Techniques For Easy and Efficient Manipulations of Large Codebases

Marko Ivanković
Goran Petrovic

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Ivanković, Marko and Petrovic, Goran, "Techniques For Easy and Efficient Manipulations of Large Codebases", Technical Disclosure Commons, (September 02, 2020)
https://www.tdcommons.org/dpubs_series/3574

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
Techniques For Easy and Efficient Manipulations of Large Codebases

ABSTRACT

Making a change to computer programs, especially those with a large codebase, is tedious and error-prone. This disclosure discusses mechanisms for refactoring the source code of a computer program easily and at scale. The techniques allow users to perform complex, across-the-codebase changes manually and enable automation of such changes. The user experience of code refactoring involves the user providing instructions via a code refactoring UI that specify the reform intent. The reform request is analyzed and broken down into smaller jobs such as running tests, generating diffs using backend tools, or creating change lists with the appropriate diffs. An estimate of the impact of the change is provided to the user for review. Finally, change lists are submitted to the codebase or other platform tools that help make the actual changes. The described techniques enable complex changes in a codebase via an easy to use user interface.

KEYWORDS

- Code refactoring
- Code restructuring
- Code transformation
- Codebase manipulation
- Codebase update

BACKGROUND

Making a change that affects many parts of a large codebase (that stores computer programs) can be an excruciating and tedious process. To make a change to an existing codebase, an engineer needs to:

1. Identify locations in the codebase that need to be changed
2. Effect the code change, creating a temporary state of the codebase (known colloquially as a "diff").

3. Submit the code change to the codebase.

An important problem in this process is that the codebase may *drift* between each step. For example, if other engineers (or automated agents) modify the codebase concurrently, the modifications can cause diffs to become obsolete or conflict with the additional edits. The bigger the codebase, the harder it is to make horizontal changes, both in terms of the number of lines in the codebase and the number of engineers that can edit the codebase.

In addition to drift, it is also difficult to effect the change for other reasons. If a change needs to be applied to thousands of points in a codebase, and the points are not 100% uniform, it is difficult to describe the change in a computer-readable format, e.g., that is suitable for automated codebase update. There is a substantial risk of under-specifying the code change request, which can cause changes to portions of the code that the engineer did not intend to change. Similarly, over-specifying a code change request can lead to not changing all points in the code that are to be changed.

**DESCRIPTION**

The techniques of this disclosure enable refactoring source code quickly and at scale. A user interface is provided that enables a user (e.g., an engineer with edit access to the codebase) to specify a precise and correct transform/reform request for a portion of a codebase.
Fig. 1: Example mechanism for refactoring source code of a computer program

Fig. 1 illustrates an example mechanism for refactoring the source code of a computer program. A primary concept per the techniques of this disclosure is a "reform." Reform is an intent from a computer engineer to change the codebase in a specific way. A user (102) can choose an intent from a list of predefined intents provided via a user interface (104). For example, intents can be clustered in three major groups as follows:

- **Code fix (106):** The user may choose to fix their project. Fixing code is conceptually at the highest level of code reform. Intents in this group do not require any additional input from the user. The user selects one or more intents he is interested in. For example, an engineer may choose "nicely format my code."

- **Change symbol (108):** The user may choose to change a specific symbol in the codebase. A symbol is a small piece of the codebase, e.g., a function or an object. To help the user choose the correct symbol, the UI provides the user with search and autocomplete functionality. Further, the UI provides an impact estimate (120) once a symbol is selected. Selecting a symbol is an essential task as it prevents over and under-specification. After the user selects a symbol, the user can choose from a limited set of
operations, e.g., "rename" and provide any additional data required by the action, e.g., the new name parameter used for rename operation, or other additional data as required by the corresponding selected operation.

- **Regular expression substitutions (110):** The user can choose to run a regular expression based code substitution across the codebase or parts of it. This substitution is the lowest level of code reform and is more dangerous in terms of possible updates to the codebase since it is relatively easier for a user to over or under-specify this type of change. Therefore, the UI provides in-depth impact estimates, e.g., including showing examples of the proposed change in the codebase, the count of impacted files, and/or expected resources required to effect the change.

Using the UI, the user can visualize their reform request on several source files that would be affected and can thereby ensure correctness. Additionally, potential compiler errors are reported immediately in the UI to help the user specify the reform correctly. For example, a user may need to rename a function in the codebase, add different parameters or an extra parameter to a function, etc. This is achieved by using a code index stored in a refactoring engine (112).

The refactoring engine is at the heart of the automated code reform system and forms the backend that supports the user interface. The refactoring engine curates a reform request that is submitted by a user. When a reform request is received, it is analyzed (114) and suggestions for further change are displayed in the UI. The user can choose to act on or ignore the automatically generated suggestions.

Once the user resubmits the intent, units of work (116) are prepared. For example, a single reform request may require two computer jobs (computer programs) run by one tool and a job by a second tool. Jobs may include running tests, generating diffs, and/or creating change
lists (CL) with the appropriate diffs. These jobs are sent to corresponding tools (122) by the refactoring engine. The progress of the jobs is monitored, and the progress report and health are displayed in the UI. An estimate of the impact of the change is provided to the user via the UI for review. Further, when code drift is detected (118), the reform intent is reapplied to the codebase. If there are any new changes in the codebase, the intent is reapplied multiple times until no code drift is detected. Finally, the change list is submitted to the codebase or to other platform tools (122) that help in making the actual changes.

Furthermore, one or more change lists (CLs) may be required depending on the scope and/or size of the refactor request. The user may face various outcomes following their refactoring. For example, all CLs associated with a reform request are successfully submitted; CL submission is in progress; or one or more CLs are reverted. The refactoring engine may rely on other tools for CL management.

CONCLUSION

Making a change to computer programs, especially those with a large codebase, is tedious and error-prone. This disclosure discusses mechanisms for refactoring the source code of a computer program easily and at scale. The techniques allow users to perform complex, across-the-codebase changes manually and enable automation of such changes. The user experience of code refactoring involves the user providing instructions via a code refactoring UI that specify the reform intent. The reform request is analyzed and broken down into smaller jobs such as running tests, generating diffs using backend tools, or creating change lists with the appropriate diffs. An estimate of the impact of the change is provided to the user for review. Finally, change lists are submitted to the codebase or other platform tools that help make the actual changes. The described techniques enable complex changes in a codebase via an easy to use user interface.