How do Incremental Software Development practices affect the overall stability of the project code base?

COMP130 - Game Architecture & Engineering

1804356

March 17, 2019

1 Introduction

Incremental Software Development (further referred to as IDS) is a model for developing software that relies on incrementally prototyping, designing and implementing pieces of software in steps towards an end product. However, one pitfall commonly discussed [1]–[4] is the overall reliability of the newly-implemented pieces of software in the code base. In other words, as the project advances, there is a tendency to re-visit previously implemented code in order to re-factor because of incompatibilities or poor design decisions that only became visible later on in the development cycle. In the present case (the author's current project), for instance, it was noted that such impediments led to several code-base-wide refactoring and re-designs that obviously slowed the development process. However, it can be argued that more time spent designing instead of prototyping or actively developing would have been sufficient to prevent such outcomes, both in the presented case and in general [5], [6]. Consequently, the author would like to present the paper's question in the following manner: Using IDS practices, how would one best balance the 3 stages of development -designing, prototyping, developing- each cycle, in order to reduce the maximum number of times specific pieces of code need to be revisited and adapted to new changes?

2 Introducing the problems

This section focuses on introducing the most common issues associated with software development in the context of IDS. Stemming from several papers and case studies are issues that, while seemingly not directly connected to code base reliability, actually play a significant role in the maintainability of code, as well as the team's productivity and the need for code refactoring across the code base.

2.1 Code reliability

One prominent feature of IDS practices is the pace at which new features are introduced. This is due to its very nature and principles of continuously adding new features to the code base. However, this inevitably leads to issues within the code base, which have been noted by several case studies. One such case study[5] mentions the importance of the "order of requirement", or the order in which new features are implemented, in accordance to the product requirements, with the addition that "refactoring is indispensable". In other words, the need for code refactoring becomes a risky[7] certainty, but the amount of work needed to refactor the code base will vary based on the order of the features implemented. Consequently, it can be argued that the amount of time spent refactoring can be correlated to the amount of time spent designing and planning each feature before its implementation within the code base. This is of significant importance, especially in the context of games' development, where the process of designing features is more arduous, and planned features can be scrapped or added in quite the sudden manner.

2.2 Effort estimation

Effort estimation can represent quite the challenge in most software development projects, and it is argued that one specific challenge is the estimation of changes between individual releases of software systems[8], while [4] argues that, at least in the case of UI development, wasted effort poses a real challenge due to the possibility of reverting changes between releases. IDS somewhat circumvents this issue through its cycle-based release of software features. However, this implies a well-determined milestone for each cycle that presents little room for changes. Finally, [9] presents a small case study on a software development project that reduced time and effort spent on testing in order to balance it on initial analysis of the product. The study goes on to note that in the following year, only four software problems needed required fixing. This goes on to show the significance of balancing time and effort, as well as the importance of flexibility when deciding what that time and effort should be spent on. Consequently, it is crucial to any software development team to be able to balance their efforts between planning, prototyping, developing and testing in order to achieve an optimum level of efficiency. In the context of IDS, however, a team will have to balance their time and effort during each development cycle, according to the planned features, while leaving little room for any major deviations, which can become an issue when certain features require more time to implement.

2.3 Productivity trends

The overall productivity of a team is influenced by a variety of factors, such as staffing stability, design compatibility and adaptability, and integration and testing[3]. It is a somewhat common occurrence for staff members to either leave or join a team during the development process. However, this has a significant impact not only on the overall team cohesion, but also on the overall output of the team over a period of time[3].

As for the overall feature implementation, the same study cited beforehand presented notable results regarding effort distribution, as the design effort was steady up until the point where a major reconstruction occurred. This was mainly due to the nature of the project, as the team was making use of IDS practices and a major code refactoring is nigh unavoidable. Furthermore, the point is made that while refactoring will help to maintain the productivity, its effect will not last too long. This is especially true due to the fact that individual features are implemented each cycle, some of which might not have many tangents with one another. Finally, the integration of new features becomes more and more difficult as more components and modules are added to the system. While seemingly obvious, it is important to note this, especially given the importance of module decoupling [4]. Two of the main causes found for this increased difficulty, as described by [3], are the lack of design for integration and code breakage and the increasing necessity of testing and fixing features in order to assure a good code cohesion. The design-specific issues can stem from a variety of factors, one of which is the uncertainty of what features will be scrapped and what features will be added to the end product, which can easily break the overall design of the software being developed. As for the code cohesion and the need for testing, due to the nature of IDS practices, it is almost certain that as more features are implemented, more prototyping and testing will be required to assure the optimal functionality of the software. Given the mixed and flexible nature of most game development teams, such issues are far more likely to appear within, and therefore the overall productivity will vary greatly from development cycle to development cycle.

3 Possible solutions

This section focuses on providing possible solutions or insights into how the problems presented can be addressed, from either academic sources or the personal experience of the author.

In terms of overall code reliability, the first thing to consider when working with IDS is how time and effort can be balanced between designing, prototyping, implementing and testing features. [9] presents the possibility of reducing testing time and focusing more on the design and prototyping phases of the development cycle. Personally, the author believes that while rigorous planning and designing will certainly increase the overall code stability and may be able to further reduce module coupling, it should be noted that design changes are bound to occur. Such changes are practically impossible to foresee, as they stem either from design oversights, feedback or last-minute changes in requirements. Therefore, one possible solution would be to maintain a real-time design of the product and maintain it to reflect the latest changes in design. For instance, UML diagrams can be used to more easily display the design of larger, more complex products, while being able to be focused on any specific aspect of the product: class hierarchies, objects, use cases, sequences etc.

While balancing time between the different stages of software development can increase the overall efficiency of the team, effort estimation plays an equally important role in the long team development of a software-based product. In other words, it is the belief of the author that a team that is able to balance their efforts efficiently will deliver a better overall product. What teams ought to lookout for is the trap of wasted effort: implementing entire features or modules that will end up being scrapped. In order to avoid such situations, the requirements for every development cycle should be as clear and concise as possible, and should not contain features that are likely to not make it into the end product. This comes down to the design of the product, and how well it was structured during the initial design phase.

As for the changes in productivity of the team, these can be caused by many different aspects. First of all, changes in personnel will likely cause delays while the new members are accustomed to the development environment and the overall project structure. Such changes are far more prominent in the games' industry, where teams are often faced with the addition of new members, as well as members leaving the team. It is often the team manager's responsibility to maintain the cohesion between individual members, as even the addition of a single member can have a significant effect on the overall productivity of the team. From the author's personal experience, team members leaving the team can even create tension between the team, as the scope of the project might need re-visiting in order to adapt to the changes.

Secondly, in the case of code refactoring, the point was made that while it certainly improves overall module de-coupling and code base stability, it is often the case that such measures take increasingly long periods of time and amounts of effort, and their effects are decreasingly visible, the author would like to point out. This is especially true in the case of IDS, because after each cycle new features are to be added to the code base and the need for code maintenance increases. The author would like to note that in their case, setting aside time for designated code revisions helped with organising their own contributions to the code base, but also with maintaining a good level of code base stability throughout their project.

Finally, the author would like to mention that during their own experience, they have noted the crucial importance of balancing out the rate at which new features are implemented in the code base with the overall quality, readability and overall stability of newly-added pieces of code. While certain projects demand more accent put on the quality of the code, it is often the case, at least in the games' industry, that the speed of implementing new features takes priority over the quality and readability of code.

4 Conclusion

The aim of this paper was to present what are the most important issues to consider when using IDS practices in the context of software development and reflect on how they can be fixed or outright avoided in their entirety. From staff stability and design choices, to balancing effort distribution and productivity changes, this paper has tried to touch on all these issues while staying true to the context of games' development. The findings of this paper are also backed by the author's personal software development experience, as that was precisely what represented the motivation for writing the paper in the first place: a way to present and suggest ways to address some of the issues presented by IDS practices when developing software of any kind, but with an accent on games' development.

References

[1] T. Fujii, T. Dohi, and T. Fujiwara, "Towards quantitative software reliability assessment in incremental development processes", in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 41–50. DOI: 10.1145/1985793.1985800.

[2] D. R. Graham, "Incremental development and delivery for large software systems", in *IEE Colloquium on Software Prototyping and Evolutionary Development*, Nov. 1992, pp. 2/1–2/9.

[3] T. Tan, Q. Li, B. Boehm, Y. Yang, M. He, and R. Moazeni, "Productivity trends in incremental and iterative software development", in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Oct. 2009, pp. 1–10. DOI: 10.1109/ESEM.2009.5316044.

[4] D. Liu, S. Xu, and W. Du, "Case study on incremental software development", in *2011 Ninth International Conference on Software Engineering Research, Management and Applications*, Aug. 2011, pp. 227–234. DOI: 10.1109/SERA.2011.43.

[5] S. Ikemoto, T. Dohi, and H. Okamura, "Estimating software reliability with static project data in incremental development processes", in *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*, Oct. 2013, pp. 219–224. DOI: 10.1109/IWSM-Mensura.2013.38.

[6] P. Trivedi and A. Sharma, "A comparative study between iterative waterfall and incremental software development life cycle model for optimizing the resources using computer simulation", in *2013 2nd International Conference on Information Management in the Knowledge Economy*, Dec. 2013, pp. 188–194.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*, en. Addison-Wesley Professional, Nov. 2018, Google-Books-ID: 2H1_DwAAQBAJ, ISBN: 9780134757704.

[8] P. Mohagheghi, B. Anda, and R. Conradi, "Effort estimation of use cases for incremental large-scale software development", in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 303–311. DOI: `10.1109/ICSE.2005.1553573`.

[9] P. E. Ross, "The exterminators [software bugs]", *IEEE Spectrum*, vol. 42, no. 9, pp. 36–41, Sep. 2005, ISSN: 0018-9235. DOI: `10.1109/MSPEC.2005.1502527`.