Software entropy in agile product evolution

Geir Kjetil Hanssen
NTNU IDI / SINTEF ICT
ghanssen@sintef.no

Aiko Fallas Yamashita
Simula Research Laboratory
aiko@simula.no

Reidar Conradi
NTNU IDI
Reidar.Conradi@idi.ntnu.no

Leon Moonen
Simula Research Laboratory
leonm@simula.no

Abstract

As agile software development principles and methods are being adopted by large software product organizations it is important to understand the role of software entropy. That is, how the maintainability of a system may degrade over time due to continuous change. This may on one side affect the ability to act agile in planning and development. On the other side, an agile process may affect growth of entropy. We report from a case study of a successful software product line organization that has adopted the agile development method Evo, showing how agility and entropy are negatively related. We conclude this study by suggesting a two-step approach to manage entropy while maintaining process agility. First, the system needs to be restructured to establish a level of manageable entropy, and then, that the agile process must be complemented with continuous semi-automated quality monitoring and refactoring support.

1. Introduction

Agile software development methods have over the past decade become a preferred approach to many project organizations and are now also finding their way into the more complex arena of packaged software product development and evolution [1]. This comes along with several challenges as the complexity of the development context increases dramatically as compared to the development of a single stand-alone solution for a single customer – which in many respects was the target of agile methods initially. One of the challenges when adopting agile software development practices to large-scale product development is the problem of software entropy, which refers to the gradual decrease in maintainability of a system as it evolves over time. With the core principles of agile software development in mind we suspect that development process agility and system entropy may negatively and naturally affect each other. That is, the rapid, incremental and iterative approach of agile development methods may actually aggravate entropy, and the other way around; that an increasing system entropy may hamper the ability to act agile.

To investigate this potential relationship and improvement actions we have done a case study of a successful software product organization that have developed their product line for over a period of 14 years and are now established in the top segment of their domain, still with a strong need and urge for further development and improvement. Five years ago the R&D department experienced severe problems related to a heavily plan-based development process and adopted the agile method Evo [2] which over the past years have proved to fit well into a hectic software product development scheme with constant releases of new versions into a competitive market [3]. However, for each release of their product, the internal structure of the product have grown more and more complex and are now a major concern as it dramatically threatens their ability to be agile in terms of reduced analyzability, modifiability and testability of the system as well as problems in separating areas of concern for the development teams. In sum this leads to a reduction in productivity and product quality.

This motivates our research questions:

1) How may system entropy and agile processes mutually negatively affect each other?
2) Can code smell analysis and refactoring be a viable solution?

We have collected data from three sources. (1) We have done an extensive interview with members of a dedicated product architecture team – a group of four expert developers serving the development teams and being responsible of improving the architecture of the product line as well as optimizing the development infrastructure. (2) We have held a workshop with an external consultant analyzing the system using a comprehensive tool called NDepend™. (3) We have interviewed members from one of the development teams in the R&D department. The interview data are analyzed according to the principles of constant comparison [4] revealing a set of problems related to the state of entropy of the system as well as the agile process as it is implemented in this case.

Based on the findings from this qualitative study and on an overview of relevant literature on code smell analysis and refactoring decision-making we discuss how these problems potentially can be resolved in a two-step process. First, the state of entropy must be reduced to benefit from the agile process. Then, having established a level of manageable entropy, proper actions (changes in process) must be taken to avoid entropy from growing again.

This paper extends an initial short-paper reporting from the study of CSoft [5]. In the remainder of the paper we first describe our research approach in chapter two. Chapter three gives an overview of relevant literature covering software entropy, code-smell analysis and refactoring decision-making. Chapter four present our findings from the case study, which is then being discussed in chapter five. Finally, we answer the research questions and give some concluding remarks in chapter six.

2. Study context and methodology

2.1. Case study context and background

CSoft (anonymized name) is a medium-sized Norwegian software company that develops, maintains and markets a product line having the same name. They serve the high-end segment of their market and have a wide international customer base. Despite having considerable challenges CSoft is now one of the market leaders. The company was established in 1996 and has grown steadily since, currently employing about 260 people, including 60+ developers. They have several development locations across Europe and Eurasia and there has been a gradual shift from building custom-made applications to a software product line. CSoft can be seen as a highly modular product line that allows many configurations and ways to use it. It contains five main modules (with numerous sub-modules). The use of these modules varies per customer and per case. Some central modules are used in any configuration, while the use of others depends on the usage situation. CSoft comes with a set of predefined configurations for the most common usage scenarios, but there is also built-in support for detailed customization to support more variants. From the start of the company, fourteen years ago, the development process matured from a more or less ad-hoc type of process (creative chaos) to a well-defined waterfall-inspired process (plan-based and non-iterative).

About five years ago the development process had become too slow and inefficient. Out of necessity CSoft changed to a radically different process: Evo [6]. This change was guided by Tom Gilb, who originally defined the process [2]. Evo is an agile method comparable to the better-known Scrum-method [7], although the terminology differs. At CSoft, work is done in two-week iterations (equivalent to sprints in Scrum), working software is deployed on test servers by the end of every iteration and invited customers evaluate the latest results and give corrective feedback to the development teams [3, 8]. Although similar to other agile methods, the perhaps most distinct characteristic of Evo is the strong focus on product qualities and the definition and use of metrics for expressing evaluating quality level goals. As adopted at CSoft, Evo conforms to the four basic values in the Agile Manifesto (see www.agilemanifesto.org): interaction is highly valued, there is a strong emphasis on delivering working software after every iteration, invited lead users participate in development, which is open to changes in requirements and design.

2.2. Study method

We have collected empirical data in three ways: First, the company had a workshop with Patrick Smacchia, an external consultant who analyzed the CSoft source code using his own commercial tool called NDepend™[1] which was used to do a live analysis of the code – numerous metrics were generated on the fly, presented graphically and the code was browsed alongside a discussion in the workshop group. Based on this "live" analysis of the system the participants (system architects, developers and researchers) developed through discussions a common understanding of the state of entropy of the system. A brief summary of this discussion is given in [9]. Secondly, we conducted an in-depth interview with two members from the four-person product architecture team. This group has two main responsibilities: (a) to ensure and improve the architecture of the system, that is, to make it easy and safe to add and improve features and to ease deployment of the product, and (b) to improve the system's development infrastructure (testing framework, code management, automated builds etc.). The interview lasted for 3,5 hours and was recorded and transcribed. This transcription (30 pages of text) was analyzed using NVivo™, a tool for tagging fragments of text with information about context, meaning etc. Finally, we also did interviews with the

[1] See http://www.ndepend.com/

leader and one developer from one of the module teams.

2.3. Threats to validity

Our case study covers only a single software development organization and the interviews were done with a few representatives from the central architecture team of four and just a few of the members of R&D. Also, the situation described and discussed here only reflects the present state and not details on how the situation have developed over time. However, findings from interviews are supported by data obtained in the workshop with the external consultant and the dynamic analysis of the code, together this gives us valuable insight into the complex problem of entropy and its reciprocal influence with the agile development process. We believe that the results and the discussions in this paper may be of practical value as well as a pointer to further research.

3. Background

3.1. Agile development and code entropy

As early as 1976 Belady and Lehman [10] defined what they call a set of laws of program evolution dynamics. Their second law – 'The law of increasing entropy' say: *The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.* This notion of the entropy problem has been followed up by later research on software evolution and maintenance. One notable example is a much-sited study by Eick et al., which investigated a long backlog of change history of a very large software system for telephony. On one hand, it demonstrated decay/entropy as a natural process. On the other hand it identified a set of useful symptoms or predictors of decay [11]. The entropy-problem have recently also been related to agile software development – one of the most notable ideas to software engineering over the past decade [12]. For example, Martin Fowler discusses evolutionary design as a possible cause for software entropy – if not managed [13]. As a countermeasure he promotes simple design and refactoring. As we will show by our case study – this is truly necessary but not easy in complex situations such as the constant evolution of a software product line. This view is supported by other studies, for example by Oizka that summarizes that "..refactoring proved to be much more difficult and time-consuming than expected." [14]. Rajlich defines the industry's interest in, and shift towards, agile methods, as a paradigm shift in software engineering and that attention to actively managing software entropy is a vital success factor [1]. Missing this focus may actually shorten the lifetime of a software product as a decaying system is eventually impossible to evolve, consequently moving into a phase-out stage. Neill and Laplante develop the refactoring concept further and define what they call strategic refactoring [15]. This is an approach, starting out with looking for code smells, where also macro- and domain-level architecture is being evaluated (and refactored).

3.2. Code smell analysis and refactoring

In a thorough search for relevant and rigorous empirical research we have identified 11 papers addressing code smell analysis and refactoring covering, broadly, four sub categories:

(a) Subjective evaluation of code smells. In [16] Mäntylä et al. report from an empirical study of subjective evaluation and detection of code smells and compare it with automated metrics-based detection. The study was done in an industrial setting and showed that subjective evaluations by developers were not uniform. However, in cases with a low level of problems, the conformance was higher than cases with a high level of problems. When investigating the demographics of the evaluators they saw that experienced developers were better at spotting structural problems in the code than regular developers who could spot problems mainly at the code level. Also, developers that had worked with the code for a long period of time tended to see fewer smells than developers with shorter experience. Finally, when comparing subjective evaluation of code with automated metric-based detection of code smells, they discovered that developers' evaluations of complex code smells did not match the results of the metrics based detection. Based on these findings they conclude that subjective evaluations and metrics based detection should be used in combination.

Mäntylä et al. also reports on a student experiment for evaluating subjective evaluation for code smells detection and refactoring decisions [17]. He observed the highest interrater agreements between evaluators for simple code smells. When the subjects were asked to make refactoring decisions he observed low agreement, thus questioning the reliability of such.

(b) Refactoring and refactoring decisions. Counsell et al. investigated refactorings done in seven open-source Java systems to see which types of refactorings were most common and which effects they had in solving code-smells [18]. The study used fifteen refactorings from the classification by Martin Fowler and Kent Beck [19]. The analysis showed that a group of six refactorings were more commonly used: Pull Up Method, Move Method, Add Parameter, Move Field, Rename Method and Rename Field. Surprisingly, these

most common refactorings were not addressing inheritance or encapsulation. Two of the refactorings, Move Method and Move Field, seemed to solve several code smells. In another study by Counsell et al. [20] the same code smells were used on the same data-set to investigate indirect and composite refactorings. Three refactorings were found to have large chains of following refactorings (Encapsulate Downcast, Extract Subclass and Extract Superclass). Refactorings inducing long chains tended to be used relatively infrequently by developers as opposed to refactorings inducing short chains. In a third study, Counsell at al. [21] used the same refactorings and the same empirical data set to investigate how refactorings affected the testability of a system. That is, to what extent the fifteen refactorings affect the re-usability of a test suite, i.e. having to update the tests is a spin-off cost of refactoring. The main conclusion from this study is that while semantically preserving refactorings may be ideal for preserving tests sets, they are not necessarily always the right refactorings to choose.

(c) General applicability of code metrics. We identified one study investigating the applicability of code metrics across different software systems. Bakota et al. [22] collected code metrics for four software systems: an OSS system vs. a closed source system and an office application vs. a telecommunication system. They found that the systems could be differentiated from each other pretty well based on the metric values, but remark that two metrics "*Response For A Class"* and "*Weighted Methods Per Class"* behaved very different on the systems analyzed. Somewhat related, Li and Shatnawi investigated how well code smells can predict post-release class defects [23], results showing that the *Shotgun Surgery*, *God Class* and *God Methods* bad smells were positively associated with the class error probability.

(d) Code cloning. Two studies address problems related to code cloning. Aversano et al. [24] studied code clone evolution by combining clone detection and co-change analysis, concluding that either for bug fixing or for evolution, most of the cloned code is consistently maintained during the same co-change or during temporally close co-changes. This finding seems to somewhat demystify the image of code clones being bad design. In the same line of research, Lozano and Wermelinger [25] report the results from an experiment to investigate the effects of code clones on maintenance. Their analysis suggests that existence of code clones does increase maintenance efforts, at times significantly, depending on code characteristics. However, they were unable to identify characteristics that systematically revealed a significant relation between cloning and maintenance effort increase.

4. Findings from the Case Study

This section presents an overview of data collected during the case study. It describes (i) the structure and the complexity of the product being developed by CSoft (ii) the problems that this structure imposes and (iii) ideas that the architecture team themselves has for improving the situation.

4.1. Complexity of the system

The system has been under constant development for the last fourteen years and is based on several technologies that have emerged over those years. Aging solutions from years ago are still part of the system, such as older ASP solutions, COM+ components, VB6 code and other legacy technologies. Today, most new code is developed in C#, and is spread over approximately 160 .Net assemblies. The complete product is best described as a traditional three-tier system with an MS SQL Server driving the data layer, a business layer and a presentation layer based on a dozen ASP.Net applications. There is a clean separation between the presentation- and the business layer. However the most obvious problem in the software is what the architects refer to as "the Blob": a very large assembly (aptly named Core) consisting of approximately 150K lines of code in 144 namespaces. The NDepend tool was used to visualize and generate code metrics for the internals of this assembly. One of the results was a so-called *dependency structure matrix*, which showed an extremely entangled structure, where most namespaces refers to most namespaces, thus creating a lot of cyclic dependencies. NDepend also provides means for additional analysis through CQL (Code Query Language) [26] which in name and function is inspired by SQL. The code is the database and structured queries can be defined to investigate the code and its internal relationships. CQL retrieves instances of classes, which display certain characteristics from a code metrics perspective.

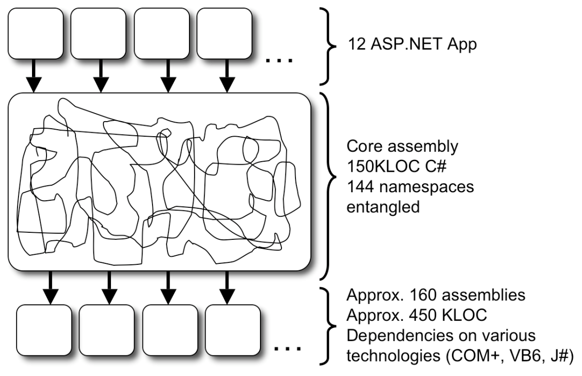

Figure 1. Overall CSoft architecture

The query language can be used for detecting code smells, and we will give a small example with the God class [19] code smell. Table 1 explains the abbreviations being used.

Table 1. Metric abbreviations

AOFD	Access Of Foreign Data
WMPC1	Weighted Methods Per Class 1
Cyclomatic Complexity	Class-level cyclomatic complexity
LCOM	Lack of Cohesion Of Methods
TCC	Tight Class Cohesion
TypeCe	Efferent coupling

A God class is a class with too many responsibilities, delegating only minor details to a set of trivial classes and using data from other classes. To detect God classes in NDepend, we adopt the detection strategy proposed by Marinescu [27] to the set of available metrics in the NDepend CQL. The original query of Marinescu is the following:

AOFD top 20% and AOFD higher 4 and WMPC1 higher 20 and TCC lower 33

We use the following query for detecting instances of god class with the NDepend CQL:

```
SELECT TOP 20 TYPES WHERE TypeCe > 25 AND
CyclomaticComplexity > 20 AND LCOM > 0.77
```

Where *TypeCe* was used instead of *AOFD*, *CyclomaticComplexity* was used instead of *WMPC1* and *LCOM* was used as the counterpart of *TCC*. As a result from this query, NDepend returned a set of business management classes, all from a module called *Reporting,* which is also the section of CSoft's *Core* with the highest defect rate and performance problems.

4.2. Development problems

We can summarize the acknowledged problems during development by distinguishing four aspects:

(a) Analyzability and comprehensibility. Due to the high complexity of the system, it is very hard for developers to get an overview of the code and its structure. Especially the central component has grown extremely large and has many internal references (each namespace depends directly or indirectly on another namespace), making it difficult to understand how it really works. This was clearly not by design, but the result of years of intense development. The system is intended to be structured as vertical modules, but as it is now there are too many relationships between the verticals – changing one will inevitable affect many others. New developers joining R&D have a steep learning curve and require close follow-up over a long period of time by more experienced developers. There exists no documentation or models that explain the structure of the system, even though this clearly would be highly useful both to existing and new developers. Even worse, having problems understanding how the code is structured leads to a fear of changing the code, both for adding new features and for improving existing code. The unclear internal structure creates a cognitive overload and a common (unfortunate) way to deal with this is code duplication: instead of modifying existing code, developers create their own copy over which they have full control. This leads to a larger cognitive overload for other developers, only making the problem worse – a self-reinforcing effect.

(b) Modifiability and deployability. As a result of the duplication and entanglement of code, developers frequently need to perform so-called shotgun surgery, meaning that even the modification of a small detail forces them to identify and change code in many places. These problems slow down the development process and the potential for errors increases due to the high chance of overlooking one or more locations. Having to deal with bad code is frustrating to the developers as they in some ways in practical terms are enforced to build bad code on bad code as there is no room to actually resolve the problem. Besides development and maintenance, also deployment of the product suffers from its structure: The current core component aggregates features and functionality for every possible configuration of the product and it has to be released as a whole, even though only a fraction of the functionality may actually be needed for a particular configuration.

(c) Testability and stability. Due to the size of the code and the many cross-references, there are too many paths through the code to test them all systematically. The test coverage is not high enough and existing tests have shown to be unstable and inconsistent. For example, the same tests run on similar systems may produce different outcomes that are hard to explain. Also, a lot of the existing tests are extremely large, meaning that they too are hard to maintain and use. When a test fails, it often takes a lot of time to locate and fix the actual problem that triggered the failure. Although such tests are supposed to act as a safety net and give developers the courage to make changes they are not trusted. This increases the fear or at least reluctance to change existing code – since the effects of a change are hard to foresee and errors can have considerable negative effects. Nevertheless, regression testing is done, albeit with a lower than desired quality.

(d) Organization and process. As both the business domain and the system are highly complex, each of the development teams (4-6 developers in each) has an expert (the so-called guru). This guru has high technical skills and extensive experience with the code, which is vital for the team to solve its tasks.

Consequently, this organization represents a considerable vulnerability; losing just a few of these gurus would have devastating effects on the development. The development process is based on two-week iterations and it is a strong focus on delivering working software by the end of each iteration. A negative effect of this focus is that delivering quality software is at times traded in for creating a working version. Each iteration ends with a review, but the high velocity typically does not give enough time to catch all issues. This causes extra work close to a release when the system is thoroughly tested as a whole, yet entropy is allowed to grow from release to release. The development teams are set up to have separate areas of concern, each team being responsible for a part of the total product, e.g., the reporting solution or the data storage. The idea is to build competence around a well-defined part. Unfortunately, the structure of the system does not reflect this organization in practice, because functionality is spread throughout the code. This forces the teams to operate outside their area of concern, which has shown to negatively affect their ability to produce enough new and improved features of the product in their releases. The total request for improvements from the market is constantly higher than what actually is delivered, thus indicating a need to improve development efficiency.

4.3. Ideas for improving

As part of the discussions with the architects, we also collected several of their high-level ideas to further improve the product and development process:

(a) Process automation. Currently too much testing is done manually and more automation is desired. In addition, to establish an efficient and trustworthy safety net for the developers, tests need to become more stable and trustworthy. With this in place, the architects can introduce what they call “pain-driven development”. That is, when a developer introduces or changes code that breaks the tests, he or she will get notified immediately to correct it.

(b) Restructuring and refactoring goals. The architects feel that components of the software need to be de-coupled from the core and the overlapping and duplicated code has to be removed. They also agreed that the system should have a clearer separation of concerns were vertical modularization should reflect business segments and horizontally, the system should better separate business and platform related code.

(c) Continuous monitoring of quality. The architects proposed a principle that they refer to as “quality-from-now”, meaning that any change to the code should be analyzed at development time, to check that it does not conflict with defined rules of good design. This can, for example, be achieved using a tool like NDepend, by defining CQL rules to detect code smells and monitor potential problems nearly constantly during development. The architects believe that this approach would considerably reduce the fear of changing the code.

5. Discussion

In this section, we discuss each of the problem areas identified in the case study. We analyze their implications in the agile process and propose potential solutions. We conclude with some avenues for future research that follow from our literature review.

5.1. Analyzability and comprehensibility

In [28], van Deursen analyses the effects of various agile practices on program comprehension, concluding that pair programming, unit testing and refactoring are the practices that support comprehension. We observe that most agile methods assume that development starts from scratch and ends with a release – post-release maintenance is not covered. In our case, the system was already very complex when Evo was adopted, and although agile methods promote communication over documentation, the lack of adequate documentation holds back the comprehension of such a system. Although XP states that “the code is the documentation”, there is no guarantee that the code can serve this purpose if the system was originally developed using different methods, in this case, a changing mix of approaches from ad-hoc via waterfall to agile. The limited number of ‘experts’ of the system, the high number of new coming developers, and urgent demands on new functionality, makes pair programming a not very practical solution for spreading knowledge. *Visualization tools* could help new developers to understand the code while refactoring, and additionally generate adequate models and documentation for the system, but the challenge here is to understand which visualizations are better for which purposes.

In addition to visualisation tools, the application of refactorings to untangle *crosscutting concerns* will improve the comprehensibility [29]. Such migrations will better distinguish the code for various business segments and separate business and platform related code, allowing newcomers to explore the code more intuitively Semi-automated tools for this kind of refactoring has recently become available [30, 31].

Finally, to overcome “the fear of change” and cope with the time pressure, we suggest *semi-automatic code inspections* (cf. [32]), potentially extended with advanced visualization and analysis tools such as [33-35]. Tools will help developers to get a better understanding of non-trivial refactorings, and can even automate the more trivial ones. Dedicated tools have been developed to eliminate *code clones* [36], although the negative effects of *code clones* are still being investigated [25].

5.2. Modifiability and deployability

According to Martin [37], dependency problems, like the ones observed in our case study, largely relate to two design smells: rigidity and immobility. Rigidity means that a change in the system implies a cascade of changes in other modules. Immobility refers to the inability of the system to encapsulate components that can be reused, because it implies too much effort or risk. If the smells are all over the system, high-level restructuring is needed to get rid of unwanted dependencies. One immediate consequence of these dependency issues is the violation of the Interface Segregation Principle [37], explaining most of the difficulties in the deployment stage. The analysis of module dependency [38, 39] could represent a feasible strategy for “leveling the code”. In [40], Bourqun and Keller proposed the analysis of code smells alongside with architectural violations for achieving high-impact refactorings, and presented a comprehensive case study where they describe how they combined several tools and techniques, the resulting architecture and the refactoring process. Our findings lead us to believe that an approach along the lines of this work will be very beneficial to improve modifiability and deployability in the context of our case study.

5.3. Testability and stability

Unit testing is one of the important components of agile methods. In the context of our case, the considerable code size combined with a large amount of dependencies in the code makes it hard to define unit tests and achieve high levels of coverage. Due to the high pace of development, there is little room for regression, integration and system testing during the iterations and CSoft relies on the feedback from external stakeholders as quality checks. Recent work has focused on methods and techniques for improving unit test suits [41-43], alongside with empirical studies on defects prediction [23] that aid planning. However, there are still various challenges to agile testing that go beyond unit testing that are not completely understood [44, 45]. Although we consider visualization and analysis tools to be useful, we know that non-trivial refactorings are risky and time consuming due to the unstable characteristic of the system. The current lack of understanding of the effects of given code smells and refactorings makes this task very challenging [21]. The usage of multiple criteria and goal-centered indicators could be a feasible solution for focusing on the relevant aspects within a project (see [46, 47]).

5.4. Organization and process

The strong focus on rapid and continuous delivery of features at CSoft has lead to the construction of teams with defined areas of concern. In the same spirit of “inspect-and-adapt” from Scrum, CSoft has deviated slightly from certain agile practices in order to adapt agile practices to their context. As mentioned before in the Analyzability and comprehensibility section, when the system and organization become too complex, the use of practices such as pair programming and team rotation seems not to provide the same advantages as in small teams. We also conjecture that an important reason for delays on the incorporation of new features is due to the system not reflecting the same separation of concerns as the development tasks. This entanglement of crosscutting concerns is a common problem with software maintenance. Refactoring towards an aspect-oriented version could help to restructure the existing code according to the areas of concerns [29], but this area is relatively new and tools have only recently been presented [30, 31]. The lack of adequate information to perform the planning could be another reason for delays. Planning of iterations could be enhanced by considering additional information, such as complexity analysis of the tasks to improve on estimates obtained from planning poker. However, such complexity analysis may still have limited effect in practice, as there is not enough empirical evidence on the impact of different refactorings [21]. These uncertainties could be compensated by continuous quality monitoring, for example by combining evolution monitoring [48-51] and semi-automatic code inspections [32] to analyze metric-based characterizations and code smells of the system, e.g. using a tool like NDepend. Such a combination could be incorporated to the development flow to detect problematic areas and decide upon refactoring strategies. There still is the challenge on deciding on the prioritization of refactorings. One prioritization approach could be to use detectors of defect or performance issues in the system [52, 53].

5.5. Initial improvement actions

In the interview, discussing improvement actions with the system architects, two types are relevant. First, on a short term basis the situation must be improved to actually release the potential in the agile process. This includes both a restructuring of the system by breaking it up, removing dependencies, to make areas of concerns possible as well as automation of costly manual activities, typically testing. This requires a major effort by the whole development organization and in the case studied here it has been decided that approximately 80% of all effort in a whole release period (approximately one calendar year) will be dedicated to plan and execute a massive restructuring. Tool support, like the use of NDepend in this case, will anyhow be invaluable but it is clear that this will require a massive amount of manual work. However, this is a one-time effort and investment for future development – an expensive yet inevitable treatment. Secondly, and on a long-term regular basis the development process and the evolving software system must be continuously monitored and controlled to prevent entropy. Here a tool for evaluating continuous system change according to a set of rules or guidelines will be invaluable. However, there is still a challenge in defining these rules and defining proper responses to violations.

5.6. Avenues for future research

From the identified literature, we see that most of the work reported on methodological aspects and tools are on the development stage. More relevant case studies and better evaluations of the available tools are needed, especially studies following development over time, evaluating actions taken and their potential effects. This could permit practitioners to evaluate the different solutions and adopt the most appropriate ones to their context. In that sense, the use of evaluation frameworks like the one suggested by Maletic et al. [54] could be useful. Mealy et al. [55] have also suggested a set of usability requirements for refactoring tools. We have seen many examples of tools supporting Java, but we have scarcely seen tools supporting other widely used platforms or languages such as C#, C++ and others. Consequently, more research on integrated, cross-platform or cross-compiler, frameworks is of interest. One of our major findings is that there is relatively little empirical evidence and methods available that supports refactoring decision making. Code smells themselves are suggestions for refactoring, but when we analyze code smells, we also need additional information to drive refactoring in a cost-effective way. Detection focuses on answering: "where are the code smells?" and analysis should focus on answering "which code smells should we refactor?" or "which refactorings should we apply for this code smell?" Moreover, knowledge and methods for assessing the cost-benefits of different refactoring are still largely an open area for research [21].

6. Concluding remarks

In this paper, we have presented some of the problems agile practitioners face when dealing with software entropy in the long-term evolution of a software product line in general. We have also emphasized how the agile development process affects, and are being affect by, the system entropy. We have consulted relevant literature addressing the problem of system entropy and code smell analysis and refactoring as a viable solution.

Through our case study we found that, to keep agile responsiveness in the presence of entropy, the agile workflow needs better support for understanding, planning and evaluating the impact of changes. We have proposed a combination of two strategies to address this issue: (1) Short term: progressive high-level restructuring by untangling crosscutting concerns, which will help to improve comprehensibility, modifiability, testability and deployability of the system – all important enablers for efficient agile development. And (2) Long term: semi-automatic quality monitoring and improvement during development, which will ensure that the above qualities are kept and make the development process more predictable and thereby easier to plan. Based on our findings and discussion we revisit our two research questions:

1) How may system entropy and agile processes mutually negatively affect each other?

We have exemplified and discussed several cases of system entropy negatively affecting the ability to act agile in the development process, e.g. how complexity hampers productivity and quality. Also, we have seen cases where the velocity of the agile process with short iterations does not give time to resolve problems. One possible solution would be to add time to solve issues; however stretching iteration length is not desirable.

2) Can code smell analysis and refactoring be a viable solution?

Based on our study of relevant literature and previous studies as well as the ideas coming from the case company itself along with the discussions of our findings it seems that code smell analysis and refactoring may help to resolve the problem of entropy on a short term and also to establish a manageable structure suitable for the speed and flexibility of the agile process.

Finally, based on identified literature, we pointed out two promising research directions were solutions are currently lacking, and where practitioners need advice: (1) refactoring decision support, and (2) task complexity analysis.

As part of our own future work, we will continue the study at CSoft by evaluating the effects of the extensive ongoing refactoring project.

Acknowledgments

Our thanks to the architecture team at CSoft and to the other staff members that have been involved in this study. We would also like to thank Patrick Smacchia for his valuable insight and ideas. This work was partly funded by the Research Council of Norway under grant # 179851/I40.

7. References

[1] Rajlich, V., *Changing the paradigm of software engineering*, in *Communications of the ACM*. 2006. p. 67-70.

[2] Fægri, T.E. and G.K. Hanssen, *Collaboration and process fragility in evolutionarily product development.* IEEE Software, 2007. **24**(3): p. 96-104.

[3] Hanssen, G.K. and T.E. Fægri, *Process Fusion - Agile Product Line Engineering: an Industrial Case Study.* Journal of Systems and Software, 2008. **81**: p. 843-854.

[4] Seaman, C.B., *Qualitative methods in empirical studies in software engineering.* IEEE Transactions on Software Engineering, 1999. **25**(4): p. 557-572.

[5] Hanssen, G.K., et al., *Maintenance and agile development: challenges, opportunities and future directions*, in *proceedings of International Conference on Software Maintenance*. 2009, IEEE Press.: Edmonton, Canada. p. 487-490.

[6] Gilb, T., *Competitive Engineering: A handbook for systems engineering, requirements engineering, and software engineering using Planguage*. 2005: Elsevier.

[7] Schwaber, K., Beedle, M., *Agile Software Development with Scrum*. 2001: Prentice Hall.

[8] Hanssen, G.K. and T.E. Fægri. *Agile Customer Engagement: a Longitudinal Qualitative Case Study*. in *International Symposium on Empirical Software Engineering (ISESE)*. 2006. Rio de Janeiro, Brazil.

[9] Smaccia, P. *Getting rid of spaghetti code in the real-world: a Case Study*. 2008 [cited; Available from: http://codebetter.com/blogs/patricksmacchia/archive/2008/09/23/getting-rid-of-spaghetti-code-in-the-real-world.aspx.

[10]Belady, L.A. and M.M. Lehman, *A model of large program development.* IBM Systems Journal, 1976(3): p. 225-252.

[11]Eick, S.G., et al., *Does Code Decay? Assessing the Evidence from Change Management Data.* Transactions on Software Engineering, 2001. **27**(1): p. 1-12.

[12]Dybå, T. and T. Dingsøyr, *Empirical Studies of Agile Software Development: A Systematic Review.* Information and Software Technology 2008. **50**(9-10): p. 833-859.

[13]Fowler, M., *Is Design Dead?*, in *Extreme Programming Explained*, G. Succi and M. Marchesi, Editors. 2001, Addison Wesley Longman: Reading, Mass.

[14]Pizka, M., *Straightening spaghetti-code with refactoring?*, in *Proceedings Of The International Conference On Software Engineering Research And Practice*. 2004. p. 846-852.

[15]Neill, C.J. and P.A. Laplante, *Paying Down Design Debt with Strategic Refactoring*, in *IEEE Computer*. 2006, IEEE Computer Society. p. 113-116.

[16]Mäntylä, M. and C. Lassenius, *Subjective evaluation of software evolvability using code smells: An empirical study.* Empirical Software Engineering, 2006. **11**(3): p. 36.

[17]Mäntylä, M., *An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement*, in *Intl Symp. on Empirical Softw. Eng. (ISESE)*. 2005, IEEE.

[18]Counsell, S., et al., *Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS*, in *proceedings of International Symposium on Empirical Software Engineering*. 2006, ACM: Rio de Janeiro. p. 288-296.

[19]Fowler, M., et al., *Refactoring: Improving the Design of Existing Code.* 2000: Addison-Wesley.

[20]Counsell, S., *Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?*, in *proceedings of Second International Conference on Research Challenges in Information Science*. 2008. p. 111-122.

[21]Counsell, S., et al., *The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph*, in *proceedings of Testing: Academic and Industrial Conference - Practice And Research Techniques*. 2006. p. 181-192.

[22]Bakota, T., et al., *Towards Portable Metrics-based Models for Software Maintenance Problems*, in *proceedings of IEEE International Conference on Software Maintenance*. 2006. p. 483-486.

[23]Li, W. and R. Shatnawi, *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution.* JSS, 2006. **80**(7): p. 1120-1128.

[24]Aversano, L., L. Cerulo, and M. Di Penta, *How Clones are Maintained: An Empirical Study*, in *proceedings of European Conference on Software Maintenace and Reengineering*. 2007. p. 81-90.

[25]Lozano, A. and M. Wermelinger, *Assessing the effect of clones of changebility*, in *proceedings of IEEE International Conference of Software Maintenence*. 2008, IEEE. p. 227-236.

[26]Smaccia, P. *Code Query Language*. 2009 [cited; Available from: http://www.ndepend.com/Features.aspx#CQL.

[27]Marinescu, R., *Measurement and quality in object-oriented design*, in *Intl Conf. on Softw. Maintenance (ICSM)*. 2005, IEEE. p. 701-704.

[28]van Deursen, A., *Program comprehension risks and opportunities in extreme programming*, in *Working Conf. on Reverse Eng. (WCRE)*. 2001, IEEE. p. 176-185.

[29]Moonen, L., *Dealing with Crosscutting Concerns in Existing Software*, in *Intl Conf. on Softw. Maintenance - Frontiers of Softw. Maintenance (ICSM/FoSM 2008)*. 2008, IEEE. p. 68-77.

[30]Binkley, D., et al., *Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects.* IEEE

Transactions on Software Engineering, 2006. **32**(9): p. 698-717.

[31]Marin, M., et al., *An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw*. Automated Software Engineering, 2009. **16**(2): p. 323-356.

[32]van Emden, E. and L. Moonen, *Java quality assurance by detecting code smells*, in *Working Conf. on Reverse Eng. (WCRE)*. 2002. p. 97-106.

[33]Parnin, C., C. Görg, and O. Nnadi, *A catalogue of lightweight visualizations to support code smell inspection*, in *ACM Symposium on Software Visuallization*. 2008: Ammersee, Germany. p. 77-86.

[34]Trifu, A. and U. Reupke, *Towards Automated Restructuring of Object Oriented Systems*, in *European Conference on Software Maintenance and Reengineering*. 2007. p. 39-48.

[35]Van den Brand, M.G., et al., *Using The Meta-Environment for Maintenance and Renovation*, in *proceedings of European Conference on Software Maintenance and Reengineering*. 2007. p. 331-332.

[36]Nasehi, S.M., G.R. Sotudeh, and M. Gomrokchi, *Source code enhancement using reduction of duplicated code*, in *Conference on IASTED international Multi-Conference: Software Engineering* 2007, ACTA Press. p. 192-197.

[37]Martin, R.C., *Agile Software Development, Principles, Patterns and Practice*. 2002: Prentice Hall.

[38]Arevalo, G., S. Ducasse, and O. Nierstrasz, *Discovering Unanticipated Dependency Schemas in Class Hierarchies*, in *proceedings of Conference on Software Maintenance and Reengineering*. 2005. p. 62-71.

[39]Leitch, R. and E. Stroulia, *Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis*, in *proceedings of International Symposium on Software Metrics*. 2003. p. 309-322.

[40]Bourqun, F. and R.K. Keller, *High-impact Refactoring Based on Architecture Violations*, in *Conf. on Softw. Maintenance and Reengineering (CSMR)*. 2007. p. 149-158.

[41]Guerra, E.M. and C.T. Fernandes, *Refactoring Test Code Safely*, in *proceedings of International Conference on Software Engineering Advances*. 2007. p. 44-50.

[42]van Deursen, A., et al., *Refactoring test code*, in *eXtreme Programming Perspectives*, M. Marchesi, et al., Editors. 2002, Addison-Wesley: Reading, Massachusetts.

[43]Van Rompaey, B., et al., *On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test.* IEEE Transactions on Software Engineering, 2007. **33**(12): p. 800-817.

[44]Pettichord, B., *Agile Testing Challenges*, in *proceedings of Pacific Northwest Software Quality Conference*. 2004. p. 481-517.

[45]Talby, D., *Agile Software Testing in a Large-Scale Project.* IEEE Software, 2006. **23**(4): p. 30-37.

[46]Mäntylä, M., *Developing New Approaches for Software Design Quality Improvement Based on Subjective Evaluations*, in *proceedings of International Conference on Software Engineering*. 2004. p. 48-50.

[47]Walter, B. and B. Pietrzak, *Multi-criteria Detection of Bad Smells in Code with UTA Method*, in *proceedings of extreme programming and agile processes in software engineering (XP)*. 2005. p. 154-161.

[48]D'Ambros, M., *Supporting software evolution analysis with historical dependencies and defect information*, in *International Conference on Software Maintenance*. 2008. p. 412-415.

[49]Jermakovics, A., M. Scotto, and G. Succi, *Visual identification of software evolution patterns*, in *International Workshop on Principles of Software Evolution: in Conjunction with the 6th ESEC/FSE Joint Meeting*. 2007, ACM: Dubrovnik, Croatia. p. 27-30.

[50]Kiefer, C., A. Bernstein, and J. Tappolet, *Mining Software Repositories with iSPAROL and a Software Evolution Ontology*, in *proceedings of International Workshop on Mining Software Repositories*. 2007. p. 10-18.

[51]Xing, Z., *Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software.* IEEE Transactions on Software Engineering, 2005. **31**(10): p. 850-868.

[52]Chaabane, R., *Poor Performing Patterns of Code: Analysis and Detection*, in *proceedings of International Conference on Software Maintenance*. 2007. p. 501-502.

[53]Wasylkowski, A., A. Zeller, and C. Lindig, *Detecting object usage anomalies*, in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2007, ACM: Dubrovnik, Croatia. p. 35-44.

[54]Maletic, J.I., A. Marcus, and M.L. Collard, *A Task Oriented View of Software Visualization*, in *proceedings of International Workshop on Visualizing Software For Understanding and Analysis*. 2002. p. 32-40.

[55]Mealy, E., *Improving Usability of Software Refactoring Tools*, in *Australian Softw. Eng. Conf. (ASEC)*. 2007. p. 307-318.