Minimal Test Practices Needed in a Software Start-up Company

Ossian Gewert*, Astrid Jansson†, Lucas Perlind‡ and Joseph Tafese§
Faculty of Engineering
Lund University
Lund, Sweden
Email: {*dic15oge, †nat15aja, ‡lu3804pe-s, §jo3353ta-s}@student.lu.se

***Abstract*—A software startup is a small organization working with a hyper-focus on developing a software product within its niche in the market. When getting started the company has a need to get a viable product on the market as cheaply and quickly as possible. The excruciating pressures with respect to resources, or a lack thereof, that these organizations face can cause them to replace standard test practices with unsustainable ad hoc techniques that might bring about their demise. It is therefore essential for software startups to reach a certain level of testing while spending minimal effort. In this paper we discuss different approaches to testing and the factors around it. We look at costs of testing, minimally required tests, resource allocation and test practices for startups in this setting through the analysis of available literature.**

I. Introduction

In the context of a Business to consumer software startup, expected to grow significantly over the coming years, we are looking at the minimum test practices needed in building software that they intend to sell to consumers. Their initial goal is to build a Minimum Viable Product (MVP) that they can sell to early adopters. These startups are characterized by having fewer than 10 employees, an emphasis on minimizing time-to-market over product quality, and very limited resources to spend on testing [1], [2]. None, or close to none, of the employ es are dedicated to software testing, which is the responsibility of the software developers [3] who are hopefully highly adaptable and skilled generalists as opposed to specialists with respect to software development. This results in the employment of ad hoc practices that can actually come to hinder future productivity due to accumulated technical debt in the documentation and testing fields [1], [4].

To find what the minimal test practices needed when building the MVP are, we establish the characteristics of startups we are looking at (in Section II-A) and how software testing relates to their context (in Section II-B) with respect to resources (in Section II-C), and analyzed the literature to compile the minimum test practices needed in a startup (in Section III).

II. Background

A. Startups

A software startup is an organization working on the cutting edge of a particular industry, in this case within the software industry, with a hyper-focus on its niche in the market. Companies like these can be found all over the world, but they share a set of characteristics that have come to define them, namely, limited resources when it comes to time, money and manpower [1].

These limitations or opportunities create an environment with interesting requirements and possibilities. Their goal in the early stage is to build and ship an MVP for its target audience. This means that traditional test practices are not feasible to execute since the product can be evolving at a rate that is difficult to keep up with. Large scale changes can also render testing efforts worthless. As a result of the context, these companies will either dispense with structured test practices or rely on ad hoc techniques.

There are a number of reasons to pursue a structured approach to testing, both when testing the programs and when testing the product at large. For example, it is important to ensure that they meet specifications and furthermore to reduce the technical debt incurred by ad hoc practices. Initial short-term gains in time-to-market can lead to significant long-term impact to team productivity and the product's ability to scale.

Although these techniques are likely to be employed in a 'good enough' manner for startup companies, the cost of diverting resources to fully realized testing may be too significant.

B. Testing

Software users will expect that the software works well most of the time they use it, therefore it is important for all software companies to have some quality assurance and continuously maintain their products.

Maintenance is something that could drown a company in work and costs. Maintenance can consist of bug fixes, adaptive maintenance and preventative maintenance. One wants to spend more time on preventative maintenance, rather than fixing bugs as a result of too little preventative effort [5]. The same way the testing where issues are found in an earlier stage improves the chances for reduced cost in the long haul.

Testing a program can be approached in four phases that offers a structure to relate the problems to be solved, namely, modeling the software environment, selecting test scenarios, running and evaluating test scenarios and measuring testing progress [6].

The modeling of the software environments requires the identification and simulation of the interfaces of the program and also the establishment of a number of inputs that might cross over interfaces. The interfaces can include APIs, file systems and communication interfaces. In the case of our software startup company the environment could mean different platforms (e.g. iOS or Android) or browsers (e.g. Firefox or Chrome). User interaction that fall outside of the control of the code under test can be explored in this domain as well.

As there is a possibility of an infinite number of test scenarios, there is a need for a test scenario selection phase of the testing design. We can refer to coverage of the code or input coverage and still have an infinite amount of test cases to deal with. Therefore, the test data adequacy criteria arrives as the way to test enough cases in a reasonable and affordable manner. These can be used to make decisions on when to release [6]. In the case of the startup, if it is known what functionalities and criteria the product depends on, relevant decisions can be made on how much testing is enough. However, testers should be aware of the limitations of the criteria being built into the methodology and the achieved report results that come forth when testing.

The phase of running and evaluating test scenarios regards to how the tests are executed and what methods are used. Using simulation of all input and output for automating the test execution is a good way to minimize the risk of mistakes and save time. Scenario evolution, the comparing of the actual software output from test scenario execution, is harder to test and here we expect the code to be flawed rather than the documented specifications. As for these issues, there are two approaches that can be used to evaluate the test, embedded test code and formalism. Embedded code can be self-testing programs or a simpler version where some internal data states are exposed so as to make it easier to validate. Formalism refers to making sure that the design and code are derived from the formalized specification. The specification takes effort to create, costs time and money a startup might not be considered to have. It is however helpful down the line and saves time in regard to the evaluation of reported bugs and writing test; In other words, it is a preventative effort that, if properly used, makes maintenance easier.

A fix to a bug can only be a solution to that specific issue and can potentially break something else in the process. Therefore, regression testing is a way to check that the program as a whole is still functioning properly [4]. However, this can be time-consuming and therefore the testing needs to be prioritized and minimized. In regression testing the test data, adequacy criteria are ignored as only the absence of fault is sought [6].

Testing contributes value to a software project by detecting faults or validating correctness. The value of a test practice could imperfectly be quantified by the percentage of faults detected but collecting detailed fault information is likely to be expensive [7]. Using coverage is one way to measure the tests. However, detecting many flaws is not necessarily of benefit. Larger amounts of tests that have failed and been fixed can either mean that we have tested well and few errors remain in the system or that the system in general has many flaws. In essence this is the challenge: making sure that the tests written provide value as well as knowing when to stop testing.

C. Costs of testing software

Testing requires significant effort and is one of the biggest costs in software development [8]. Many software companies find that they have excessive testing costs or that their testing is not cost-effective [9]. The costs consist both of human effort and machine time, with cost components including identifying a test model and the test requirements, implementing the tests, test execution and test diagnosis [7]. The available time and budget limits the testing that can be done in a software development project [7], with the lack of resources for testing being especially constricting for start-ups [10]. The human effort could translate either into increased development time or an opportunity cost when other efforts are deprioritized. If the machine time required to run the test suite is too large, developers might skip tests or run them less often [10].

Comparing the cost of different testing approaches before implementing them is difficult due to problems with cost estimates. The cost of a test suite is often assumed to be proportional to its size but using size to compare costs between different test techniques is not likely to be valid due to the large number of cost components [7].

Automating testing is a common strategy to minimize the cost, but manual testing cannot be eliminated [8]. Manual testing continuously requires human effort, while automated testing requires human effort when creating the test cases and then some human effort to maintain them [10].

How much to invest in testing is a trade-off between keeping initial costs down and increasing the software reliability, which is likely to return long-time value [11].

III. Analysis

A. Minimally required tests

When measuring test progress in the context of a startup company, the restrictive nature placed on development teams, due to the lack of resources must be taken into account. Startups will often need to prioritize development of features rather than testing and maintenance of the current code base.

Thus, a startup company needs to strategically select how to test and what to test, namely selecting test scenarios. This is a significant task and can greatly reduce the overall amount of testing required while still maintaining the quality of the software. Functions more critical to the product's function can be prioritised when applying effort to creating tests. This can be analyzed by the designer or by using analysis techniques such as retrospective mining [10]. Measures of an artifact's functionality through usage and functional consequences can contribute to perceived value of that artifact, with those of higher value to have more tests written for them. Such processes can be done in an informal manner or be more structured through frameworks focused on the testing process. Many models/frameworks to help outline how testing

should be done and how much is required exist today to aid development teams in improving their testing suites.

Popular models include the Testing Maturity Model (TMM) and the Test Process Improvement (TPI). Unfortunately, these frameworks are too expensive and large for a small team to realistically follow, in response smaller frameworks targeted towards smaller businesses and startups have been created. The Minimal Test Practice Framework (MTPF) [3] was designed to cut the amount of testing practices required to be done by a development team in comparison to the larger models, while still maintaining the core principles of these frameworks. These practices are staged by team size, so that, as the startup's development team grows and their product expands, more systematic and broad testing practices can be implemented. This way the testing can scale with the amount of resources the startup has, encouraging the company to not overspend into testing in their initial stages. Additionally, it ensures that when their capabilities and resources expand, testing practices increase to match the scope. The MTPF is significantly less expensive and resource intensive compared to more traditional models; making it a much greater and more accessible choice for structuring the organization in terms of its test suite. Furthermore, it can help reduce the amount of technical debt generated by potential poor decision making if a more formalised structure is not followed. This can provide tangible long-term savings for the company as the software product(s) continues to age and mature.

It should be noted that testing within the startup will largely be based on a 'good enough' mentality. Thus, any selected strategies will likely be implemented not in its entirely but in a way to achieve the core principles of the practice.

B. Allocating resources

As a company needs to allocate resources towards the testing effort, considerations need to be made when selecting test scenarios. Due to the potential significant costs of testing; these considerations need to be taken with consequential gravity.

When running tests, developers can choose to employ automated and/or manual forms of testing. One could assume that 100% automation would be a valid long-term cost saver to testing. Especially in the context of a startup with limited human resources, reducing the required man hours to perform tests. In reality such levels of automation are impossible to achieve and in fact has its own drawbacks in the testing process [8]. Thus, a combination of manual and automated testing could be implemented in any testing suite to achieve a less costly and effort efficient solution. A comparison of automated and manual testing can be seen in Table I.

Table I
AUTOMATED TESTING VERSUS MANUAL TESTING

Automated testing	**Manual testing**
Reduces costs of repeated tasks	Expensive and time consuming to perform repeated tasks
Hard to analyse qualitative testing	Qualitative metrics are easy to asses
High initial costs with long-term savings	Low short-term costs although expensive for larger projects
Initially requires manual effort	Can create automated tests

Such automated testing can be implemented through third party tools; usually in a software package where testing is done through a user interface (UI). [8] reasons that testing should be done at as low level as possible. Automated tests set up through a UI tend to be more prone to failure and require more effort in maintenance. Furthermore, having developers and testers at the same lower level of abstraction with the code can improve the machine costs for running tests and can decrease implementation costs of these tests [8]. Thus, startups that do not have a dedicated testing team could follow a strategy of seeing the developers performing the tests themselves.

Further considerations to select a specific test report is to recognise the strengths and weaknesses of each test. Teams should value test practices that align greater towards their overall software testing model. Studies and papers must be scrutinized for their viability for the project [7]. A project heavily using functional programming practices may not want to implement tests designed for testing software designed in an object orientated manner.

Furthermore, the found efficiency and effectiveness of a test have been to host a great deal of randomness [7]. Human factors account for variation in the testing suite. We should not assume humans are perfect, especially in using practices and techniques that are more challenging to implement. With these aspects in mind, a startup can more strategically select testing practices that will provide them with the most value from the pool of limited resources.

C. Ad Hoc Testing

Without any formalized testing structure a development team would resort to ad hoc testing. This involves completely random processes by the developers. With nothing formalized tracking issues and faults in the code is largely impossible. Reducing the confidence that the software is properly functional alongside increasing technical debt of the project [4]. Thus, to ensure the products quality more structured testing processes should be followed and implemented in the testing suite.

D. Selecting Test Scenarios

Keeping in mind that the goal of software testing is twofold, finding defects and demonstrating program correctness [7], there are practices that can be utilized in a startup to generate test cases for a given product.

An effective method of test case generation, based on program requirements, can be done before any code is written. Equivalence Partitioning and Boundary Value analysis are black-box techniques that provide a specification for the input and output classes for each method/function based on the program requirements. The biggest hindrance to this process

in a startup is that requirements are made up on the go and only validated after the product is released, nonetheless, the practices can be implemented in smaller iterations through the development cycle [1].

As noted above, a common challenge in a startup's development cycle is the changing requirements, but more importantly, the rate at which the requirements change. Building on the assumption that the startup's goal is to build an MVP, there is a process mining approach to test case generation and prioritization [10]. This method is effective when it comes to identifying what parts of the program experience the highest traffic as well as points of failure, which can be valuable information for generating and maintaining test cases for a given program. The knowledge of high traffic aspects of the program does not mean that parts of the program with low traffic are any less important, but it gives insight into what might need an extra eye before being released to the user base. The process mining approach can also be used to find more efficient solutions to the products user experience [10] since it is able to identify these "pain points", information which is critical to a startup in the MVP phase.

E. Defect Reporting

Developing a uniform syntax and method of communication within the startup is quintessential to the defect reporting and tracking process. A clear understanding and structure to this process, however primitive it may be, will enhance the efficacy of development efforts [3].

Efforts can range from maintaining a string of emails holding defect reports, a common file that is written to or a defect reporting system such as Jira. Nonetheless, the focus lies in the common syntax, which can be understood by everyone on the team, so that any member of the development team can contribute to a solution. A streamlined method of reporting defects is also fundamental to decreasing the amount of technical debt incurred [1] since knowledge of defects and shortcomings is made readily available to any interested party in the organization. Furthermore, an emphasis on implementing and practicing preventative measures with respect to documentation debt, a subset of technical debt, has been shown to have a statistically significant impact on the startup's future productivity [4].

Finally, a functional defect reporting system reduces the reliance on individuals withing the company as knowledge bearers [1]. Lack of proper documentation practices becomes apparent when for example minor changes to the composition of the team hinders the productivity of everyone else.

F. Test Types

A startup can benefit from the implementation of different test types, such as tests for correctness, with respect to product specifications, and market viability of the product. Regression and unit tests [4] as well as A/B Tests are cost effective implementations of the respective test types. Test cases for unit tests can be generated via the aforementioned methods and be run at least before every release. Successful tests can then be added to the suite of regression tests, although the constant changing nature of requirements in this environment can make it a challenge to maintain a set of regression tests [4].

The market viability or acceptance of certain features can also be tested using freely available A/B Test suites [1] such as Facebook's Planout. Due to the startup's lack of resources, building the scaffolding of a feature alternative and pushing them to different user groups can provide valuable insight into what the next steps should be. This can save money, time and effort since it can be a relatively simple way to validate a feature idea without pouring resources into an unverified lead. Moreover, A/B tests allow the startup to explore multiple viable additions/modifications to their product without incurring technical debt from failed pursuits. This is of particular importance since the code base is not infiltrated with exploratory code that lingers, due to the simple structure of the tests to begin with [1].

IV. Conclusion

Software startups developing an MVP are under significant time pressure to get their product onto the market ready for their first customers. They have few employees, none of which may be designated to do testing. The startups are also likely to lack the financial resources to purchase significant outside tools or assistance. Due to their lack of resources software startups have to deprioritize numerous areas of the development cycle, with testing often being one such area.

Testing software is important to ensure that the software works as expected and meet customer demands of quality. It is necessary to keep the long-term maintenance costs down if a software company will succeed. How much to invest in testing is a unique, per-company trade-off depending on the available resources and the requirements on the product. The requirements will depend on the type of product, the goals of the product, the type of customers and competing solutions.

Testing practices are only as good as they can be if followed. Therefore, the importance of choosing how to approach it is of essence as it can define a company's work for a long time. The approach the company takes need to fit their people and their style of working. Even if the work done is agile, or testing done by developers or dedicated testers, but a structure regarding the requirements and goals are of essence.

A combination of manual and automated tests can be utilized for the MVP's correctness, and A/B testing frameworks can be employed to the development cycle to provide a structure to the flow of features. Manual testing takes continuous human effort, while automated testing should require less human effort over time, but a healthy collaboration of the two can provide the best results for the resources spent.

Although testing debt has not been shown to have a statistically significant impact on a startup's future productivity, documentation debt does impact future productivity. This implored the focus on a combination of testing and documentation practices that would be necessary for a startup

to grow rapidly, without severely impeding its future growth, while delivering a "good enough" feasible product.

V. Contribution statement

The authors made equal and significant contributions to the project and this resulting paper.

All group members participated in an initial brain-storming session and later meetings with literature review, project planning and follow-up. All group members found relevant literature and jointly drafted an initial outline. All group members drafted subsections of this paper and contributed to subsections originally written by the other authors.

References

[1] C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, "Software Development in Startup Companies: The Greenfield Startup Model," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 585–604, Jun. 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7360225

[2] C. Giardino, M. Unterkalmsteiner, N. Paternoster, T. Gorschek, and P. Abrahamsson, "What Do We Know about Software Development in Startups?" *IEEE Software*, vol. 31, no. 5, pp. 28–32, Sep. 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6898758/

[3] D. Karlström, P. Runeson, and S. Nordén, "A minimal test practice framework for emerging software organizations," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 145–166, Sep. 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.317

[4] E. Klotins, M. Unterkalmsteiner, P. Chatzipetrou, T. Gorschek, R. Prikladnicki, N. Tripathi, and L. Pompermaier, "Exploration of Technical Debt in Start-ups," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, Gothenburg, Sweden, May 2018, pp. 75–84. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8449238

[5] P. Grubb and A. A. Takang, *Software Maintenance: Concepts and Practice*, 2nd ed. World Scientific, 2003.

[6] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70–79, Jan. 2000. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/819971

[7] L. C. Briand, "A Critical Analysis of Empirical Research in Software Testing," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, Sep. 2007, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4343726

[8] R. Kazmi, R. M. Afzal, and I. S. Bajwa, "Teeter-totter in testing," in *Eighth International Conference on Digital Information Management (ICDIM 2013)*. Islamabad, Pakistan: IEEE, Sep. 2013, pp. 194–198. [Online]. Available: http://ieeexplore.ieee.org/document/6693991/

[9] V. Garousi, M. Felderer, and T. Hacaloğlu, "What We Know about Software Test Maturity and Test Process Improvement," *IEEE Software*, vol. 35, no. 1, pp. 84–92, Jan. 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8239941/

[10] A. Janes, "Test Case Generation and Prioritization: A Process-Mining Approach," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Tokyo, Japan: IEEE, Mar. 2017, pp. 38–39. [Online]. Available: http://ieeexplore.ieee.org/document/7899028/

[11] C.-Y. Huang, S.-Y. Luo, and M. R. Lyu, "Optimal software release policy based on cost and reliability with testing efficiency," in *Proceedings. Twenty-Third Annual International Computer Software and Applications Conference*, Phoenix, Arizona, Oct. 1999, pp. 468–473. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/814328