the aforementioned Git plugin has not been distributed. However, as the source code of the plugin is open source, some of the necessary test files have been adapted to work with the plugin under development.

As a result, the integration test cases are using IntelliJ IDEA's testing framework, which spawns a test project with each test case execution. For the *LinkRetrieverService* class, a dummy project containing a set of markdown files, each containing links of multiple types is created with each test case. The test cases then verify that all of the markdown files within project scope are correctly identified, that all of the links inside these markdown files are retrieved and that the links are categorized into the right classes. The tests for the *ChangeTrackerServiceImpl* class work as well on a test project, manipulating the Git history specifically for each test scenario and testing the outputs of the methods of the class depending on this history. Lastly, the *LinkUpdaterService* class tests the update operation of links in a project containing a set of test markdown files.

Manual Testing

Manual testing has also been performed on the code of the plugin. It has been one of the most utilized methods of testing the product, as all of the other sub-levels of testing the plugin required a great number of dependencies to be setup before testing could be performed.

For this, all team members used a common project dedicated to testing, namely the demo-project that was coded for the selection process of the students interested in working on this project. In this project, multiple test markdown files have been added, each containing multiple types of links. With each feature being added, the plugin has been run on this project, making sure that each of the links in the project gets a correct result.

5.4.2 Code Quality Assurance Methodologies

Code quality is very important for future maintainability of the source code. After this internship, a different team from JetBrains could be interested in developing the project further and bringing it into production. Therefore, the code that the group is writing now should be safe, easy to understand and stable. To achieve this, various procedures have been followed during the project development:

1. **Documentation**: Javadoc comments have been included in most classes and methods. A piece of higher level documentation has also been included in the repository, explaining the software architecture of the plugin.
2. **Static Analysis Tools**: Kotlin Linter (KtLint) has been used to maintain the same code style throughout all classes in the project.
3. **Code Review Procedures**: An approval of minimum two developers has been set on each merge request. This way, more than half of the project team was aware of any change that was introduced with every feature addition.
4. **Continuous Integration**: A build and test stage have been added to the continuous integration pipeline. The pipeline was run with each commit being pushed to the remote repository, making sure that no bugs are introduced during the development process.

Chapter 6

Final Release

Over the course of this project, the plugin has been built under the requirements listed in Chapter 3. In Section 6.1 of this chapter, an evaluation of the final product is performed by providing a comparison between the features included in the final release and the requirements. Section 6.2 will list the main challenges faced during the implementation process.

6.1 Product Evaluation

This section refers to Chapter 3 for the requirements used by evaluation. The project can be recognized as a success since most of the requirements have been completed. All of the must-have requirements defined in the MoSCoW method have been implemented. Other should-have and could-have requirements are partly implemented.

6.1.1 Required Features

The final product has been built by prioritizing the Must Have requirements. As a result, the end result satisfies all of the aforementioned requirements:

- ✓ It is implemented as a plugin that is compatible with the IntelliJ IDEA platform.
- ✓ It can fully support maintaining relative links.
- ✓ It can support maintaining links to web-hosted repository code that correspond to the currently open project.
- ✓ It works on all projects using Git as a version control system.
- ✓ It detects links to files, single lines, multiple lines, and directories in markdown files within the open project.
- ✓ It fully supports tracking lines irrespective of the programming language in which the lines are written in.
- ✓ It is capable of detecting changes that have occurred in files, lines and directories targeted by links.
- ✓ It can detect, for files targeted by links: the file being renamed, deleted, moved or modified.
- ✓ It can detect, for directories targeted by links: the directory being renamed, deleted or moved.
- ✓ It can detect, for lines targeted by links: the lines being moved or deleted.
- ✓ It can generate new, equivalent links to broken ones, depending on the identified changes in the broken link target.
- ✓ It gives the user the ability to perform an (batch) update operation of broken links with more recent ones via the user interface.
- ✓ It gives the user the ability to configure in a settings page a content threshold similarity for files, directories and lines, under which a tracked link target is considered deleted. Changes having similarities above these thresholds are considered as moved.
- ✓ It gives the user the ability to save via the settings page Personal Access Tokens (PAT) to GitHub repositories, which are required to track links that do not correspond to the currently open project and require calling the code-hosting platform API.
- ✓ It can persist to disk the commit SHA hash at which the plugin has been last run on.
- ✓ It has been optimized to work with projects containing a large commit history.
- ✓ It has read access to all files in the repository, as the plugin uses Git4Idea as a dependency, an IntelliJ IDEA plugin devoted to executing Git commands from within the IDE.

6.1.2 Optional Features

Most of the optional features have not been implemented for the final release of the plugin, with one exception: a VCS integration with IntelliJ IDEA has been used for the plugin. This has been fulfilled by using the Git4Idea dependency, a version control system integration with the platform, that was used for retrieving changes in link targets.

The code of the plugin has not been fully modularized. Out of the three main operations of the plugin — retrieving the links, tracking the changes in link targets and updating the links — only the operation of gathering the changes has been adapted to work in different environments. A substantial refactoring would have been required near the end of the project in order to fix this goal, but other issues, such as optimizing the performance of the change gathering process have been given higher priority.

Maintaining web-hosted repository links that do not correspond to the currently open project has also not been within this project's scope. The client has mentioned not to give this issue priority, as implementing such a feature relies on using the APIs of the platforms that host the code being referenced by the links. These APIs go through substantial changes very frequently, and as such our feature implementations would become deprecated after a short period of time.

The plugin only supports maintaining links of projects that work with Git as a version control system. The initial plan was to fully support links of projects using Git, and then move on to supporting other VCS, as time allows. This has not been the case in this project, but given more time, this feature could have been implemented.

Tracking links from comments of programming languages has also not been included. This is caused mainly due to the initial design of the plugin accommodating mostly links defined in markdown files. As a result, incorporating support for tracking links from programming language comments would have required a substantial amount of refactoring and redesign of the plugin near the end of the project, which would not have been ideal as the core functionality of the plugin would have been changed and as a result errors could have been introduced.

6.2 Encountered Challenges

This section will enumerate the parts of the project that required the most attention from the project team, as well as reflect on those parts would have benefit from more attention. In total, there have been three main challenges:

1. **Testing**: A good amount of time has been devoted to testing the product. It has been done on three different testing levels: unit, integration and manual. Throughout the project, manual testing has been the most utilized mean of testing. However, automated cases using IntelliJ IDEA's testing framework have been added later in the project, due to the difficulty of setting them up, and as a consequence a few test scenarios that would normally require to be tested in order to assure production-level safety of the code have been left out due to time shortage.

2. **Tracking Links to Lines**: The most complex part of the plugin is by all means tracking the links to lines. To achieve this, the team required to perform a research study in order to determine the most viable solution of mapping lines between versions of a file. To this extent, LHDiff has been determined to be a suitable solution, and an implementation of it has been added to the core functionality of the plugin.

3. **Modular Code**: As mentioned earlier, throughout the project the developers have tried to keep the code as modular as possible, but this was not always possible. The plan was to leave the modularizing process of the code near the end of the project, when the main functionalities of the plugin are fulfilled, but the priority of this has changed by that time.

Chapter 7

Discussion on Ethical Implications

This chapter is meant to discuss the ethical implications of the project's development and use. Its aim is to mitigate the risks involved with the production of the software and to ensure that both the project and the product have an overall positive impact on their users and society at large. Firstly, this chapter individuates the various stakeholder groups that will be affected by the software's use in Section 7.1. In Section 7.2 it discusses the main ethical issues concerning the product. Lastly, in Section 7.3, for each of these issues, it discusses choices that have been taken and presents future recommendations for mitigating and addressing these issues.

7.1 Stakeholder Groups

During the preliminary design stage the development team discussed and analyzed the product requirements in order to understand who would be the stakeholders in the realization of the final product. The following are the entities or groups found to have a direct or indirect stake in the product:

1. **JetBrains**: As the paying client who commissioned the project, JetBrains is the first and most direct stakeholder. It is in their interest to obtain a well functioning, stable, maintainable and possibly marketable product. They have an interest in receiving software that not only works well, but is designed well enough to be open to future upgrades should JetBrains wish to do so. Developers who may have to work to maintain and update the product in the future are considered part of this stakeholder group.
2. **Plugin Users**: The final product is designed to be used on a regular basis by other software developers. It is in the users interest to receive a product that not only works well and does not cause unwanted issues, but also has a positive impact on their workflow without limiting their autonomous choices.
3. **Readers of Documentation**: Our product is designed to contribute to the production and maintenance of software documentation. Readers of such documentation have an interest in receiving clear, up to date and useful documentation.

7.2 Main Ethical Issues

The team has identified the following significant ethical issues that can arise from the development, marketing and use of the product. They will now be listed with a brief explanation of their nature.

7.2.1 Code Quality and Readability

The team made a conscious choice to consider code quality as an ethical issue, in addition to it being a practical one, according to Value Sensitive Design. This is because work on a piece of software does not stop once the software enters production, and it can involve many software developers at different points in time. In fact, most of the resources spent on the production of a piece of software are used not in the development, but in the maintenance and upgrade of the same [15]. This means that the team's choices during development can and will have an impact on the work of other people in the future. As an example: another developer could be assigned to make updates to the software in the future, with specific requirements to deliver and a deadline for them. The amount of time that they will have to spend on such a task will depend on the quality of software the team produced. In other words, the team's choices when developing the software could impact another developer's use of their free time in the

future. Additionally, should this software be made open source, its documentation and readability will significantly impact its openness and value to other developers. This issue mainly impacts the first group of stakeholders: JetBrains.

7.2.2 Automation of User Decisions

The product's goal is to automate a task: that of updating links which are out of date. The easiest way to implement this would be for the software to, in a single pass, detect outdated links and update them straight away. This however risks falling into what scholar Evgeny Morozov, in his book "To Save Everything, Click Here" calls *Technological Solutionism*, the design strategy to solve small practical problems by inbuilding specific choices into technological products, bypassing the user's own judgement about those problems [16]. While this approach may seem attractive, Morozov argues, on a broad scale it discourages users from forming their own judgments and training their critical reasoning. This issue mainly impacts the second group of stakeholders: plugin users.

7.2.3 Privacy Risks in Handling Sensitive Data

The product will require users to input credentials to track links to repositories on code hosting services such as Github. This means, the users will have to trust the product to handle their sensitive data securely and transparently, according to GDPR regulations. Additionally, the product will have access to potentially sensitive and/or valuable software files. The users may have reasonable concerns on whether the product is collecting any information about the files that are being scanned. At the time of writing the product does not have the capability to collect significant amount of identifying data. Nevertheless, since the product does handle at least some significant data, and given the widely accepted moral reasons for protecting personal data [17], an effort was made to comply with the strictest international privacy regulations. This issue also mainly impacts plugin users.

7.3 Design Choices and Recommendations

This section will now list, for each of the aforementioned issues, the design choices the team made to address them, and the recommendations the team has for future use and update of the plugin.

7.3.1 Code Quality and Readability

Throughout the project, and as much as time, skills and resources permitted, the team put effort into making the product modular. This was done by designing replaceable components that would be as independent as possible on the underlying platform and other external dependencies. An effort was also made to keep the code clean and readable, according to JetBrains' own guidelines for code style, comments were used throughout the code wherever deemed necessary or useful.

7.3.2 Automation of User Decisions

A choice was made to avoid designing the product to automatically update links without user input. Instead, the product will show the user the results of the preliminary scan and, for each of the out-of-date links, offer the user the option to update it according to a computed suggestion. This strategy is beneficial both for transparency and usability and should not be significantly altered in the future.

7.3.3 Privacy Risks in Handling Sensitive Data

The product is designed so as to only store credentials locally, within the settings storage of the IntelliJ IDEA platform. The user will have the option to change or remove such stored credentials at any time. Additionally, the team advises that the code for the product be made open source so as to guarantee transparency on the product's operation and security. While publishing the software's source code would imply security risks, at the time of writing the software does not have the potential to expose vulnerabilities serious enough to counterbalance the benefits gained from transparency and openness.

Chapter 8

Conclusion and Recommendations

The purpose of this report was to answer, with regard to the technical aspect, how a plugin that automatically maintains links to code could be implemented in the IntelliJ environment, describing the process of implementing such a proof-of-concept plugin. To do so, a research study has been done in order to determine the types of links that require to be tracked, as well as an investigation on how tracking changes in the targets of links could be fulfilled. The findings of the research have been incorporated into the design of the plugin, which in turn dictated the effective implementation of the software.

The result of this software project endeavour is an IntelliJ IDEA plugin that fully supports the maintenance of relative links, as well as links to web-hosted repositories that correspond to the currently open project in the IDE. The plugin has the ability to scan through all of the markdown files within project scope, retrieve the links defined in these files, retrieve the changes that have affected the link targets and based on these found changes generate newer links. Finally, the user has the ability to update each particular decayed link via the user interface of the plugin.

The features that have been included in the final release of the plugin satisfy all of the client's initial expectations and needs, as the client directly expressed his satisfaction with the built product. This is due to the fact that all of the client's needs have been reflected by the Must Have requirements, which have been given the highest priority throughout the project and thus have been fully covered.

On the other hand, most of the optional categorized features have not been realized by the project team due to the short time of the project. The code has only been partly modularized, with one operation, retrieving changes in link targets being adapted to work in other environments. Furthermore, links to web-hosted repositories not corresponding to the open project are also not supported. As a result of most of the optional features not being implemented, the project team has in mind four main important improvement ideas that can be added to improve the plugin further:

1. Links to web-hosted repositories that do not correspond to the currently open project could be implemented.
2. The source code of the plugin would need to be refactored in order to modularize and ease the portability to other environments of the operations of retrieving and updating of links respectively.
3. The source code of the plugin needs to have more automated integration test cases implemented before bringing the plugin into production, as the code was tested but it would benefit from more tests being added.
4. Further optimizations on the plugin's retrieving changes in link targets operation can be done, as it was optimized but not to a level that allows the plugin to perform well irrespective of the number of commits in a project's history. The main bottleneck of the plugin is represented by calling Git commands, induced by their loss of performance with the increase in the number of commits.

Acronyms

API Application Programming Interface. 4, 19, 20

GDPR General Data Protection Regulation. 22

IDE Integrated Development Environment. iii, v, 1, 2, 4, 8, 9, 10, 15, 19

SHA Secure Hash Algorithm. 3, 19

SVN Subversion. 8

UI User Interface. 7, 9, 11, 15, 16

URL Uniform Resource Locator. 2, 3

VCS Version Control System. 7, 8, 20

Bibliography

[1] E. Toporov. (2017). The art of code, visualized, [Online]. Available: https://blog.jetbrains.com/team/2017/02/08/the-art-of-code-visualized/.

[2] T. U. Delft. (2018). Logo, [Online]. Available: https://www.tudelft.nl/huisstijl/logo/.

[3] JetBrains. (2020). Jetbrains logo, [Online]. Available: https://www.jetbrains.com/company/brand/logos/.

[4] H. Hata, C. Treude, R. G. Kula, and T. Ishio, "9.6 million links in source code comments: Purpose, evolution, and decay," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 1211–1221. DOI: 10.1109/ICSE.2019.00123. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00123.

[5] L. Li, L. Zhang, L. Lu, and Z. Fan, "The measurement and analysis of software change based on software repository," *2nd International Conference on Software Engineering and Data Mining, SEDM 2010*, pp. 289–294, Jan. 2010.

[6] R. Pierce and S. Tilley, "Automatically connecting documentation to code with rose," in *Proceedings of the 20th Annual International Conference on Computer Documentation*, ser. SIGDOC '02, Toronto, Ontario, Canada: Association for Computing Machinery, 2002, pp. 157–163, ISBN: 1581135432. DOI: 10.1145/584955.584979. [Online]. Available: https://doi.org/10.1145/584955.584979.

[7] A. Pritchard. (2017). Markdown cheatsheet, [Online]. Available: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet#Links.

[8] GitHub. (2020). Getting permanent links to files, [Online]. Available: https://help.github.com/en/github/managing-files-in-a-repository/getting-permanent-links-to-files.

[9] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, USA: IEEE Computer Society, 2009, pp. 595–598, ISBN: 9781424434534. DOI: 10.1109/ICSE.2009.5070564. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070564.

[10] Git. (2019). Git blame, [Online]. Available: https://git-scm.com/docs/git-blame.

[11] S. P. Reiss, "Tracking source locations," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 11–20, ISBN: 9781605580791. DOI: 10.1145/1368088.1368091. [Online]. Available: https://doi.org/10.1145/1368088.1368091.

[12] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta, "Lhdiff: A language-independent hybrid approach for tracking source code lines," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13, USA: IEEE Computer Society, 2013, pp. 230–239, ISBN: 9780769549811. DOI: 10.1109/ICSM.2013.34. [Online]. Available: https://doi.org/10.1109/ICSM.2013.34.

[13] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in *Proceedings of the 29th International Conference on Software Engineering Workshops*, ser. ICSEW '07, USA: IEEE Computer Society, 2007, p. 14, ISBN: 0769528309.

[14] ——, "Tracking your changes: A language-independent approach," *IEEE Softw.*, vol. 26, no. 1, pp. 50–57, Jan. 2009, ISSN: 0740-7459. DOI: 10.1109/MS.2009.26. [Online]. Available: https://doi.org/10.1109/MS.2009.26.

[15] D. D. Smith, *Designing Maintainable Software*. Berlin, Heidelberg: Springer, 1999, ISBN: 9780387987835. [Online]. Available: https://books.google.nl/books?id=Hb-zwrgN4UcC.

[16] E. Morozov, *To Save Everything, Click Here: The Folly of Technological Solutionism*. Public Affairs, 2013, ISBN: 9781610391382. [Online]. Available: https://books.google.nl/books?id=b3n8AgAAQBAJ.

[17] J. van den Hoven, M. Blaauw, W. Pieters, and M. Warnier, "Privacy and information technology," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Metaphysics Research Lab, Stanford University, 2020. [Online]. Available: https://plato.stanford.edu/archives/sum2020/entries/it-privacy/.

Appendix A

Project Skills Reflection Reports

This appendix will contain all of the reflections reports of the project team members. A section will be dedicated to each member of the team.

A.1 Reflection Report of Tudor Popovici

This section of the appendix incorporates the reflection report of the student Tudor Popovici, as part of the deliverable for the Project Skills module of the Context Project course.

Introduction to Project Team

Over the past 10 weeks, our group has been working on the project with the name "Tracking Changes in Links to Code" offered by the company JetBrains, as part of the Q4 Context Project. We are a total of 4 students contributing to this project: Ceren Ugurlu, Irem Ugurlu, Tommaso Brandirali and Tudor Popovici, the person writing this report.

The project's goal is to extend the functionality of the IntelliJ IDEA development environment by creating a plugin capable of maintaining links that reference code. The plugin should work in the following way: the plugin should first scan for all the markdown files in the current open project, retrieving all the links to code that are defined in these files. Following that, the plugin should track for each link the changes that have affected that link's path over time. Using these changes, the plugin should then generate a newer link, containing an equivalent and more recent path, which will be displayed next to the old link in the user interface of the plugin. The user has then the capability of manually updating each link with its newer version from the UI.

Task-related Conflicts Encountered

In the last ten weeks of plugin development, there were no major task-related conflicts arising among the team members of this project. However, as it is "natural" in any project team to happen, during these ten weeks a few inevitable task-related conflicts have occurred, most of them during the "Storming" stage of the project. According to the task-circumplex model presented in the video lectures of the Project Skills module, the conflicts that our group has encountered have been part of the third quadrant: these tasks correspond to resolving conflicts of viewpoints. These conflicts result because of different, non-black-and-white opinions of different members on a certain subject.

To give an example of such a situation, a lengthy discussion was made on whether to include a static analysis step in the CI/CD that is triggered on each commit being pushed. Some of the team members believed that a simple run, before each commit, of the IDE's functionality of formatting code on the newly added source code will suffice. Other team members believed that a static analysis step should be included in the pipeline steps.

Another situation in which a cognitive conflict task occurred is that when the group was trying to decide whether some features should be included in the final release of the plugin or not. Some team members were trying to pin-point the importance of those features being included in the final release, while other team members were not so convinced about the priority of such features. Again, a discussion was made over this topic, under which each team member got to express his point of view.

Intragroup Conflicts Encountered

As far as intragroup conflicts are concerned, a few conflicts of this type have also developed throughout this iteration of the Context Project. All these conflicts that occurred had something in common: they had the task and the process of conducting the task at the main root of the conflict.

One example of a situation which generated conflict is when a merge request to one of the main, protected branches of our repository had mistakenly deleted almost 800 lines of testing code when merged. To better understand the problem, the deletion of those lines was not an intentional act by any of the team members. It has happened as the branch which was merged was

not containing all the changes of the other branch. The main problem in this situation was that this has gone unnoticed by both the reviewers of the merge request, as well as the developer that created the merge request. After a relatively short amount of time, one of the team members has noticed this issue and called the problem out on the platform that the group uses for communicating. Following this, the group members have tried to perform a mutual root cause analysis of what could have caused this to happen, while settling on a solution against such a scenario repeating once again.

While the team was overall highly functional over the development phase of the project, there were some moments in which what is described as process loss developed in the teamwork. This has been caused by short periods of reduced effort, as it has happened for members of the team to not be able to finish the tickets they were assigned to in that specific sprint iteration. However, this has not been a continuous problem through-out the development of the project: most of the sprint iterations had mostly fully done tickets, and where a ticket was unfeasible to be done in that iteration, a valid explanation existed for that.

Conflict Resolution Techniques Used

The task-conflicts described in the section above have been solved by means of transparent communication among the team members: in the case of the extra static analysis step being added to the CI/CD pipeline, the team members have listened to each other's opinions and have reached a mutual consensus: the team will use the code formatting functionality of the IDE, with the option of adding the extra step as proven to be needed in the future.

For the final release feature list creation, again solving uncertainty via transparent communication resolved the problem, as well as moderating stress. Therefore, the goals of the plugin development have been refined to a set of features that are more within the reach of this project timeline.

The intragroup conflict of the accidental deletion of the code has been resolved by all team members by having a highly collaborative approach to the problem. Immediately after announcing the problem, the team members have tried to identify what exactly has caused this to happen, as well as find a procedure with which this will not happen again: all team members will be more careful at the changes that a merge request introduces to a main branch, as well as each team member making sure that the branch that (s)he is working on is up to date with the main branch.

In the case of reduced performance that occurred over short period of times, we have made sure that each team member is identifiable by his contributions to the project. This is done by listing the tickets that each member is assigned to into the Sprint Retrospective for each sprint iteration, where each ticket is marked as being done or not, and in the latter case an explanation is being given as to why the ticket has not been completed. Team members have also been assigned to more tickets with a higher impact on the outcome of the plugin as previously. These types of tickets have been found more interesting by the team members. This latter approach has been proved to work the best in the scenario of reduced effort in our project.

A.2 Reflection Report of Ceren Ugurlu

This section of the appendix incorporates the reflection report of the student Ceren Ugurlu, as part of the deliverable for the Project Skills module of the Context Project course.

Introduction to Project Team

As part of the CSE2000 Software Project, we were working on the "Tracking Changes in Links to Code" project during the last 10 weeks. The project is offered by JetBrains which is a software development company whose tools are targeted towards software developers and project managers. We are a group of 4 students working on this project. The project group consists of me, Ceren Uğurlu, and other group members Irem Uğurlu, Tommaso Brandirali, and Tudor Popovici.

The purpose of this project is to create a plugin that helps to keep links in markdown files up to date. An IntelliJ Idea plugin has been created for this. Users can see all the links contained in the markdown (.md) files in their projects through the UI (User Interface) of the plugin. It classifies the links as 'changed', 'unchanged', and 'invalid'. If these links are invalid, the user can learn the reason. If it is a changed link, the plugin finds the new link path and the user can update this link with its newer version via UI.

Task-related Conflicts Encountered

During the project, we did not encounter many situations that would cause a conflict in the scope of our project. However, the most difficult part was the brainstorming stage. Our client gave us the freedom to choose the direction of the project and the

tools will be used in the project. It can be seen as an advantage but it also means a big responsibility for all of us. Because those decisions we made would have a big impact on the final product. Therefore, we had many brainstorming meetings to make the correct choices with my groupmates. As usual in every project team, there were moments that we had different ideas and conflicts. We discussed all different options and everyone in the group respected each other's thoughts. I believe that we managed it very nicely and it became a healthy conflict. As a result, these different opinions and conflicts did not cause any problem. We made common decisions and nicely concluded these meetings.

As I said we did not encounter a big conflict that causes process or motivation loss. We had small conflicts such as using Kotlin Linter as a static analysis tool. In our brainstorming meetings, we had a decision to use it to make our code more styled. Then, some of us did not want to use it at this stage. Because it gives a lot of warnings and errors which can be counted as unnecessary and time loss at the beginning of the project. We discussed this and everyone agreed to continue to use this tool in a later part of the project. As we agreed, we started to use it again in the later weeks of the project.

There are motives why we do not have big conflicts. Firstly, I believe that we had good harmony within the project team. Everyone was open to new ideas and was respectful to each others' thoughts. Anyone did not blame someone else for an issue related problem. Everyone tried to help each other about issue related problems. If someone has more knowledge about that part of the plugin, (s)he shared it with the others to speed up and streamline the process. Thanks to this, I personally felt very comfortable while asking questions about the problems I encountered. We had a very transparent communication and this created a trust environment in the group.

Intragroup Conflicts Encountered

When we consider the Tuckman’s model, we were very lucky about the forming of the group development. Irem and I are sisters and Tudor and Tommaso knew each other before. Therefore, the process of becoming familiar with each other was easy and quick for us. We also did not encounter any personal conflict in the group. There was the cooperation within the group and everyone was pleasant to each other.

Since the project duration is short, the issues that need to be done for each week were obvious. Also, we can call a group of 4 people as a small group. Since issues are clear and the number of people is not high, everyone got a chance to choose the issues that they were interested in and felt comfortable with. We set challenging but possible goals during the sprint iterations. There were weeks when I and my groupmates could not finish the issues of that week. But no one did not blame the others for it, and the ones who could not finish their issues finished them as soon as possible in the next week.

Like all software projects, everything did not always go smoothly. We had minor conflicts during the code review process. Merge requests were sometimes problematic for us. The first problem we encountered was the accidental deletion of a test of approximately 800 lines during the merge request. After one of us realized the issue, we quickly discussed the reason causing this issue in the team to solve it. Then, the team resolved the problem as quickly as possible. After that, we showed an extra effort to not live such a problem again. Another problem we experienced was the late approval of merge requests. Sometimes an opened merge request was not approved or merged for days. But we all might have different tasks and not always have time for code review. This is an understandable reason. Therefore, this problem never caused conflict within the group. I believe that we have completed this code review process successfully by reminding each other these merge requests more frequently.

Conflict Resolution Techniques Used

The cohesiveness of our team was good enough. This allowed us to prevent some problems from occurring in advance. For example; before one of our group members take an important step, (s)he would usually first ask it in the group chat and take everyone's opinion about it. In this way, we were able to prevent some problems and had a less problematic process.

If a problem or conflict occurred, we usually resolved it within a short time. The most important factor in this was our transparent communication. We all have different characteristics, but we all tried to be as honest and helpful as possible during the project. Personally, it was a very efficient process and I learned a lot from my groupmates. Another factor that allows us to experience a small number of conflicts is the task division. Nobody took an issue (s)he did not want to do. Everyone took the issues they are interested with and this enabled us to perform well. Each person's contribution to the project is clear.

In summary, we have achieved good harmony within the group. As a result of the discussions we have in our weekly meetings, all team members had the same expectations from the issues during weekly sprint iterations. Everyone did their best to meet

these expectations. During this period, we tried to solve the encountered problems as a group, not individually. And this is done in a short time, as quickly as possible. Since there was no intrapersonal conflict, we established an efficient and transparent communication within the team. Thus, we experienced almost no big and important conflict in our project.

A.3 Reflection Report of Irem Ugurlu

This section of the appendix incorporates the reflection report of the student Irem Ugurlu, as part of the deliverable for the Project Skills module of the Context Project course.

Introduction to Project Team

I, Irem Ugurlu worked in a group of four Computer Science students for the course Software Project. Software Project is the final course of the second-year Computer Science Programme of the Delft University of Technology. In my group, I have 3 other group mates: Ceren Ugurlu, Tommaso Brandirali, and Tudor Popovici. We have worked together for 10 weeks starting from 20 April until 28 June 2020. Our project was assigned to us by the company JetBrains, which is a company that creates software development tools.

The goal of our project was to solve the problem of broken links inside Markdown files. We have created an IntelliJ Idea plugin to solve this problem. After running the plugin, the user can see all of the links contained inside a markdown file. The component groups these links as "Changed Links", "Unchanged Links" and "Invalid Links". If a link is valid but changed, so it is broken, the user sees it as changed and can update it with the right link which the component recommends. If it is not broken user interface shows it as unchanged and if it is invalid it is shown under "Invalid Links".

Task-related Conflicts Encountered

We are not a group who had lots of issues during our development process so we did not have so many conflicts. Cohesiveness in our group was quite good. But of course, there were some moments that we had different ideas in the group. These were generally in the brainstorming stage. But I can say that none of the conflicts we had was unhealthy conflicts. If I go chronologically, I can give an example from the start of the project. Our client gave us a problem at the beginning which is solving the problem of broken links inside markdown files, but they did not specify what kind of product we should create. So we had a lot of flexibility while we decide on the component we are going to create. So this flexibility, of course, created some conflicts, we had lots of different ideas and finally decided to create a plugin. Because of this, our brainstorming part took more time than expected and we have started programming later than expected.

Another task-related conflict we had was about using a programming static analysis tool. Our client has requested us to follow a basic static analysis tool. Then we have decided to use Kotlin Linter and put it to continuous integration. But later on in the group some of us did not want to put it to CI because checking it each time could be problematic. As a result, we have removed it from CI, and when we were close to the end of the project. We have started to follow it again.

Lastly, another conflict we have could be our ideas during determining the issues. Sometimes, we had different ideas about how a feature should be but this is normal in a group. Because this is a decision-making task and sometimes there were no right answers. But these were not big conflicts and this is because all of us in the group were respectful to each other's ideas. We have trust in each other and when we assign a task to each other we knew that (s)he is going to do her/his best. Because of this trust, our communication was very transparent and this created a comfortable environment inside the group.

Intragroup Conflicts Encountered

Inside our group, we did not have any personal conflicts between group members. This is probably because of the nice environment inside the group. In my opinion, this has two main contributors: the first one is that when forming our group Tudor and Tommaso were knowing each other before and I and Ceren are sisters. So the team was not completely foreign to each other. Also secondly, we all were kind to each other, we always tried to be nice and respectful during our communication.

Another factor that can cause intragroup conflicts would be task splitting. But fortunately, we did not have any issues with this, because we are already a small group, just four people. Also, we split our tasks during our weekly sprint meetings. Noone took a task which (s)he did not want. I think this has a big impact on efficiency. Also, this made our tasks determined and we tried to keep the balance. Of course, as it happens always, there were some moments some of us could not finish their tasks for that week, but in those situations, we politely mentioned these. So, no one is offended and we finished our tasks as soon as possible.

As it happens in all software projects, everything did not always go perfectly. We had minor conflicts during the process. These were generally about the merge requests. I can come up with two examples. The first one was in the first weeks of the project. I had a ticket about converting relative links with dots into the correct form. When the task is completed I have created a merge request. But one of my group mates did the same task in his branch in a better way. So that merge request is merged. And mine is not. That was one of the few merge requests which we did not merge. Lastly, if I give another example to task-related conflicts our group was generally late about merging the merge requests, one of my merge requests is merged after one week. This is of course caused by the additions and fixes to feature but even after it is completed I had to wait 3 or 4 days for it to be merged. But these were small conflicts and beyond these, I think that we successfully managed and completed our development process with our group.

Conflict Resolution Techniques Used

I believe inside our group we managed to solve the problems before they get bigger and they did not cause any big conflicts. I think the main reason behind this was the big commitment and effort of the group members. Because all of us tried to be as efficient as possible.

Even though we had a conflict, it was easy to resolve it inside the group. Because we had a transparent communication between team members. When we had a problem or a decision to make we always made meetings and everyone presents their ideas. We were also respectful of each other's ideas. We also had autonomy inside the group, when someone had an idea to resolve the problem, other group members had the trust of that member.

A.4 Reflection Report of Tommaso Brandirali

This section of the appendix incorporates the reflection report of the student Tommaso Brandirali, as part of the deliverable for the Project Skills module of the Context Project course.

Introduction to Project Team

For the 2020 iteration of the CSE2000 Software Project I, Tommaso Brandirali, was assigned to work with Tudor Popovici, Irem Ugurlu and Ceren Ugurlu on the project "Tracking Changes in Links to Code". The project was offered in collaboration with JetBrains s.r.o., a software company mainly known for the development of IntelliJ IDEA, a popular open source integrated development environment for the Java Virtual Machine, and various other IDEs for other languages based on the same framework. JetBrains' IDEs support the installation of plugins to extend the IDEs' functionalities.

The project's aim was to implement one of such plugins for IntelliJ. The plugin's goal was to solve the problem of obsolete links in code documentation: that is when links in documentation files, such as Markdown files, are not updated as the resources they point to change, and therefore become invalid. Maintaining links in documentation up to date is a time consuming and non productive task that the plugin should automate and facilitate. The basic functionality required from the plugin was: scanning `.md` files, parsing links within them and identifying their linked resources, tracking such resources throughout the version control history of the project, detecting changes in linked resources that could have invalidated the links, and suggesting replacements for the links.

The project lasted about 10 weeks in total, which is not much time to build group trust and entitativity from scratch. Nevertheless, I believe we achieved a satisfying level of performance and teamwork.

Task-related Conflicts Encountered

Due to the characteristics of the project, we did not encounter a significant amount of tasks that were likely to generate conflict. To use Tuckman's Stages Of Group Development: during the storming phase, which is most likely to cause conflicts, we were still busy planning and brainstorming how to structure the project, so we didn't encounter significant conflict. Since the project's goal was defined externally by the client, and since the development's stages and process were defined by the course's structure, there weren't many significant decisions to make that would have made room for disagreement to arise. Additionally, the project did not have room for competitive tasks, which also could have generated conflict.

Nevertheless, some minor conflict arose as it is natural. During the planning stage when it was necessary to choose the tools to use for the build system and continuous integration, there was disagreement on whether to use a linter or not. Some members of the team believed it was important to have a fast feedback loop for code quality, while others believed it was too time consuming to fix all styling errors before every commit. A compromise was eventually found which consisted in not using the

linter during the first stage of development, when code is rapidly evolving and it is more important to get basic functionality done, and introducing it in a later stage of development once the codebase became more structured and final.

In other cases it happened that a team member's tasks were highly dependent on the results of another member's ongoing tasks, which created some confusion and discussion on which issues should be prioritized and whose responsibility it was to get a specific task done. Flexibility from all team members helped put these discussions behind us and all such discussions were promptly deescalated and a solution was found within the same sprint. One factor that made task planning difficult was the unpredictability of the time required to perform tasks that heavily dealt with the IntelliJ platform, as it requires to research solutions through the IntelliJ source code. Again, flexibility was very helpful to solve such problems.

Intragroup Conflicts Encountered

Regarding intragroup conflicts, once again the structure of the project naturally encouraged a horizontal organization of the group, which mitigated any conflicts of power or control. For most of the project we simply had to achieve our common goals and there was not much place for power dynamics. Having at least partially chosen our groupmates (I knew Tudor from before the project, Ceren and Irem are sisters) helped avoid hostility within the group. Additionally, the small size of the group, just four people, meant that the productive contribution of each and every member of the group was necessary to deliver the product within the deadline, and that each member's contribution was easily identifiable.

This, of course, does not imply that we achieved 100% productivity, that would be impossible. Personally I will admit that I did not always give the best of my efforts to the project. Due to having other commitments outside of the project I had to divide my time between different activities and sometimes that meant a delay in the delivery of my tasks. I may have been the social loafer in the group at times. When there was a delay in my task deliveries my groupmates politely pointed that out in a constructive way and I increased my efforts to meet all of my responsibilities within the deadlines.

Another occasion for intragroup conflicts were the few accidents or mistakes which slowed down the development at times. Mostly these were related to the pull based development workflow, such as merge requests wrongly deleting lines and failed CI pipelines. Such things happen normally in any work team, and they risk creating conflict because people naturally tend to try and assign blame for errors. It is a natural way of coping with accidents, but it is not conducive to positive interpersonal relationships and surely not to teamwork. Luckily we always responded to such events with calm and worked on solving the problem instead of assigning blame. We succeeded at keeping problems on a professional level and not bringing them to a personal level.

Conflict Resolution Techniques Used

Throughout the project, when there were conflicts, they were easily managed and resolved within a short time. The team members had different levels of assertiveness in dealing with conflict, but all of us were highly cooperative in the resolution of such problems. We were successful at deescalating conflicts when they arose.

In the first place, frequent and transparent communication was fundamental in catching problems before they even became conflicts. Oftentimes the moments that are more prone to generating conflict are the ones when there's uncertainty or confusion about the steps to take or a decision to make. In such moments, we always communicated with each other before taking decisions, so that we would all be conscious of exactly what was going on with the project and none of us would find out that a decision was made after such a decision had been implemented.

When conflict was inevitable, all members of the team proved themselves cooperative and tried to solve the conflict through collaboration. When collaboration wasn't immediately successful, a compromise was found that allowed us to continue to work productively, as it was the case with the discussion about the linter.

Appendix B

Project Infosheet

Title of the project: Tracking Changes in Links to Code

Name of the client organization: JetBrains

Date of the final presentation: 28 June 2020

Name and affiliation of the client: Vladimir Kovalenko, Senior Researcher in the Machine Learning Methods in Software Engineering group, JetBrains

Name and affiliation of the project coach: Prof. dr. Arie van Deursen, Head of the Department of Software Technology, Delft University of Technology

B.1 Description

Over the course of the last 10 weeks, our group has worked on an assignment given by JetBrains, a world leading vendor of professional software development tools. JetBrains is well-known for building numerous high-quality Integrated Development Environments, out of which, arguably the most popular is the IntelliJ IDEA, which targets programming languages running in the Java Virtual Machine.

The core challenge of the project was to create a plugin from scratch for IntelliJ IDEA, which helps to automate the problem of maintaining links to code. The application's goal is to automate the mundane and often forgotten task by the developers of updating links targeting code once changes have been made to the code targeted. The application should mainly work on relative links and links targeting code hosted on remote repositories, within the markdown files in the currently open project.

The first two weeks of the project have been dedicated to the research study related to how code targeted by links could be tracked. The results of this research indicated that using Git to retrieve the history of a project over time would be a viable solution to detect changes in links. Furthermore, after Git has been decided to be used as a way to retrieve changes, there was only one problem left to solve via research: how could lines be tracked between the versions of a file. The answer to this question has been found by comparing multiple researched line-mapping techniques' performances. The most viable solution in terms of accuracy has been proven to be the LHDiff algorithm. After the research questions have been answered, the team started designing the product, followed by implementing it.

Our team has used Scrum as a software development framework, hosting the code on a TU Delft hosted GitLab repository. Sprint cycles of one week have been used throughout the development process. Testing of the product has been done mostly by means of manual testing throughout the development process, as setting up automated tests was proven to be quite difficult, due to the amount of dependencies of the plugin on the IntelliJ platform. However, some developers of our team have been able to finally set up the testing infrastructure, and as a result integration tests have been added to an automated testing suite. On top of this, unit tests were performed wherever deemed necessary.

The final product consists of a plugin that fully automates the maintenance of relative links and links targeting web-hosted repositories that correspond to the currently open project. The user is capable of running the action of the plugin, which detects all of the links defined in the `.md` files, tracks the changes that have affected the code targeted by those links, and based on the retrieved changes automatically generates a newer link as a replacement to the old one. The plugin incorporates an intuitive user interface, with the help of which the user can update the links with their newer counterparts. The user can perform a batch update, by selecting multiple links at a time to be updated.

Currently, the application has been brought to an implementation level that would allow it to be used in experimental mode. Some things still require to be tested that they work as intended, performance improvements would also have to be addressed and new features would still have to be added in order to bring the plugin into production. Our team has made recommendations to the client, that target most of the aforementioned issues, such as adding the feature of tracking changes in links to web-hosted repositories that do not correspond to the open project, testing the product using more integration tests, solving the performance issues and modularizing the code such that the source code of the plugin is easily portable to other relevant environments.

The client has directly expressed his satisfaction with the end product. Furthermore, the client has also indicated that the source code of the plugin might be later used in other areas of the IntelliJ platform, in relevant contexts.

Appendix C

Division of Labour

This chapter presents a mirror-like table of the GitLab issue board of the project. It contains two columns: one for the task description and one for the developer's name assigned to the task.

Task Description	**Assignee**
Create basic version of ChangeTracker	T. Popovici
Implement functionality for retrieving commit at which a line containing a link was created	T. Popovici
Research line mapping techniques	T. Popovici
Make tracking links to files more robuts	T. Popovici
Implement functionality for getting the web reference type	T. Popovici
Add functionality for running git commands on specific branches	T. Popovici
Fix link updater not comitting PsiTree after updating links	T. Popovici
Use GitHub API to retrieve changes for directories	T. Popovici
Add support for other syntaxes for links in markdown files	T. Popovici
Improve settings page	T. Popovici
Track a single line across multiple versions of a file	T. Popovici
Track multiple lines across multiple versions of a file	T. Popovici
Add tests for LinkFactory class	T. Popovici
Check that a referenced line is not negative for links to a single line	T. Popovici
Improve get start commit method in GitOperationManager	T. Popovici
Get diff of file with working version of a file	T. Popovici
Write higher level documentation of project	T. Popovici
Add javadoc comments on methods and classes	T. Popovici
Allow get start commit method to work on different markdown syntaxes	T. Popovici
Refactor long backend methods	T. Popovici
Allow tracking lines / directories on specific branches	T. Popovici
Setup CI/CD pipeline for the repository	T. Brandirali
Implement basic framework for storing state of plugin	T. Brandirali
Define APi of Core module	T. Brandirali
Implement basic structure of Core module	T. Brandirali
Format new links according to the formatting of their original link	T. Brandirali

Task Description	Assignee
Setup integration tests	T. Brandirali
Update UI of progress indicator	T. Brandirali
Handle path separators differently for different OSs	T. Brandirali
Add integration tests for updating links	T. Brandirali
Prevent changes to files after scanning from breaking link offsets	T. Brandirali
Setup and prepare LaTeX files for final report	T. Brandirali
Setup changes for parsing changes	T. Brandirali
Improve performance of costly git operations by implementing a cache	T. Brandirali
Add a listener for git actions that would invalidate the cached results	T. Brandirali
Create a basic version of UI for plugin	C. Ugurlu
Add buttons to update links	C. Ugurlu
Implement functionality for tracking directory paths	C. Ugurlu
Restructure UI layout	C. Ugurlu
Make UI for settings	C. Ugurlu
Add sanity check for links to lines	C. Ugurlu
Make tracking directories more robust	C. Ugurlu
Run Kotlin Lint and resolve errors	C. Ugurlu
Fix bug when parsing changes for links with whitespace	C. Ugurlu
Tool Window Icon should be 13x13	C. Ugurlu
Allow multiple links to be updated one after the other	C. Ugurlu
Remove a link from the UI after it is updated	C. Ugurlu
Test the changes data classes	C. Ugurlu
Test the links data classes	C. Ugurlu
Add dialog popup to inform the user of updating a working tree change	C. Ugurlu
Test retrieving directory changes in ChangeTrackerServiceImpl	I. Ugurlu
Test retrieving file changes in ChangeTrackerServiceImpl	I. Ugurlu
Test retrieving line changes in ChangeTrackerServiceImpl	I. Ugurlu
Update links to multiple lines	I. Ugurlu
Show after paths in the UI	I. Ugurlu
Add checkbox to UI	I. Ugurlu
Fix check relative link on corner cases	I. Ugurlu
Implement functionality for retrieving versions of a file between two commits	I. Ugurlu
Add support for links with whitespace	I. Ugurlu

Task Description	Assignee
Plugin should support relative paths containing ../	I. Ugurlu
Parsing functionality for different types of links (lines, directories, files)	I. Ugurlu
Basic link retrieving functionality from markdown files	I. Ugurlu
Create IntelliJ plugin project setup	I. Ugurlu
Track working tree changes to files doesn't detect moving file to it's original location	I. and C. Ugurlu
Add method to update a link to a line in LinkUpdaterService	I. and C. Ugurlu
Classify relative links with single/double dots correctly	I. and C. Ugurlu
Improve tool window page UI	I. and C. Ugurlu
Update links when button is pressed	I. and C. Ugurlu
Practicing Kotlin with Kotlin Koans	All team

Appendix D

Original Project Description

This appendix incorporates the original description of the project, as formulated by the client upon the submission of the project.

Copied Original Description

We are JetBrains. We build smart IDEs (IntelliJ IDEA, PyCharm, WebStorm) and collaboration tools (YouTrack, TeamCity, Space) to help individual developers and teams build software more efficiently.

This project is presented by the ML4SE research group at JetBrains Research.

During the lifetime of a software project, it is natural for code to change and be reorganised. Links to source code, either from within the project (e.g. Markdown files with documentation) or from other sources (e.g. issue trackers) may become obsolete when code changes.

The goal of this project is to build a component capable of tracking changes in links to source code files and individual lines, and assisting with their maintenance by automatically proposing link updates in relevant contexts.

The resulting component should be system-agnostic, have minimal external dependencies, and be easy to integrate into any relevant tool, be it an IDE or a code review system. We would strongly prefer it to be implemented in Kotlin.