Diggit: Automated Code Review via Software Repository Mining

Robert Chatley
Imperial College London
180 Queen's Gate
London, UK
rbc@imperial.ac.uk

Lawrence Jones
GoCardless Ltd
338-346 Goswell Road
London, UK
lawrence@gocardless.com

***Abstract*—We present Diggit, a tool to automatically generate code review comments, offering design guidance on prospective changes, based on insights gained from mining historical changes in source code repositories. We describe how the tool was built and tuned for use in practice as we integrated Diggit into the working processes of an industrial development team. We focus on the developer experience, the constraints that had to be met in adapting academic research to produce a tool that was useful to developers, and the effectiveness of the results in practice.**

***Index Terms*—software maintenance; data mining;**

I. Introduction

Peer code review is a well established practice amongst development teams aiming to produce high quality software, in both open source and commercial environments[1]. The way that code review is most commonly carried out today is that the reviewer is presented with just a snapshot of the proposed change, without the historical context of how this code has changed over time. Mostly commonly they see a diff giving a before/after comparison with the preceding version. We show that tooling can support and improve code review by automatically extracting relevant information about historical changes and trends from the version control system, and presenting these as part of the review.

Institutional memory is codified in version control systems. By building analysis tools on top of this that integrate into a developer's workflow, we are able to automatically provide guidance for a more junior developer, or someone new to a codebase. This paper reports on the experiences of a commercial development team using our tool, but we also see potential for use in open source projects, where there may commonly be a large number of contributors proposing individual patches and changes, compared to a relatively small community of core maintainers who may need to review these changes.

We aimed to integrate our analysis tools as seamlessly as possible into developers' regular workflow. We have therefore developed tools that integrate with the GitHub pull request[1] and review flow. When a pull request is made, our analysis engine runs, and our code review bot comments on the pull request. In order to provide timely feedback, we need to ensure that we can perform our analysis in a relatively short time-box. When developing their static analysis tool Infer[2], Facebook examined how quickly results needed to be returned in order for their developers to pay attention to them. They determined that a window of 10 minutes was the maximum that they could allow for static analysis to run before developers would give up waiting and move on. Based on this, we have also taken this 10 minute timebox as a benchmark for our tools, which informed our decisions when selecting and tuning analysis algorithms. Another observation from Facebook's work on Infer was that developers had a very low tolerance for false positives. As soon as their tool began warning about potential bugs that turned out not to be real problems, developers began to ignore the tool entirely. We therefore paid great attention to false positive rates and the relevance of generated comments.

This paper presents Diggit, a tool for running automated analysis to produce code review comments. Diggit will comment when a) past modifications suggest files are missing from proposed changes, b) trends suggest that code within the current change would benefit from refactoring, c) edited files display growing complexity over successive revisions.

We show that repository analysis using a well-chosen algorithm can provide automated, high quality feedback in a timely fashion, and allows us to present results in a context where immediate action can be taken. We evaluate the effectiveness of these methods when used by an industrial team.

Our priority when building Diggit was to produce a tool that would be useful in practice. To encourage adoption we needed to reduce the barrier to entry, making it easy for developers to integrate our tool into their existing processes. This resulted in us paying a lot of attention to aspects like authentication, creating a smooth setup experience for new users, integration with existing tools, and the general user experience – things that research projects might often place less emphasis on. The Diggit tool is now available as open source software[2].

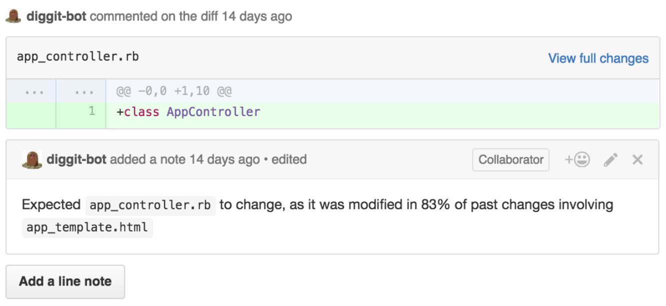
II. Related Change Suggestion

Working on large codebases can be disorientating, even for seasoned developers [3], [4]. Making a change is often not just a case of adding new code, but also integrating with and adapting existing parts of the system, test suites, configuration,

[1]https://help.github.com/articles/about-pull-requests/

[2]https://github.com/lawrencejones/diggit

documentation etc. New developers especially may struggle if they have yet to become familiar with the conventions, patterns and idioms used in a particular project. Even for developers relatively familiar with a codebase, it is easy to miss things. For example, perhaps in a particular codebase when a change is made in module X, it is normally required that a system test in module Y is updated too, but our developer has neglected this, either by accident or because they did not know about this relationship. Or it may be the case that the developer has correctly changed the relevant file, but simply forgotten to add it to the current commit. Diggit's *related change suggestion* analysis aims to suggest files that may have been omitted from the current commit. It mines common file groupings based on temporal coupling[5] from a codebase's Git history. Such suggestions act as both a safeguard and learning tool for developers as they work within a project, raising awareness of file coupling and the nature of idiomatic changes.


diggit-bot commented on the diff 14 days ago

app_controller.rb

```
@@ -0,0 +1,10 @@
+class AppController
```

diggit-bot added a note 14 days ago • edited

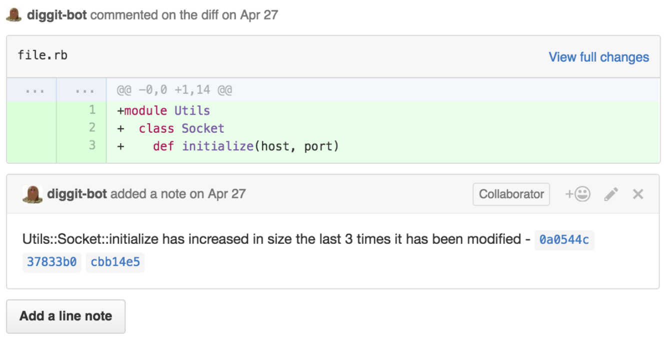
Expected `app_controller.rb` to change, as it was modified in 83% of past changes involving `app_template.html`

Fig. 1. Suggesting files commonly changed together.

We mine association rules from a given codebase's version control history. Similar analysis has previously been performed in work such as ROSE [6] and TARMAQ [7]. Here we do not aim to present a novel algorithm, but to describe the forces at play when a tool based on research was developed to integrate into the working processes of an industrial team.

For our implementation we investigated a number of possible algorithms for mining association rules and generating suggestions. We started with the Apriori algorithm, together with the Apriori-TID optimisation, as described by Agrawal [8]. We compared this with an implementation of the FP-Growth algorithm described by Han et al [9], with optimisations as discussed by Borgelt [10] to improve the memory usage.

We had two goals in selecting and tuning an analysis algorithm. We wanted to provide feedback in a timely manner, but also to maintain a low false-positive rate. Our tool will be no use if suggestions are simply ignored by developers due to latency or inaccuracy. Requiring high confidence might mean that for many projects, especially those without a long revision history, we simply do not have enough data in the repository to produce many suggestions. Allowing a lower confidence may well mean that we produce erroneous suggestions. For a compelling developer experience it was important that we come up with appropriate confidence values.


diggit-bot commented on the diff on Apr 27

file.rb

```
@@ -0,0 +1,14 @@
+module Utils
+  class Socket
+    def initialize(host, port)
```

diggit-bot added a note on Apr 27

Utils::Socket::initialize has increased in size the last 3 times it has been modified - `0a0544c` `37833b0` `cbb14e5`

Fig. 2. Highlighting code quality trends over successive changes.

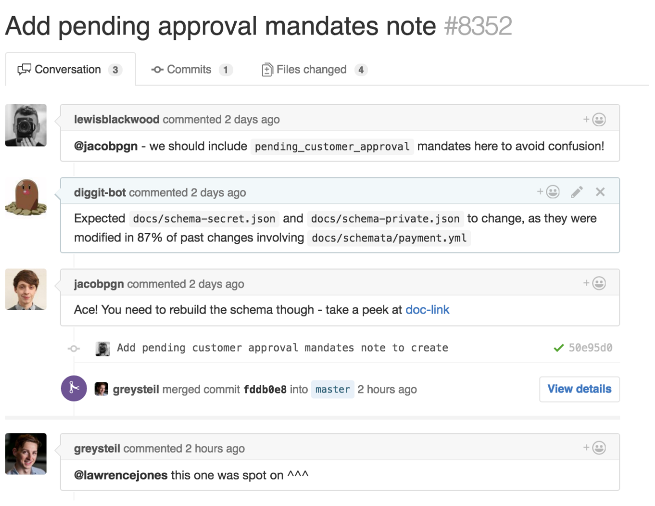
There is not room in this paper to present a full analysis of our performance experiments, or how parameters such as mimimun support were tuned, but full details can be found in the accompanying technical report [11]. As a summary, FP-Growth produced results orders of magnitude faster than Apriori in our experiments, whilst still giving useful results. Diggit therefore uses FP-Growth for its analysis.

III. Code Quality Trends

Agile methods are prevalent in industrial software development, and as such it is expected that a codebase will be changed frequently over time. In such an environment it is common for a codebase to deteriorate with age [12], especially if there is no sustained effort to improve code quality, through continuous refactoring [13].

The continuous application of small improvements helps developers to maintain hygiene standards in a codebase, and to prevent the accumulation of technical debt that may make future changes difficult and possibly economically unviable. Developers rarely introduce significant design problems all at once – or at least if they do, we hope that the process of code review should catch them before the change is integrated. More difficult to detect is when there is a gradual trend of things getting worse over time. Agile teams often favour a culture of collaborative code ownership [14], so it may well be that every developer on a team changes many different areas of a codebase during their work on a system, but may not have a long term engagement with any particular area of the code. They may make a minor change to an existing class or method that adds some new functionality, a small addition to an existing foundation. If every change makes the code just a tiny bit worse, then we may have the problem of the proverbial "boiled frog", where we do not notice until it is too late.

To help with this, we added the detection of code quality trends to Diggit. When a change is made, Diggit analyses trends in this particular area of code over previous revisions, and generates comments suggesting that the developer might want to consider code quality at this point. The aim is to provide feedback "just in time" to encourage developers to refactor before the work on this particular area of code is completed. This is in contrast to an offline review tool that may be able to highlight areas of a codebase that might benefit



Add pending approval mandates note #8352

Conversation 3 · Commits 1 · Files changed 4

lewisblackwood commented 2 days ago

@jacobpgn - we should include `pending_customer_approval` mandates here to avoid confusion!

diggit-bot commented 2 days ago

Expected `docs/schema-secret.json` and `docs/schema-private.json` to change, as they were modified in 87% of past changes involving `docs/schemata/payment.yml`

jacobpgn commented 2 days ago

Ace! You need to rebuild the schema though - take a peek at doc-link

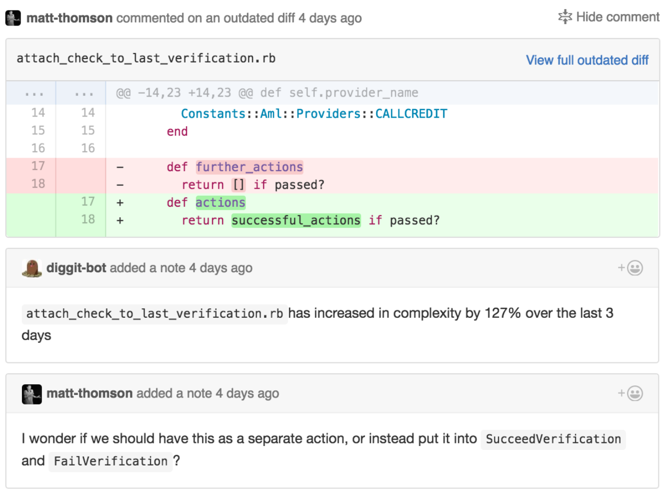
`Add pending customer approval mandates note to create` ✓ `50e95d0`

greysteil merged commit **fddb0e8** into `master` 2 hours ago · View details

greysteil commented 2 hours ago

@lawrencejones this one was spot on ^^^

Fig. 3. Diggit correctly highlighting a forgotten change.



matt-thomson commented on an outdated diff 4 days ago · Hide comment

`attach_check_to_last_verification.rb` · View full outdated diff

```
@@ -14,23 +14,23 @@ def self.provider_name
          Constants::Aml::Providers::CALLCREDIT
        end

-       def further_actions
-         return [] if passed?
+       def actions
+         return successful_actions if passed?
```

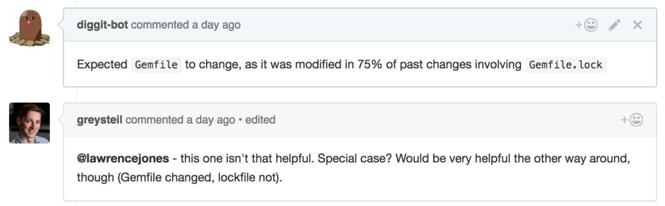
diggit-bot added a note 4 days ago

`attach_check_to_last_verification.rb` has increased in complexity by 127% over the last 3 days

matt-thomson added a note 4 days ago

I wonder if we should have this as a separate action, or instead put it into `SucceedVerification` and `FailVerification`?

Fig. 4. Manual reviews addressing growing complexity.



diggit-bot commented a day ago

Expected `Gemfile` to change, as it was modified in 75% of past changes involving `Gemfile.lock`

greysteil commented a day ago • edited

@lawrencejones - this one isn't that helpful. Special case? Would be very helpful the other way around, though (Gemfile changed, lockfile not).

Fig. 5. Not all of Diggit's suggestions are helpful.

from refactoring, should the team ever get around to it as a separate maintenance activity.

The first quality trend that Diggit analyses is based upon Feathers' observations on refactoring diligence [15]. Feathers analyses repository data to give a summary profile, for example revealing that there are 135 methods that increased in size the last time they were changed, 89 methods that increased in size the last two times they were changed, and so on. High numbers of methods that are consistently expanded are indicative of a lack of diligence in refactoring.

In generating code review comments, we do not generate a profile for the whole codebase (as Feathers does), but trace back through the history of the code in the current change, and look for consecutive increases in method length.

We created a similar analysis module to highlight trends in computational complexity, using a method based on measuring whitespace and indentation [16] to give an approximation of code complexity whilst preserving some language independence. It is not always the case that complex code needs to be simplified – some modules implement complex algorithms, but are closed and need no further change. The more problematic case is when we have code of high complexity that changes often. Therefore, detecting increases in complexity at the point of change (and review) allows us to highlight a combination of high (or increasing) complexity and frequent change, which may indicate a hotspot in the codebase where refactoring would be likely to be beneficial. The tool can generate comments about trends either over the last n revisions, whenever those changes occurred, or trends over a period of time. In our trials, a threshold of $n = 3$ was used to trigger a warning.

IV. Use In Practice

To explore whether Diggit is a useful tool in practice, we studied its use with the development team at GoCardless. GoCardless is a company that runs a platform facilitating electronic payments. They have a development team comprising approximately 20 developers. The company already has a development process that involves manual code review managed through GitHub pull requests. The team used Diggit on one of their core services, for historical reasons known simply as `gocardless`. This is the main repository for the GoCardless API, with around 150k lines of Ruby code with contributions from over 50 developers over the lifetime of the project. Development typically proceeds at a rate of around ten pull requests a day. Diggit was set to analyse pull requests on the `gocardless` repository, at first in a hidden mode, so that we could see what it would do and tune the parameters, and then in a live mode, where it commented on the developers' pull requests as part of their normal review process.

We asked the developers to provide feedback on comments that have been useful to them during review, as well as those that were not helpful so that we could further improve the system. One example where the automation worked very well is a change made by a developer from the GoCardless support team, who wanted to modify a schema description in response to a comment from a customer. The developer who made this change, Lewis, makes infrequent changes to `gocardless` and consequently forgot to rebuild the schema files after changing the schemata[3].

The pull request in Figure 3 was created by user lewisblackwood, and we see that the first comment to appear is from the Diggit bot, suggesting that `schema-secret.json` and `schema-private.json` were likely missing from this

[3]https://github.com/interagent/prmd

```
+-------------------------------------------------------------+---+---+---+---+---+---+
| gocardless/pull/8322                                        | 1 | 2 | 3 | 4 | 5 | 6 |
+-------------------------------------------------------------+---+---+---+---+---+---+
| Complexity#attach_check_to_last_verification.rb             | X | X | X | X | X |   |
| Complexity#aml/checkers/base.rb                             | X |   |   |   |   |   |
| Complexity#state_machines/verification.rb                   |   |   | X |   |   |   |
| ChangePatterns#aml/checkers/checker_a.rb                    | X |   |   |   |   |   |
| ChangePatterns#aml/checkers/checker_b.rb                    |   | X | X | X |   |   |
| ChangePatterns#aml/checkers/checker_c.rb                    |   | X | X | X |   |   |
| ChangePatterns#aml/checkers/checker_d.rb                    |   | X | X | X |   |   |
| ChangePatterns#aml/checkers/trigger_manual.rb               |   | X | X | X |   |   |
| ChangePatterns#spec/legal_entity_check_spec.rb              | X | X | X |   |   |   |
+-------------------------------------------------------------+---+---+---+---+---+---+
```

Fig. 6. Comment occurance on pull request 8322 of `gocardless` (some filenames anonymised).

change. Examining the analyses in Diggit's database revealed that Lewis subsequently rebuilt the schema, pushed a new commit to the pull request. As this problem was now resolved, when Diggit ran over later commits to the same pull request (before it was merged), the analysis produced no comments. This is exactly the pattern that we would expect if a developer updates their change to resolve a problem that Diggit reports.

In Figure 3, we can see a correlation between the comment by jacobpgn and the comment that Diggit generated. The correctness of the analysis is further highlighted by the comment from greysteil, a member of the technical leadership team, noting Diggit's accuracy in this case.

More evidence of Diggit's accuracy was pull request 8322 for `gocardless`, which was a large refactoring across 24 files. The initial push modified 14 files, triggering several file suggestion and complexity warnings from Diggit. The developer continued to refine this change over 6 revisions before it was approved and merged into the master branch. The table in Figure 6 shows how Diggit commented on each of those revisions, and how by the end of the process all the comments Diggit made were resolved. It is interesting to note that in the second push, two warnings from the first revision were resolved, but four new ones were triggered. These are file suggestions for `aml/checkers` which were triggered when `checker_a.rb` was added to the diff, and then subsequently fixed. The comment about the increased complexity of `attach_check_to_last_verification.rb` was matched by a manual review comment (Figure 4) that suggested moving the change out into a new action, hence splitting the code into a larger number of simpler components.

V. User Feedback

As well as gathering data from Diggit and the GoCardless code repository to see what analysis was generated, and subsequent changes, we also asked the GoCardless developers to provide qualitative feedback and used this to refine the tool. One issue that GoCardless experienced with the file suggestions were false positives caused by links between two files where changing file `A` would require a developer to modify file `B`, but modifying `B` would not require modifications to `A`. This issue was raised after an automated tool created pull requests upgrading each dependency of `gocardless`, with Diggit commenting (Figure 5) on pull requests that did not change the `Gemfile`, only the `Gemfile.lock`.

As greysteil comments in Figure 5, the association between `Gemfile` and `Gemfile.lock` is significant when a change to `Gemfile` leaves `Gemfile.lock` untouched. Unfortunately the reverse is not useful. In Ruby projects the `Gemfile` lists the required libraries, but the the `Gemfile.lock` records the particular versions of these libraries, so adding a new library requires a change to both, but upgrading a version only requires a change to the lock file. Increasing Diggit's confidence threshold could prevent these warnings, but would reduce the overall recall. Also, tuning the parameters to vary confidence requires specialist knowledge of the mining algorithm. Most GoCardless developers working with the tool preferred to treat it as a black box, and instead to specify individual exceptions to the analysis rules using an ignore file. This allowed them to filter out false positives.

GoCardless also highlighted a few spurious results from the complexity analysis reporter. In some cases minor alterations to files were causing warning comments about increasing complexity, despite the change being isomorphic. Rubocop[4], the prevalent Ruby linter, suggests that method parameters be aligned on following lines when a single line method call would exceed the set line-length. This often leads to method calls where the code is formatted such that the parameters are broken onto the next line. Whitespace integration often detected this additional indentation, resulting in a large (but misleading) increase to reported complexity.

Reducing our analysis sensitivity to these stylistic issues required the tool to have a great understanding of the code. This demanded language-specific tooling (which we had initially been trying to avoid, in order to remain language agnostic), but once this decision was taken we could perform more detailed analysis. We changed Diggit's analysis engine to use a plugin mechanism that allows language-specific analyses, keyed against particular file extensions, and used an ABC complexity measure [17] for Ruby files, making use of the abstract syntax tree. This complexity metric is unaffected by

[4]https://github.com/bbatsov/rubocop

formatting changes, and so although we lost a little generality, we reduced false positives.

Diggit has a 21% comment ratio at GoCardless: it produces a comment on approximately 1 in every 5 pull requests processed. Anecdotal feedback from the team showed that this amount of feedback felt about right to them. They were aware of the tool doing something useful, without it overwhelming their existing review process.

VI. Evaluation and Conclusion

We evaluated the effectiveness of Diggit's comments by comparing Diggit's suggested actions for a given pull request to those observed after manual code reviews. We explored this correlation by running analysis on twenty software projects (inside and outside GoCardless), looking at pull requests with manual reviews. We consider a Diggit comment to be effective if it is generated for one revision within a pull request, but not in subsequent revisions (the problem has been fixed). If we see this fix pattern for a pull request when the developers only had access to the manual review comments (with Diggit running in hidden mode), we infer that Diggit is automatically generating similar feedback to what a human would give.

TABLE I
Diggit comment resolve rates.

Reporter	Resolve Rate	Resolved	Total
Refactoring Diligence	38%	24	64
Complexity	44%	38	87
Change Suggestion	59%	104	176
Overall	51%	166	327

Our results (Table I) support the conclusion that the same issues Diggit highlights are being tackled during manual review. On average, over 50% of the comments Diggit makes are resolved prior to merging of the pull request, suggesting a strong correlation between comments made by human reviewers and Diggit's analysis. Change suggestion has the highest resolve rate, with 59% of suggestions fixed before the pull request is merged. Complexity and refactoring diligence comments also show strong resolve rates, both seeing over a third of comments being resolved before merge.

In criticism of resolve rate, it only approximates developers taking action on analysis suggestions. Comments made to files that were then later removed from the change would be seen as resolved, for example. Conversely, sometimes comments would suggest taking action that is subsequently addressed in a separate change, which our statistics would miss. Overall these results indicate a general agreement between Diggit's comments and the actions taken in code review, but show that there is still room for improvement.

Looking specifically at GoCardless projects, 65% of the comments generated were taken as actionable by the developers and resulted in a fix. This higher rate may be due to the ability to suppress false positives on a project-specific basis.

Given these results we believe that Diggit demonstrates the potential for analysis tools to support code review. Although, we note the particular attention that needs to be paid to practical issues to integrate tools into an existing development process, and have engineers engage with them. Diggit provides an example of successfully harnessing techniques developed in research, and applying them to historical data already accumulated by the vast majority of industrial teams, and using this to help developers improve code quality by providing timely automated feedback on proposed changes.

References

[1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 192–201. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597076

[2] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 459–465. [Online]. Available: http://dl.acm.org/citation.cfm?id=1986308.1986345

[3] S. Elliott Sim and R. C. Holt, "The ramp-up problem in software projects: A case study of how software immigrants naturalize," in *Proceedings of the 20th International Conference on Software Engineering*, ser. ICSE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 361–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=302163.302199

[4] L. M. Berlin, "Beyond program understanding: A look at programming expertise in industry," *Empirical Studies of Programming*, vol. 93, no. 744, pp. 6–25, 1993.

[5] A. Tornhill, *Your code as a crime scene : use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. Frisco, TX: Pragmatic Bookshelf, 2015.

[6] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, 2005.

[7] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, and D. W. Binkley, "Generalizing the analysis of evolutionary coupling for software change impact analysis," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 201–212.

[8] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[9] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM Sigmod Record*, vol. 29, no. 2. ACM, 2000.

[10] C. Borgelt, "An implementation of the fp-growth algorithm," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*. ACM, 2005, pp. 1–5.

[11] L. Jones, "Diggit mining source code repositories for developer insights," Imperial College London, Tech. Rep., 2016. [Online]. Available: http://www.imperial.ac.uk/computing/prospective-students/distinguished-projects/ug-prizes/archive/

[12] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287. [Online]. Available: http://dl.acm.org/citation.cfm?id=257734.257788

[13] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Object Technology Series. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[15] M. Feathers, "Detecting refactoring diligence," dec 2014.

[16] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: Indentation as a proxy for complexity metric," in *2008 16th IEEE International Conference on Program Comprehension*, June 2008, pp. 133–142.

[17] J. Fitzpatrick, "Applying the ABC metric to C, C++, and Java," 1997.