Beyond Test Flakiness: A Manifesto for a Holistic Approach to Test Suite Health

Phil McMinn
University of Sheffield, UK

Muhammad Firhard Roslan
University of Sheffield, UK

Gregory M. Kapfhammer
Allegheny College, USA

***Abstract*—Large numbers of flaky tests are a sure sign of a dysfunctional, or "unhealthy", test suite. In this paper, we identify several further indicators of unhealthy test suites, arguing that trade-offs exist among some indicators, with others complementing one another. We encourage researchers and practitioners not to stop at test flakiness—or any individual metric of "good" or "bad" tests—and instead focus on developing and adopting techniques and tools that holistically address test suite health. For more information, see https://philmcminn.com/test-suite-health.**

I Introduction & Motivation

Since flaky tests [7] may fail when a program is actually working or pass when it is not, developers cannot trust them to accurately assess their software. This paper argues that flaky tests are just one of a series of indicators of an "unhealthy" test suite—a test suite that is not fit for the purpose of giving developers timely feedback about the correctness of the software they are developing, either at the present time or in the future. It contends that considering flakiness in combination with other test suite health indicators is advantageous for determining what is a truly useful, functioning test suite. These indicators can be organized into a checklist, with the aim of minimizing or maximizing the presence and impact of each one. Critically, we argue that, while some indicators are complementary, trade-offs exist among others, suggesting that it is not appropriate to optimize for each one separately from the others. This paper therefore encourages researchers to look "beyond" test flakiness, and, in particular, the isolated consideration of it and other indicators of test health. Instead, it suggests that test suite health is a big-picture issue that must be holistically tackled to encourage practitioners to write "good" test suites that are fit for their purpose, supported by researchers who design and experimentally evaluate automated techniques and tools to assist them in effectively doing so.

II Indicators of Unhealthy Test Suites

Unhealthy test suites have several tell-tale signs. In this section, we identify a list of nine test suite health indicators, the first being **I0: Flakiness**. We discuss further indicators starting with those that are the most well-understood and measurable, to those that are less so and more diffuse. We do not claim our list to be exhaustive, but rather as a starting point for researchers, and to promote discussion at the workshop.

I1: Low Code Coverage. A test suite with low code coverage does not execute regions of the program under test and is thus the first and most obvious indicator of poor test suite health. While high coverage does not indicate good test suite health either, as we will later argue, low coverage tests will not be able to reveal many defects or give developers useful feedback.

I2: High Pseudo-Testedness. Pseudo-tested program elements (e.g., statements or methods) are executed by tests, but can be removed from the program without any impact on the pass/fail behavior of the tests [6], [11]. High levels of pseudo-testedness reveal poor test suite health in the form of a lack of assertions. Perhaps paradoxically, a test suite with high levels of pseudo-testedness is likely to be less prone to flakiness (**I0**)—due to fewer checks, and brittleness (**I6**)—due to less stringent assertions, suggesting a trade-off between health indicators.

I3: Low Mutation Score. A low mutation score indicates a test suite that is not very sensitive to faults, and is therefore also an indicator of poor test suite health [6]. If there is high pseudo-testedness then the mutation score will be low, since if code can be removed without the test suite "noticing", it is unlikely that seeded faults will be detected either. Yet, tests with a high mutation score may be more brittle (**I6**) if they are "change detectors" that myopically focus on the program's current implementation rather than its intended specification.

I4: Long-Running Test Suites. If a test suite is not fast to execute, it will not provide quick feedback to developers who thus may be dissuaded from using it as frequently as they should. While there are techniques to reduce test suites and prioritize important tests, developers must be careful not to use them as a "sticking plaster" that deceptively masks problems and/or significant bottlenecks in the test suite's execution. This may, for example, require a slow-running component to be mocked to improve test speed. However, testers need to be careful this does not reduce realism (**I7**) or potentially increase either the flakiness (**I0**) or brittleness (**I6**) of their test suite.

I5: Low Diversity of Tests. Test diversity can be measured in a variety of ways, for example, based on execution traces [12], or the actual text of test cases [5]. Regardless, low test diversity likely indicates that tests are executing similar program paths, which may increase pseudo-testedness (**I2**) and decrease mutation scores (**I3**). It may also contribute to a high level of brittleness in tests (**I6**), since if the interfaces to frequently-called methods change, then a large number of tests will also need to be changed. This can also arise when developers copy and paste from prior test cases when they create new ones [1].

I6: High Brittleness. Large numbers of tests that break due to production code changes suggest that tests are highly coupled to implementation details [4], another signal of an unhealthy test suite. This is one way in which the pursuit of either high coverage (**I1**) or mutation scores (**I3**) can be detrimental since, whenever possible, tests should focus on behaviors of the program under test, not how they have been implemented.

I7: Low "Realism". Tests that do not mimic the way an API, library, or component is used in production may suffer from a lack of realism. This means that the tests may be exercising the program differently than real-world users, potentially leading to false outcomes. Such tests may also contribute to brittleness (**I6**) and require excessive maintenance when the code under test changes. Tests may become unrealistic when they rely too heavily on mocks to simulate parts of a system [4] or test via non-public methods rather than through a public interface [8].

I8: High Variability of Indicators. Variability of indicator metrics is itself an indicator of an unstable and hence potentially unhealthy test suite. Hilton et al. [3] observed that test suites often differ in their coverage levels from run to run. Flakiness (**I0**) itself is a variation in test outcomes. Other indicators such as wildly differing execution times or mutation scores would appear to be signs of undependability, and hence variation of key metrics is potentially an important but largely unexplored area of research in the context of test suite health.

III Test Suite Health: A Research Agenda

Having identified some indicators of test suite health, the question is what to do about them to help practitioners maintain healthier test suites in future development practice. This requires more research. As part of such a research agenda, we contend that several pressing challenges must be addressed:

C1: Further Indicators. The previous section is a list of what we believe to be a good initial starting point for analyzing test suite health. However, there may be further useful indicators that could be added to our checklist, while others may turn out to be less useful than initially thought and could be removed.

C2: Synergies & Trade-offs. We have outlined some synergies and trade-offs between each of the indicators. More are likely to exist, and identifying these will be important for techniques that attempt to measure and/or optimize for overall test suite health. Where there are trade-offs, we need to establish how much a decrease in one metric is tolerable for an increase in another. Following this, is there a Pareto-front of potentially acceptable options, and if so, is there any advice we can give as to which should be preferred by the tester?

C3: Measurability. While several indicators, such as coverage (**I1**) and mutation score (**I3**), are established metrics, others, like realism (**I7**), lack obvious means of quantification. Furthermore, test brittleness (**I6**) is, in its various forms, a serious and costly problem in development practice, but has received relatively little attention in the literature compared to flakiness.

C4: Metrifying Test Suite Health & C5: Making Actionable Recommendations and Fixes. Once we have established which indicators are practically useful, and how they can all be measured, the question then is how to use them all in combination to build a holistic picture of test suite health? While putting a number to it may be useful in some contexts, it is unlikely to be helpful to developers seeking concrete actions on how to go about improving the health of their tests. Future research needs to address this, and link measurements to actionable tasks that a developer might want to perform.

C6: Tooling. There already exist several tools for some of the individual indicators (e.g., coverage and mutation score) and they may help with both of the two previous challenges. However, for others, tooling and more research is required.

C7: How Does Test Suite Health Change? Finally, it would be interesting to study how test suite health changes over the lifecycle of a project. Is it something that gradually deteriorates over time, creating a technical debt in the same way that all code tends to require maintenance and refactoring? Are differing checks and interventions needed at different maturity stages of a project, and are there points in the project's lifecycle where some indicators are more important than others?

IV Related Work

Test Smells [10] characterize poor testing practices. While smelly tests may contribute to an unhealthy test suite, we argue that "test smelliness" and "test suite health" are two separate concepts. On the one hand, test smells tend to be *static* properties related to how *individual* tests are implemented, thereby characterizing bad programming practice. Test suite health, on the other hand, holistically characterizes how well a complete test suite functions in giving developers fast and reliable feedback about the correctness of their software.

Other Work on Test Quality (e.g., [2]) tends to focus on individual tests or factors of "good" tests, ignoring the potential relationships between them. Jason Swett provides a different definition of "Test Suite Health" [9] that also does not consider the interplay between factors; and provides a manual check service. Yet, our vision is that test suite health assessment will be an automated process that provides recommendations to improve the test suite health and potentially automated fixes.

ACKNOWLEDGMENTS. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

References

[1] M. Aniche, C. Treude, and A. Zaidman. How developers engineer test cases: An observational study. *TSE*, 48, 2022.
[2] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan. How good are my tests? In *Proc. WETSoM*, 2017.
[3] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *Proc. ASE*, 2018.
[4] E. Kuefler. Unit Testing. In T. Winters, T. Manshreck, and H. Wright, editors, *Software Engineering at Google: Lessons Learned from Programming Over Time*, chapter 12. O'Reilly Media, 2020.
[5] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *ASE*, 2012.
[6] M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring pseudo-testedness: Empirically evaluating extreme mutation testing at the statement level. In *Proc. ICSME*, 2024.
[7] O. Parry, M. Hilton, G. M. Kapfhammer, and P. McMinn. A survey of flaky tests. *TOSEM*, 2021.
[8] M. F. Roslan, J. M. Rojas, and P. McMinn. Private — Keep out? Understanding how developers account for code visibility in unit testing. In *Proc. ICSME*, 2024.
[9] Jason Swett. https://www.codewithjason.com/test-suite-health-check.
[10] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. Refactoring test code. In *Proc. XP2001*, 2001.
[11] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *ESE*, 2019.
[12] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proc. ISSTA*, 2009.