Efficient Refactoring in Industrial Projects

Whitepaper | V1.0 2020-03
# Table of Contents

1. Refactoring source code
   1.1 Essential and challenging
   1.2 Controlling cloning in industrial projects

2. Looking for code clones
   2.1 State of the art
   2.2 Introducing an industrial approach
   2.3 Code cloning detection proposal
      2.3.1 String-based textual cloning detection
      2.3.2 The token-based algorithmic cloning detection

3. Efficient refactoring strategies
   3.1 Static cloning filtering
   3.2 Dynamic cloning filtering
   3.3 Factorization Strategies

4. Field usage

5. Conclusion

Bibliography
1 Refactoring source code

1.1 Essential and challenging

Software projects are expanding on several fronts: source code size growth, multiple developing teams, diversity of coding languages and standards. Maintaining control over these projects requires limiting of code size explosion, tight monitoring of Technical Debt, and clear understanding of what should be monitored (new code) or not (legacy code).

In these conditions, the activity of code refactoring cannot be omitted as it helps regulate ever evolving code and avoid developing new code similar to already existing one. Unfortunately, the bigger the code base the more lengthy and hazardous refactoring can become.

To assist developers in this task, an automated solution should point to the best candidates for code refactoring. By using results from algorithmic cloning detection, Technical Debt computation, and code stability monitoring in a Continuous Integration chain, this solution should not only help optimize refactoring effort, but also focus on relevant areas first, all the while monitoring ongoing developments.

1.2 Controlling cloning in industrial projects

Software projects are prevalent in all industrial sectors, and produce as diverse items as services, tools, products, or embedded functions. Albeit a relatively recent field, software development has evolved at a rapid pace, and counts a multitude of programming languages and variants, many coding practices (good and bad), and sometimes overlapping coding regulations.

Controlling the source code quality is a required activity during the development itself, as produced code should be analyzed as soon as possible, but also during the project lifetime, during maintenance and refactoring tasks, as well as in product line situations where code is reused and branched.

In this context, large scale, long term projects often contain bulky legacy code too old or risky to modify, several third-party code modules which may or may not comply to the same coding principles, and the unfortunately too common duplicated code blocks, functions, or even files. To make things worse, this code only gets bigger as the project evolves.

These coding flaws have a direct negative impact on Maintainability, as it becomes harder and riskier to modify code. As a side effect, Reliability is also deteriorated, when new features are developed on an unsound base.

There are several ways to improve this situation; reducing code cloning may be among the first ones to apply.

Code refactoring doesn’t mean adding features or fixing bugs. It means rewriting portions of the code so that it becomes easier to understand, to enhance or to correct. A refactored code becomes less complex, more maintainable.

Threats on Maintainability

- Legacy code
- 3rd party code
- Cloned code

(source: US software development and maintenance trend [9])
2 Looking for code clones

2.1 State of the art

Research work and experimentations have extensively addressed code cloning detection. From cloning types classification [1], detection methods families and coding language specificities, numerous tools have been proposed, either in the academic ( [2], [3], [4], [5]), commercial ( [6], [7]), or open-source field ( [8]).

Several detection techniques are used, from simple string matching (PMD) to abstract syntax analysis (CloneDr, Bauhaus), and token analysis (CCFinder). Methods more evolved than string matching often limit their scope to a few language (generally C, C++, Java, sometimes fewer).

And if they aim at cross-languages clone detection, they require similarity-based algorithms, which are generally based on language grammars from the same language family. For example, works presented in [3] detect clones on C# and Java code, based on differences analysis between source code commits, requiring access to a version control system.

2.2 Introducing an industrial approach

There is however room for another approach, as experience and analysis of industrial development needs helped identify the following requirements, which previously cited tools don’t fully support:

- **Flexibility**
  The ability to support multiple languages and provide extensible grammars.
  As projects sub-components rely on several technologies and programming languages, clone detection has to support this diversity.

- **Time resilience**
  Cloned code should be tracked from one version to another.
  Finding a code clone should not be a one-time finding. Because the source code is bound to evolve, and the clone might not be handled the moment it is detected, it must be followed through time, and its location updated accordingly.

- **Adaptability**
  The ability to tailor cloning detection (detect name changing, exclude comments, adjust detection thresholds).
  Depending on the project, or the development team practices and objectives, clone detection must be configurable, thus allowing adapted detection and fewer false positives.

- **Dual mode**
  The ability to detect both textual or algorithmic clones.
  Textual cloning is a common issue to fix, generally caused by copy/paste. It helps factorize functions from duplicated code.
  Algorithmic cloning detection provides a finer result, assisting the refactoring process.

2.3 Code cloning detection proposal

A proposition supporting the needs previously cited is based on a dual cloning detection mechanism: string-based and token-based.

2.3.1 String-based textual cloning detection

Although simple and based on well-known techniques, the string-based textual cloning detection helps answer the classical search for code copy/paste, which still is a cause for high maintainability costs.

To produce relevant results, the proposed text duplication search may:

- Ignore blanks lines and comments,
- Define how much cloning between two artefacts is necessary for them to qualify as clones,
- Set the minimum size of duplicated text that can be considered as a clone.

2.3.2 The token-based algorithmic cloning detection

The token-based algorithmic cloning detection relies on the analysis of a source code’s control flow representation, which is using a generic analyser platform developed with several goals in mind:

- Support multiple languages. To this day, 18 coding languages are supported by our analyser, and more are continuously being developed.
- Extract a control flow representation, based on a modular, generic parser.
> Compute several static analysis metrics (from line counting to complexity, vocabulary frequency, nesting level …)
> Perform several programming rules checks (Recursions, Missing compound statements, Fallthrough …)

Analysing this symbolic representation, algorithmic clones can be detected (with similar parameters and thresholds as the textual cloning search).

With this method, we can address the following situations:
> Manage clones between multiple files, projects, and languages
> Detect as “cloned” copied source code where variables have been modified
> Detect as “duplicated” files where functions have been reordered

3 Efficient refactoring strategies

3.1 Static cloning filtering

Using algorithms previously cited helps detect all source code cloning (Textual or Algorithmic, Internal or External). However, all found clones may not be candidates to refactoring, and some may in fact have disastrous impacts if refactored.

Therefore, we propose to filter cloning results according to:
> Potential risk (e.g.: refactoring critical legacy code may produce unpredictable results)
> Induced costs (e.g.: refactoring highly tested code will imply heavy test updates)

The objective of this phase is to produce the best list of refactoring actions, with limited risk and additional costs, improving the project Maintainability.

3.2 Dynamic cloning filtering

The first phase produces results based on an analysis of the code at a given time. With the dynamic cloning search, the natural evolution of the source code is considered, to track “suspicious” code cloning.

A code cloning is detected as suspicious when it was first detected on a source code component, then manually relaxed, and is still being detected after the source code starts to change.

It is also possible to detect components which are still clones when the source code starts diverging.

These cases are particularly interesting, as they can help process similar code or algorithms, when their code starts to change.

A code factorization action is highly recommended in this case, before these codes are too far apart.

The objective of this phase is to anticipate design issues, improving the project Reliability.

3.3 Factorization Strategies

Based on the previous filtering phases, an efficient factorization can be elaborated according to these strategies:
> Minimum effort factorization
  This is achieved by addressing clones on components with small Technical Debt (i.e. with fewer non conformities to solve), and high cloning ratio results (i.e. highly similar, hence easier to factorize).
> Improved Maintenance factorization
  This is achieved by addressing clones on components with high Technical Debt, cloning ratio and density violation. This strategy focuses on solving non conformities on clearly cloned components, which directly improves Maintainability.
> Improved Reliability factorization
  This is similar to the previous case, but it also restricts the focus to components with the fewer number of associated tests, and test coverage ratio.

Several benefits are obtained from this strategy:
> Maintenance is improved (as with the previous case)
> Components in need of tests are detected, which will help improve Reliability
> If cloned components are linked to different tests, these tests should be factorized as well

4 Field usage

The proposed cloning detection, which is part of an extensive analytics solution, is being used in several software development projects from different industries (Automotive, Aerospace, Energy).

In a typical use case, R&D teams commit development to a version control system, which triggers a Continuous Integration pipeline. Code cloning detector, along with a Software Analytics assessment, is also part of this chain.

Assessment results are then accessible to team members. Regarding cloning detection results, possible actions help the team maintain a mature process:
> Continuously track clones as they appear and are handled
> Focus on clones according to the selected factorization strategy
> Relax clones when it is justified
> Deal with suspicious clones, automatically flagged as incriminated source codes drift one from another

5 Conclusion

Effective source code factorization can be achieved, based on code cloning detection able to detect Textual, Algorithmic, Internal or External clones. Additional knowledge such as risk, induced costs, or Technical Debt should be used to guide the factorization effort in an optimal manner.

Our ongoing studies aim at using algorithmic clone detection to point out bad (or good) coding practices within a project, establish a coding team profile from their coding habits, and develop a learning algorithm to classify projects with respect to their algorithmic similarities.

It is to be noted that the Squore monitoring tool from Vector provides an industrial solution fully addressing the software project factorization topic.

Squore embeds a static code analyzer able to detect and track Textual and Algorithmic clones, within or between components. It also computes a full set of industrial KPIs (among which Technical Debt), and can be included in a Continuous Integration loop to allow project history analysis, and anticipate issues.
Bibliography


Get More Information

Visit our website for:
> News
> Products
> Demo software
> Support
> Training classes
> Addresses

www.vector.com