

Tracking Changes in Links to Code

JetBrains

Ceren Ugurlu

Irem Ugurlu

Tudor Popovici

Tommaso Brandirali

Table of Contents

Table of Contents	1
Problem analysis	2
1.1. Motivation	2
1.2. Problem statement	2
1.3. Deliverables	3
1.4. Issues to encounter	3
1.4.1. Defining the types of links to be tracked	3
1.4.2. Detecting changes in/of files between component runs	5
File tracking	5
1.4.3. Component assumptions	6
1.5. Identifying the stakeholders	7
Feasibility study	7
2.1 Technical Feasibility	7
2.2 Financial Feasibility	8
2.3 Market Feasibility	8
2.4 Resource and Time Feasibility	8
2.5 Risk Feasibility	8
Requirements	9
3.1 Functional Requirements	9
3.2 Non-Functional Requirements	10
Project approach	11
4.1 Development Methodology	11
4.2 Preliminary Design	12
Programming Language	12
Framework	12
4.3 Quality Assurance	12
Documentation	12
Testing	13
Version Control	13
General planning	13
References	14

Problem analysis

1.1. Motivation

Links to code can appear in any description, anywhere. They can appear in documenting software, in descriptions of software tickets, in software tutorials and many other places.

To illustrate the importance of valid links in descriptions, we will take a look at documenting software. Software projects are most often made up of a large number of directories, files and breaking it into even smaller pieces, lines. All of these elements play a role in the big picture of the project and this role is usually explained by documenting that specific element. Most often, inside of documentation, there will be links to files and lines for which an explanation exists. But code changes. It can change substantially, be it via refactoring, adding new functionalities and many other reasons.

Without updating the documentation after each change to the project that would affect it, it can happen for numerous links to code inside documentation to become invalid; either because the link is referencing a file or line that does not exist anymore and/or because the description that accompanies the link does not correspond to the contents given by the link.

According to [1], “Poor documentation causes many errors and reduces efficiency in every phase of a software product’s development and use”. Obviously, broken links inside documentation would result in documentation of a lower quality and thus have consequences on the overall development of the product.

1.2. Problem statement

We are planning to build a system-agnostic, language-independent component capable of tracking changes in directories, files and lines that are being referenced by links inside of descriptions and check whether after these changes are applied, the links are still valid; in the case where the link is not valid anymore, the component should suggest a new link that would correspond to the new state of the element, where this is applicable (the element could have been removed).

The component should be easy to integrate in all relevant environments, be programming-language independent and scale up to projects with a large number of links and files.

1.3. Deliverables

The deliverables would consist of:

- A component capable of tracking changes in links to code, inside of an IDE environment (IntelliJ IDEA), in the form of a plugin. This plugin will track relative and web-hosted repository links to code.
- Depending on the timeframe, a component using the same core logic as the one for the IDE environment, but adapted to work in any environment: tracking links to code in web-hosted repositories.

1.4. Issues to encounter

1.4.1. Defining the types of links to be tracked

We will strictly refer to links to code in the following explanations. Therefore, let us examine the formats which a link to code can take, based on the environment in which it appears:

IDE environment

Links can be defined in markdown files. They can also be defined in documentation comments of many programming languages, such as JavaDoc for Java [2]. When defined in markdown files, they follow the formats imposed by the markdown syntax [3]. To define a link you have to specify 2 components, the link text and either a relative path or a link to a web-hosted repository.

Links to code can be either regular web-hosted repository links or relative links:

a. Remote web-hosted repository links

Links to remote repositories are of the form *https://platform/name/project_name/blob/commit/path*, where:

- “platform” corresponds to the name of the platform followed by domain (e.g. github.com)
- “name”, “project_name” correspond to the name of the web-hosted repository owner and project name is the name of the project that is linked
- “blob” is a constant, it is the same for each link
- “commit” represents anything that can be mapped to a commit: be it either a branch name, a tag name or a commit SHA [4]. When specifying a branch name, the link will point to the contents of the path at the latest commit (where HEAD is pointing at).
- “path” represents a path to a directory or file. It can also point to a specific line or block of 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.

When the “commit” part of the URL is a commit SHA, the link is called a permalink [4]. The link will always point to the version of the path corresponding to that commit, no matter how many

changes have been applied that would have affected that path since that commit. But in the case where 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 [4].

Possible problems?

If the link refers to the project that is currently open in the IDE editor, we can use the versioning system functionalities of that editor to retrieve changes (see section 1.4.2 for details). Otherwise, the project along with its changes would be a bit out of our immediate reach. In this case, we would have to make calls to platform specific APIs (e.g. GitHub’s API) to retrieve information about the versioning of that project.

b. Relative links

Relative links listed in an editor (e.g. “[link text](my_path)”) will link to a path relative to the markdown file’s location in which that link is listed. These relative paths can reference files, directories, lines or a block of lines. Lines and block of lines are referenced in the same way as for regular web-hosted repository links: you would need to append to the path of the file #L21 to reference line 21 in that file or #L21-L25 to reference lines numbered 21 up to and including 25. However, once that markdown file is pushed on a web-hosted repository, the file will be converted to an HTML file and that relative path will become a relative link of the form *platform/name/project_name/blob/commit/path* [5]. The components of this URL will have the same meaning as for a regular web-hosted repository link, with the exception of the “commit” and “path” part. The “commit” will have as a value the branch name of the currently checked out branch in the web-hosted repository. The “path” will have the value provided in the editor for the relative path.

This means that relative links listed in the editor will always refer to the latest commit version in the currently checked out branch.

Due to the fact that these types of links refer to the HEAD of the currently checked out branch, these types of links will be susceptible to change and should be tracked by our component.

Web-hosted repositories

There are 2 ways to link to code from web-hosted repositories themselves: using regular web-hosted repository links or relative links.

Outside the IDE and web-hosted repositories

Outside the IDE and remote repositories, you can only reference code via regular web-hosted repository links. To track changes in the links inside these environments, we would have to implement our own functionality of retrieving the changes for the project that is linked, by using platform specific APIs. Tracking changes in these environments corresponds to the case of tracking links to projects that are not open in the editor inside of the IDE. Therefore, by tackling that problem in the IDE environment, we could also solve it for these environments.

1.4.2. Detecting changes in/of files between component runs

A problem that we would need to solve would be detecting the changes that happen between consecutive component runs in the elements that are being referenced by links inside of a project, along with which operation caused these elements to change, so that we can update the links accordingly.

To be able to detect changes between runs, we need to have access to snapshots of the project between each run of the component, such that we can compare these snapshots, and see the differences between them. Inside of the IDE, this information could be retrieved by accessing the IDE's functionality for versioning systems. Outside of the IDE, we would probably have to resort to calling the API of the platform that hosts the element being referenced.

File tracking

Using this type of information, we would be able to track changes in files within a project that is using a versioning system. In the case of tracking links to files, we would be able then to get a straightforward summary of the changes in the files between commits.

Line tracking techniques


However, for lines, we would like to get a concrete mapping of each line from the old location to the new location in a file. This can be done by comparing the previous version and the current version of the file in which the referenced line/lines reside in. In order to concretely map a line to its new location, we can make use of existing language-independent algorithms that tackle this problem [6, 7, 8, 9, 10, 11].

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

Line tracking techniques performance comparison

Asaduzzaman et al. [9] provide a comparison between LHDiff and other line tracking techniques, such as ldiff [6], git's blame command [7], W_BESTI_LINE [8], sdiff (which is a language-dependent technique, and therefore we will not consider it for this case). Their experiments have shown 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 for the "Reiss" benchmark. The 3% incorrect results are broken into 42.5% of the algorithm being able to find a mapping of a line but the mapping is not correct, 57.5% of the algorithm detecting the deletion of a line but the line still exists in the file and 0% of the algorithm detecting mapping of a line but that line is deleted. Time-wise, LHDiff took 4 seconds on the tests that were performed. Compared to other techniques time results and taking into account the overall good performance of this method, we can say that LHDiff would be a suitable line tracking solution.

Reiss [8] discusses numerous other line tracking methods that work across versions of files, listing these methods into four categories: those "that match the line in question, those that textually match the local context around the line (but not the line itself), those that match the abstract syntax tree for the line in question, and methods that use whole file information". Their research papers also provide experiment results that prove the efficiency of each method described. As part of the experiments, methods have also been combined, to provide even better results. We will take a look at the results of



the combined methods, because these are performing generally better than the methods on their own. Based on their experiments, variations of W_BESTI methods, W_BEST_LINEI, W_SMARTI_LINEI_METHOD are promising techniques all with over 97% correctness accuracy.

Other promising approaches [10, 11] are presented by Canfora et al. which provide an overall accuracy of 96% respectively 92%, based on their experiments.

1.4.3. Component assumptions

The component will run before each commit

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

Previous run will leave the links in a valid state

Between component runs, we are making the assumption 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.

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

When our component is run for the first time by a user, which links can we declare valid and which not? We would essentially need to trace back the snapshot of the project that corresponds to the moment the link was created. That is how we can see the real meaning of the link that we are checking. After that, we can follow commit-by-commit the history of that/those line(s)/file up to the current commit. This way, we 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, we can check the validity of each link and suggest a new valid link to the user.

Storing link state and last run information

We need to store on disk some information that would tell us when our component was last run. Also, to avoid doing the costly operation described above on each component run, but solely when the component is first run, we could persist the state of each link on disk, starting from the first run. The subsequent runs will use the previously stored information for their computation of the validity of the links and so on. A good option would be to set a similarity threshold value for lines/files, above which the links are automatically updated. Values below the threshold would result in asking for user's permission before updating the links.

1.5. Identifying the stakeholders

We have identified the following stakeholders:

1. The client - the component will be used in different platform products developed by JetBrains.
2. The group of platform users, mainly software developers from any level of experience working on collaborative projects requiring documentation - that will be using the functionalities of this component once it is integrated in a specific platform.
3. The users of the software produced by the software developers that will use our component.

Feasibility study

In projects, the feasibility study is applied to determine the applicability of an idea. For example, it ensures whether a software project is technically feasible as well as economically justifiable or not. It gives information about the worthiness of a project investment. Sometimes, a project may not be doable. There can be plenty of reasons for this infeasibility such as technical, financial, resource or time related issues. In this section, we will examine the feasibility of our project in many different aspects.

2.1 Technical Feasibility

In our research, we found a project which is similar to our project [12]. The project is written in Javascript language. It is a plugin to validate Markdown links and images reference existing local files and headings but it does not check for external URLs. For this functionality, it detects local Git repositories. The similarity between ours and this project is they both aim to track changes in markdown files. If we compare them, our component will have more features such as checking the changes of external URLs and lines. Also, it will support different environments and languages.

The main tools and technologies assessed with our project are:

- Kotlin
- VCS integration of IDE
- JetBrains' IDEs

Each of these technologies is easily accessible and all of the required technical skills for these technologies are manageable by our programmers. We calculated all existing and possible future limitations with our team. Based on these limitations, we endeavoured to choose the most appropriate tools and technologies for our component.

With these, it is clear that our project is technically feasible.

2.2 Financial Feasibility

This project is the collaboration of JetBrains and TU Delft. There are 7 people directly involved in the project. These people will spend their time and effort on the project in this process. Therefore, there will be a certain amount of payment to them by the company or the university. The distribution of responsibilities of these 7 people and the amounts paid to some of them are as follows:

- 1 client contact person
- 1 TU Delft coach
- 1 teaching assistant
- 4 students (monthly 500 euros per person)

As a result, the project is not completely free. But both the company and the university are capable of compensating this cost. Therefore, we can say that the project is financially feasible.

2.3 Market Feasibility

This section is about the market feasibility of our project which declares whether our product is something that people want or not. This feasibility type ensures that we do not build a product no one needs or wants.

Since JetBrains is a software development company whose tools are targeting software developers, the main goal in products is to improve the functionalities in tools. We will try to solve a very common problem, the issue of outdated links and references in markdown files, in this context. Namely, if this works the target users will get a better product. As a result, our product will offer beneficial functionality for users and we can state it as market feasible.

2.4 Resource and Time Feasibility

We have roughly two and a half months to complete this project. It is somewhat enough for our development process. Resources required for our project are the following:

- Programming device
- Programming tools (freely available)
- Programming individuals

Since we have enough time and everything required, the project is feasible regarding the time and resource aspect.

2.5 Risk Feasibility

There are multiple risk factors affecting risk feasibility during this project. In this section, we will discuss the most important risk factors and how we could handle these factors.

- a. **Kotlin:** Our first risk factor is our decision to use Kotlin as the programming language for our component, even though we are all unfamiliar with it. However, we do not think it will lead to a

problem because Kotlin is a language that is very similar to Java [13]. Additionally, we are all familiar with Java.

- b. **Optimizing solution:** Our second risk factor is the probability of the high complexity of our component. We will focus on optimizing our solution to prevent this risk. Since we check each change on files and give suggestions based on the evaluation of these changes, this can cause a high run time and complexity. We also cannot use any algorithm we want because of these reasons. We aim to find the most efficient algorithms for our project to optimize it. For example, we will use the diff feature in VCS integration of IDE for checking changes in files and lines. Because we found it less complex when we compared with the others.
- c. **Supporting different IDE's:** Our third risk factor is the adaptability of our component with other environments. We will start by creating a component that is working well in IntelliJ Idea. Then, our goal is to develop our component in a way that it does not depend on the environment or programming language. It should work well in any programming language or environment. It will be challenging for us because if we want to use our plugin in all IDE's we should use dependencies and libraries which are available in all environments. Accordingly, we need to consider this factor during the whole process.

Requirements

In the context of this project, we researched the requirements for our product with the client as our main stakeholder. Since the scope of the project as defined by the client was arguably vague, we evolved the requirements in parallel with restricting the scope of the product research and development.

The final product should be a plugin, with a core, system agnostic logic, and modular implementations for different environments. During this project we will focus on an implementation for the IntelliJ IDEA platform, but the structure should be modular to allow extensions for different environments.

3.1 Functional Requirements

The requirements will be described 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; Won't Have, for least critical or low-payback features that won't be included in the release unless rediscussed at a later time.

We will list the following requirements referred to a generic implementation of the component as an IDE plugin and applied to a software repository.



1. Must Have:

- a. The component must be able to detect links to files, lines and directories in markdown files within the open project.
- b. The component must be able to detect between component runs changes in files, lines or directories referenced by a tracked link.
- c. The component must be able to detect changes to files between different runs. Possible cases:
 - i. Renaming a file.
 - ii. Deleting a file.
 - iii. Moving a file to a different directory.
 - iv. File content modification.
- d. The component must be able to detect changes to single or multiple lines within files between different runs.
- 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 component must inform the user about the broken link and not try to automatically update it.
- f. The component must work on projects using Git as a VCS.
- g. The component must track links to web-hosted Git repositories that do not correspond to the local repository on which the component is run.

2. Should Have:

- a. The component should work on projects using different VCS, such as SVN and/or Mercurial.
- b. The component should be able to track links from JavaDoc comments.

3. Could Have:


- a. The component could support reading links to code in the project from outside the project itself, like git issue trackers and external documentation.
- b. The component could track links to web-hosted repositories, other than Git repositories, that do not correspond to the local repository on which the component is run.
- c. The component could support IDEs different from IntelliJ, such as Eclipse.
- d. The component could support tracking of links from comments of other programming languages than Java.

4. Won't Have:

- a. The component won't have a generic implementation that stores snapshots of the project under which the component is run.

3.2 Non-Functional Requirements

We will now define the structural requirements. The component will be made up by a Core module implementing the basic logic, accepting standardized inputs describing the latest changes and producing standardized outputs describing the outdated or broken links. The Core will be used by implementations of the component for specific environments, which will take care of finding links in



and outside the project, tracking changes, feeding those into the Core and, if needed, fix broken links according to the Core's outputs. The Core module should have a single, environment agnostic implementation, while plugins using the Core should have different implementations for different use cases.

For example, an IDE implementation would consist of an IDE plugin, compatible with the IntelliJ platform, which would require a Link Tracker module to find all links to files and lines within the project. This implementation would also require a Change Tracker module to extract data about the latest changes in the project. The Change Tracker for an IDE module should make use of the IDE's VCS support. Finally, the IDE plugin would make use of the Core's functionality to produce the expected outputs.

We will now list further non-functional requirements of the project:


1. Must Have
 - a. The component must have read access to all files in the repository.
 - b. The component must store information about the version of the project on which it was last run.
 - c. The component must have an implementation compatible with the IntelliJ IDEA platform.
 - d. The component must be reasonably usable independently from project size and number of commits.
2. Should Have:
 - a. The component should use the VCS support functions of the IDE to parse changes.

Requirement 1.d is purposefully vague as its evaluation will be done by the client once the product is testable.

Project approach

4.1 Development Methodology

To have a nice and efficient development process, we chose to follow a development methodology during our project. Having this methodology makes us more efficient and helps with finishing our tasks on time. In software development, there are lots of methods available. The most popular ones are Scrum, Kanban and Waterfall. So if we explain each of them shortly: Waterfall is a traditional and linear method of explaining the software development process in software engineering. The drawback of this method is that going back to deal with the changes is not possible in this method. Scrum and Kanban both are processes that help teams to collaborate and work effectively. But Scrum has more transparency and visibility than Kanban and also Scrum is easy with changes. It accommodates changes. In our team, we want to be open to changes and further development so Waterfall and Kanban are not the best choices for us. Therefore, we decided to follow the Scrum methodology, which we think is the most suitable for us.



Lastly, Scrum is an agile methodology. It helps us to deliver our project early. Scrum works in iterations of a predefined length which is 1 week in our case. And in every iteration, new features/changes can be chosen to be addressed during the next iteration. This makes deadlines more clear for us. Another reason for choosing Scrum is we all have some experience using Scrum. So we will not need much time getting used to the methodology. In conclusion, we are going to apply scrum methodology but we will be flexible, we will not follow all of the scrum rules such as certain time estimates.

4.2 Preliminary Design

Before we started the development process, we had to make a lot of design choices. In this section, we will discuss our design options in the design process and explain our design choices.

a. Programming Language

The first design choice we had to make was about which programming language we will use. There were two options along us using Java or Kotlin. We preferred to use Kotlin because Kotlin is fully compatible with Java and it compiles with the existing java code. Also, Kotlin has advantages over Java. If we give some examples: Kotlin's syntax is not as verbose as Java therefore code written in Kotlin is very concise. It helps to reduce errors and bugs in the code. Also, one of our non-functional requirements is using Kotlin. There was just one drawback of using Kotlin: none of us in the group has previous experience with Kotlin so this can be counted as a project risk. But still, Kotlin and Java are similar languages and we are all familiar with Java.

b. Framework

Our main goal is to create our component in a way that does not depend on the environment or programming language. It should work well in any programming language or environment. But as we plan our process we have decided that we should start with a smaller task and then generalize it. So we have decided first to create a component that is working well in IntelliJ IDEA. After successfully completing this, we can extend our product to other IDEs and platforms.

4.3 Quality Assurance

In this section, we will talk about methods we plan to apply to have some certain quality in our code. We plan to do this by having the necessary documentation and testing.

a. Documentation

For documentation, we have decided to use just one platform so we will use GitLab. Having all of the documents in one specific place helps to reduce complexity. Also, we all can easily review each other's work in this environment. Another thing we want to have in our repo about documentation is having nice and clear instructions to run our plugin.

b. Testing

Testing is an important phase of software production. We will do both manual and unit testing. In general, our tests will be functional. We are going to use JUnit 5 as a testing framework. We are going to aim for an average of 70% test coverage. Another factor that is going to help to test our product is the quality feedback that we get from the client. In this way, we are going to make sure that the feature we have works as it is expected. In addition to these testing criteria, we are also planning to use some static code analysis tools to keep our code nicely styled. For this we are going to use IDEA and klint linter.

c. Version Control

We will use GitLab for version control in our project. We will manage our code base by using Git-flow meaning we will have a master branch. Each of us will create a branch from master when we need a new branch for a new feature. When we are done with that feature we will create a pull request to update master. We are going to determine our tasks by creating issues. These issues are going to define what we have to do. They need to be nicely defined. We will create these issues during our meetings with the team, in order to decide upon tasks and how to split them. Our issues will have the following labels: open, to do, doing, code review, testing, and closed. Every issue will have a responsible assignee or assignees. After we are done with the feature we are going to update the label of the issue as well.

To make sure of the quality of each other's work, we will give importance to code review. Also in our code review processes, if necessary, we will point the client. This helps to get the feedback of the client and to make sure that we are on the same page. Also having the client's feedback helps to make sure the quality of the feature and whether it matches the expected criteria. We will not do this in each request because then this will be bothering the client in unnecessary or simple operations.


General planning

To plan the development process we will be using an Agile methodology, borrowing organizational rules from Scrum and Kanban, although without following their strict requirements. We will proceed in sprints of 1 week, from Monday to Friday. On Mondays around 10am there will be a meeting with the TA and Coach to clarify the planning for the upcoming sprint and propose solutions to any problems that might have arisen in the previous sprint. On Fridays around 4pm there will be a meeting with the client to review the results of the past sprint. During the week the team may hold meetings as needed to confront any challenges that might arise. Meetings will be held virtually using Slack. We will be using various instant messaging platforms for communications within and outside the development team: Whatsapp for various informal communication within the team, Slack for communications with the client and Mattermost for communications with the TA and coach.

In week 18 we will start the development process by setting up a software structure and researching the specific tools and solutions we will use in the rest of the development. We will likely start by reviewing the code we wrote for the toy plugin for markdown files and find out which parts of that we can reuse for the purpose of this project. Active development will start in week 19. Week 25 will be the deadline for the delivery of all components of the project. The following two weeks will be dedicated to preparing a final presentation on the work done and the progress we will have achieved.

References

- [1] D. L. Parnas, "Precise Documentation: The Key to Better Software," *The Future of Software Engineering*, 2010, pp. 125–148
- [2] corochann. (2015, Oct 23). "Javadoc coding rule of @link, @linkplain, @see" [Online]. Available: <https://corochann.com/javadoc-coding-rule-of-link-linkplain-see-372.html> [Accessed 3-May-2020]
- [3] A. Pritchard. (2017, May 29). "Markdown Cheatsheet" [Online]. Available: <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet#Links> [Accessed: 29-Apr-2020]
- [4] Anon. (n.d.) "Getting Permanent Links to Files" [Online]. Available: <https://help.github.com/en/github/managing-files-in-a-repository/getting-permanent-links-to-files> [Accessed: 29-Apr-2020]
- [5] Y. Mendelssohn. (2013, Jan 31). "Relative links in markup files" [Online]. Available: <https://github.blog/2013-01-31-relative-links-in-markup-files/> [Accessed: 29-Apr-2020]. [Accessed: 29-Apr-2020].
- [6] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool," 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, 2009, pp. 595-598.
- [7] Anon. (n.d.). "Git blame" [Online]. Available: <https://git-scm.com/docs/git-blame>. [Accessed: 29-Apr-2020].
- [8] S. Reiss, "Tracking source locations," 2008 ACM/IEEE 30th International Conference on Software Engineering, Leipzig, 2008, pp. 11-20.
- [9] M. Asaduzzaman, C. K. Roy, K. A. Schneider and M. D. Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines," 2013 IEEE International Conference on Software Maintenance, Eindhoven, 2013, pp. 230-239.

- 
- [10] G. Canfora, L. Cerulo and M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories," Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007), Minneapolis, MN, 2007, pp. 14-14.
 - [11] G. Canfora, L. Cerulo and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach," in IEEE Software, vol. 26, no. 1, pp. 50-57, Jan.-Feb. 2009.
 - [12] T. Wormer, "Remark Validate Links," GitHub. [Online]. Available: <https://github.com/remarkjs/remark-validate-links> [Accessed: 29-Apr-2020].
 - [13] E. Sundin, "Perception and effects of implementing Kotlin in existing projects : A case study about language adoption", Dissertation, 2018.