Mapping Class Dependencies for Fun and Profit

Allen Kanapala[1], Gregory Gay[2]

[1]University of South Carolina Salkehatchie, Allendale, SC, USA
[2]University of South Carolina, Columbia, SC, USA
kanapalaa@acm.org, greg@greggay.com

Abstract. Classes depend on other classes to perform certain tasks. By *mapping* these dependencies, we may be able to improve software quality. We have developed a prototype framework for generating optimized groupings of classes coupled to targets of interest. From a pilot study investigating the value of coupling information in test generation, we have seen that coupled classes generally have minimal impact on results. However, we found 23 cases where the inclusion of coupled classes improves test suite efficacy, with an average improvement of 120.26% in the likelihood of fault detection. Seven faults were detected only through the inclusion of coupled classes. These results offer lessons on how coupling information could improve automated test generation.

Keywords: Coupling, Search-Based Software Engineering, Software Testing

1 Introduction

In complex systems, *coupled* classes depend on other classes to perform certain tasks [6]. By *mapping* and grouping these dependencies, we may be able to offer valuable information that can improve software quality.

Automated test generation can be performed to control testing costs. However, a question remains—which classes should be targeted for generation? Often, only the classes that are known to be faulty are targeted. However, a class that is coupled to a faulty class may still exhibit unexpected behavior. By generating tests for coupled classes, we may be able to detect faults that would otherwise be missed.

We have developed a prototype framework to investigate the effect of coupling in test generation. The framework maps the dependencies between Java classes into a directed graph. This graph can then be used to generate small, dense groupings of classes centered around selected targets. To understand whether test generation is more effective when including coupled classes, we have performed a pilot case study. Using 588 real faults from 14 Java projects, we have identified groupings of classes, generated test suites for these groupings and the faulty classes alone, and assessed whether the inclusion of coupled classes improves the likelihood of fault detection.

Overall, there is only an average improvement of 3.79% in the likelihood of fault detection when incorporating coupled classes. However, when these additional classes have *any* impact, there is an average improvement of 120.26% and seven additional faults were detected only through coupling. The inclusion of coupled classes could

yield significant efficacy improvements if we can identify in advance where they would be useful, and improve coverage of dependencies. In addition, our optimization process often yields unnecessarily large groupings.

We hypothesize that the ability to map and optimize groups of coupled classes could benefit many areas of software engineering research—particularly when automating tasks. Our framework has offered promising preliminary results. We will further explore how coupling information could improve automated software engineering.

2 Coupling Mapping Framework

We have developed a framework that maps class dependencies into a directed graph[1]. We then use this graph to optimize small, highly-interconnected groupings of classes coupled to designated targets using a simple genetic algorithm. The following basic process is used to generate groupings:

1. This framework first maps dependencies between classes. In this case, we consider dependencies to be either method calls or variable references to another class.
2. A directed graph is created, where each class is a node, and each edge indicates a dependency. Any classes that have no dependencies and are not the target of a dependency will be filtered out from consideration at this stage. If no classes are coupled to a changed class, the changed class will still be added to the target list.
3. We generate a population of 1,000 groupings, formed by randomly selecting classes.
4. Each grouping is scored using the fitness function described below, and a new population is formed through retention of best solutions (by default, 10%), mutation (20%), crossover (20%), and further random generation (50%).
5. Evolution continues until the time budget is exhausted—by default, five minutes[2].
6. The best grouping is returned. The changed classes are added to that grouping.

The fitness function used to score groupings is:

$$F_G = \sqrt{\overline{size}^2 + \left(\overline{coverage} - 1\right)^2 + \overline{avg(distance)}^2} \tag{1}$$

That is, we prioritize groupings that are closer to a *sweet spot* of fewer classes ($size$), where the chosen classes are coupled to a large number of other classes ($coverage$), and where more classes are either coupled directly to the changed classes or through a small number of indirect dependency links (average $distance$). This should result in a relatively small grouping of classes that are densely coupled to each other and other classes. $\overline{x}$ is a normalized value $0 \leq \frac{x - min(x)}{max(x) - min(x)} \leq 1$. Scores range from $0 \leq F_G \leq \sqrt{3}$ and lower scores are better.

3 Case Study

Traditionally, in unit test generation research, tests are generated solely for the classes we know to contain faults. However, other classes may depend on the faulty classes, and

[1] Available from `https://github.com/Greg4cr/Coupling-Mapping`

[2] Experimentation suggested that convergence was often reached before that time.

by targeting these coupled classes, we may be more likely to detect faults. We wish to examine whether we could use knowledge of class dependencies to enhance test generation. Specifically, we wish to address: (1) *Can the inclusion of coupled classes improve the efficacy and reliability of test suite generation?* (2) *Are the groupings produced by our framework small enough to be of practical use?*

We have performed the following experiment: (1) We have gathered 588 real faults, from 14 Java projects. (2) For each fault we generate 10 groupings of coupled classes. (3) For each fault, we generate 10 suites per grouping (and for the set of faulty classes) using the non-faulty version of each class. We allow a two-minute generation budget per targeted class. (4) For each fault, we measure the proportion of test suites that detect the fault to the total number generated.

Defects4J is an extensible database of real faults extracted from Java projects [4][3]. Currently, it consists of 597 faults from 15 projects. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault. The Guava project was omitted from this study, as its code uses features not supported by our framework. We have used the remaining 588 faults for this study.

EvoSuite applies a genetic algorithm in order to evolve test suites over several generations, forming a new population by retaining, mutating, and combining the strongest solutions [7]. It is actively maintained and has been successfully applied to the Defects4J dataset [2]. In this study, we used EvoSuite version 1.0.5.

Tests are generated from the fixed version of each class and applied to the faulty version in order to eliminate the oracle problem. Tests are generated targeting Branch Coverage, and EvoSuite is allowed two minutes per class—a time chosen to fit within the strict time constraints of the continuous integration (CI) process that testing is commonly performed as part of. In the CI process, changed code is built, verified, and deployed. As this process may be performed multiple times per day, test generation and execution must take place on a limited time scale. As results may vary, we generate 10 groupings of classes per fault, and we perform 10 test generation trials for each fault, grouping, and budget. Generation tools may generate flaky (unstable) tests [2]. We automatically remove non-compiling test cases. Then, each test is executed on the fixed CUT five times. If results are inconsistent, the test case is removed. On average, less than 1% of tests are removed from each suite.

4 Results & Discussion

In Table 1, we compare the average likelihood of detection between the normal case—where only the faulty classes are targeted—and when we generate for a set of targets including coupled classes. From this table, we can see that there is often *some* improvement, but the overall effect is minimal. The inclusion of coupled classes fails to improve results for six systems. For the others, we see average improvements of up to 13.36%. Overall, the average improvement from including coupled classes is only 3.79%.

To understand when coupled classes can benefit generation, we can filter out situations where their inclusion does not improve results. Table 2 lists the average likelihood

[3] Available from `http://defects4j.org`

Project	Detection Likelihood (Changed-Only)	Detection Likelihood (With Coupled)
Chart	40.00%	42.58%
Closure	4.10%	5.10%
CommonsCodec	31.36%	35.55%
CommonsCSV	55.00%	58.50%
Jsoup	19.80%	21.70%
Lang	35.20%	35.50%
Math	28.68%	29.29%
Time	34.40%	35.90%
Overall	22.69%	23.55%

Table 1. Average likelihood of detection when only changed classes are targeted and when coupled classes are included, omitting systems with no observed differences.

Project	Detection Likelihood (Changed-Only)	Detection Likelihood (With Coupled)
Chart	15.00%	27.50%
Closure	30.00%	62.50%
CommonsCodec	13.33%	44.00%
CommonsCSV	30.00%	51.00%
Jsoup	15.00%	27.80%
Lang	10.00%	30.00%
Math	6.67%	28.33%
Time	25.00%	44.00%
Overall	18.26%	40.22%

Table 2. Average likelihood of detection–omitting cases where coupled classes have no effect.

of detection for the 23 faults where the inclusion of coupled classes had an impact. These filtered results show that when additional classes have *any* impact, it is a major one. In such cases, the likelihood of detection improves by an average of 120.26%. In fact, seven new faults were *only* detected by including coupled classes.

While the addition of classes can be very powerful, it is also very expensive given that—by default—the same amount of time is devoted to generating test cases for each class. Our results illustrate that we should not generate tests for such classes if there is a low likelihood they will help detect faults. **To decide when to add additional targets, we must understand when their inclusion will be helpful.**

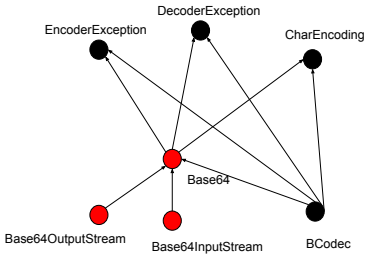

Fig. 1. Partial visualization of coupling. Relevant classes are colored red.

Figure 1 depicts a selection of classes in the CommonsCodec project. Three faults—centering around the `Base64` class (faults 12, 15, and 20[4])—see improved efficacy from the inclusion of coupled classes `Base64Input Stream` and `Base64OutputStream`. Tests generated solely to target `Base64` are able to detect all three faults, but not reliably. The incorporation of these two coupled classes greatly increases the likelihood of detection. Class—`BCodec`—is also coupled to `Base64`, but does not contribute to efficacy.

These three examples are interesting because the two additional classes are not just coupled through in-code dependencies, but all three are linked by a common conceptual purpose—encoding binary data by treating it numerically and translating it into a base 64 representation. One option for incorporating coupled classes would be to periodically present human developers with coupling information and ask them to filter groupings. Each time that any class in that grouping is altered, those coupled classes could be included in generation.

Of course, not all situations where coupling assists are as straightforward as the CommonsCodec example. For instance, consider fault 31 for the Math system[5]. Generated tests never detect the issue when targeting faulty class `ContinuedFraction`.

[4] `https://github.com/Greg4cr/defects4j/blob/master/framework/projects/CommonsCodec/patches/[12/15/20].src.patch`

[5] `https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/31.src.patch`

Instead, the fault is only detected when tests are generated for `Gamma` (coupled to `Continued Fraction`) and `GammaDistribution` (coupled to `Gamma`). The reason the coupled classes are useful is likely because they provide guidance to the generator in how to make use of `ContinuedFraction`. Exposing the fault requires setting up a series of values and calling `ContinuedFraction.evaluate(...)` on those values. By attaining coverage of `Gamma`, EvoSuite is able to set up and execute the functionality of `ContinuedFraction`. Without that guidance, it struggles.

While only a small number of classes are coupled to `Continued Fraction`, there is not a common conceptual connection like with the CommonsCodec example above. In retrospect, we can explain these situations. However, more research is needed to recognize patterns in when the inclusion of coupled classes is beneficial. Further, asking developers to name useful couplings creates additional maintenance effort, and may not offer sufficient benefit for the time and knowledge required. Therefore, we also need further research into automated means to suggest and prune couplings.

We should also endeavor to make the inclusion of coupled classes more useful by focusing on increased coverage of those dependencies. **The efficacy of generation—when coupled classes are included as targets—may be improved if coverage is ensured of references to changed classes.**

Test generation for each class is an entirely independent process. While the attained Branch Coverage may be relatively high for each targeted class, we have no guarantee that dependencies *between classes* are covered. Steps could be taken to improve coverage of such dependencies by considering coverage of dependencies between classes. Jin and Offutt have proposed coverage criteria for integration testing that could be used to ensure class dependencies are covered [3]. These forms of "Coupling Coverage" could be used to prioritize suites that attain a higher coverage of the specific code segments that require data or functionality from a changed class.

Recent work has found that combinations of coverage criteria can be more effective than individual criteria [2]. For example, combining Branch and Exception Coverage yields test suites that both cover the code and force the program into unusual configurations. "Coupling Coverage" metrics could be thought of as another situationally-appropriate orthogonal criterion. Rather than generating tests using Branch Coverage alone, the generator could combine Branch and "Coupling Coverage" when targeting coupled classes—potentially creating suites that are especially effective at exploiting dependencies between classes, and in turn, at detecting faults.

	Number of Classes
Chart	18.63
Closure	73.92
CommonsCLI	1.91
CommonsCodec	3.51
CommonsCSV	3.40
CommonsJXPath	20.80
JacksonCore	5.03
JacksonDatabind	34.96
JacksonXML	2.80
Jsoup	26.34
Lang	7.12
Math	12.64
Mockito	34.43
Time	52.87

Table 3. Average number of classes in the groupings.

Regardless of the use, our framework is intended to produce small, effective groups of coupled classes. The size of that group must be small enough to be of practical use. In Table 3, we list the average grouping size for each system. In many cases, we can see that these groupings are larger than would be practical. We used them for this case

study, as they are useful for understanding the benefits of such information. **However, we must refine the optimization process to further limit grouping size.**

We found that, in situations where coupling affects the results, only a small number of classes are useful, and they are closely linked to the target classes. Therefore, these groupings could be easily pruned down to a more appropriate size. We will reformulate our fitness function to further constrain grouping size.

5 Related Work

Coupling between classes is a well-established area of research [6]. Similar search-based techniques have been used to suggest refactorings. The CCDA algorithm uses a graph structure and a genetic algorithm to restructure packages based on class dependencies [5]. However, we are aware of no other use of such techniques to optimize groupings for test generation. Past work on integration testing has suggested ways to better ensure that class dependencies are tested [3, 1], but has largely not addressed the question of *which* classes to test. In addition, we are not focused purely on integration testing, but a broader set of scenarios.

6 Conclusions

We have developed a framework to optimize groupings of classes. The results of a pilot study on the applicability of coupling to test generation show potential benefits from generating tests for coupled classes and offer new research challenges.

References

1. Alexander, R.T., Offutt, A.J.: Criteria for testing polymorphic relationships. In: Proceedings 11th International Symposium on Software Reliability Engineering. pp. 15–23 (2000)
2. Gay, G.: Generating effective test suites by combining coverage criteria. In: Proceedings of the Symposium on Search-Based Software Engineering. SSBSE 2017, Springer Verlag (2017)
3. Jin, Z., Offutt, A.J.: Couplingbased criteria for integration testing. Software Testing, Verification and Reliability 8(3), 133–154, https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%281998090%298%3A3%3C133%3A%3AAID-STVR162%3E3.0.CO%3B2-M
4. Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440. ISSTA 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2610384.2628055
5. Pan, W., Jiang, B., Xu, Y.: Refactoring packages of objectoriented software using genetic algorithm based community detection technique. International Journal of Computer Applications in Technology 48(3), 185–194 (2013), https://www.inderscienceonline.com/doi/abs/10.1504/IJCAT.2013.056914
6. Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: 22nd IEEE International Conference on Software Maintenance. pp. 469–478 (Sept 2006)
7. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering, Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-22183-0_7