Project Report
How should agile practices in large industrial systems be improved to avoid software entropy during maintenance?

Empirical methods and evidence-based decisions in software development - INF5500

Aiko Yamashita
Department of Informatics, University of Oslo

simula . research laboratory]

Table of contents

I. Introduction

Recently more and more software product organizations have adopted principles of agile software development in an attempt to improve efficiency of development and responsiveness to the market [1]. Fundamental principles of agile development are a high degree of change responsiveness, fast delivery and customer-driven development.

However, the high-pace of agile development imposes demands on the code base and the developers, potentially increasing the level of complexity and disorder in the code base (this phenomenon is known as *software entropy*, where repeated changes gradually degrade the structure of the system, making it hard to understand and maintain). On the other side, the ability to rapidly produce new increments to a product may get hampered by an unmanageable (in terms of size and complexity) code base.

One practice suggested for managing software entropy is refactoring [2]. In software engineering, "refactoring" source code means to modify its internal structure without modifying its external functional behaviour or existing functionality. The process could be informally referred to as "cleaning up" or "taking out the garbage." Refactoring could also be catalogued, as “preventive maintenance task” and it should not be confused by other maintenance tasks such as corrective or perfective tasks. As refactoring is an operation that produces revised programs from originals, it is a special case of program transformation.

Recently, the use of code smells has been introduced for identifying code segments that may need refactoring. The term code smells, which was coined by Kent Beck and Martin Fowler [2], is informally defined as bad or inconsistent parts of the design of object-oriented software refactoring. Some examples are: *Duplicate code* (identical or very similar code exists in more than one location), *Large method* (a method, function, or procedure that has grown too large), *Large class* (a class that has grown too large, see God object), and *Feature envy* (a class that uses methods of another class excessively).

Despite the common usage of refactoring and the growing popularity of code-smell based analysis, we lack a compendium of available tools, methods and empirical knowledge for detecting code smells and making trade off when implementing refactoring. This holds back agile practitioners’ possibilities for achieving more efficient and strategic refactoring decisions that could support the agile working flow.

The research question addressed within this report is: How should agile practices for software product development in large industrial systems be improved (focusing primarily in the usage of code smells and refactoring) to avoid software entropy during maintenance? and how could these improvements be incorporated in the particular case of an agile method called EVO?

In order to investigate potential improvements to agile for handling software entropy, we need to understand both the current problems and the available solutions. Consequently, this work has two-fold research questions: (a) which are the most relevant challenges and difficulties perceived by software architects in agile projects related to software entropy? And (b) which is the state-of-art (i.e., methods, tools and knowledge) for supporting code smells detection/analysis, and making trade offs when prioritizing refactoring? To address the first question we conducted a case study where we studied a medium sized software organization that develops and maintains a highly complex software product, by using evolutionary software development, also known as the EVO method [3]. To address the second question, we have conducted a literature review summarizing the current work on code smells detection and analysis, and on refactoring support.

Based on the results achieved by both the case study and the literature review, we intend to analyze he strengths and limitations of the state-of-art on code smells detection and analysis and refactoring decision, with respect to the currents challenges faced by the practitioners, aiming ultimately to suggest potential improvements on the AGILE methodology within the EVO context and suggest a study in order to evaluate those improvements within the aforementioned context.

The remainder of this report is organized as follows. Section 2 presents the methodology followed in order to answer the two research questions. Section 3 presents the findings from the studies conducted. Section 4 discusses the results and derives potential improvements in the context of inquiry. Section 5 analyses the validity of the methodology and sources from which the improvement suggestions were drawn. Section 6 presents the synthesis of the report, finalizing with Section 7, which will present a description of a study in order to evaluate the improvement suggestions.

II. Methodology

This section describes the procedures followed for conducting the case study and the literature review, respectively aimed at answering the research questions: (a) which are the most relevant challenges and difficulties perceived by software architects in agile projects related to software entropy? and (b) which is the state-of-art (i.e., methods, tools and knowledge) for supporting code smells detection/analysis, in order to make trade offs when deciding upon refactorings?

A. *The Case Study*

In this explorative case study, we used a real industry case to discover and explore issues related to software entropy. The company we interviewed will be called CSoft (an anonym). CSoft is a medium-sized Norwegian software company that develops, maintains and markets a product (with the same name) that is used for market and customer surveys. CSoft is a product in the high-end segment of the market and has a wide customer base that includes some of the world's largest market research agencies. The company was established in 1996 and has since grown steadily and has today about 260 employees including 60+ developers. The main office is located in Norway with offices also in UK, USA and Russia. CSoft can be defined as a single product, but there are many ways to use it, as it is highly modular. It has five main modules (with numerous sub-modules) e.g. to plan and design surveys, setting up panels, a central survey engine that executes the actual surveys, reporting, and data transfer to feed the database for analysis. The use of these modules varies according to the customer case. CSoft operates 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 [3] under the guidance of Tom Gilb, which originally defined the process [4]. Evo is an agile method comparable to the better-known Scrum-method [5], although the terminology differs. At CSoft, work is done in two-week iterations (equivalent to the 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 [6, 7]. Looking at the Agile Manifesto[1], Evo – as it is adopted at CSoft – conforms to the four basic values; interaction is highly valued, they have a strong emphasis on delivering working software after every iteration, invited lead users participate in development and finally, development is open to change in requirements and design.

We have collected data in two ways; first the company had a workshop with Patrick Smacchia, an external consultant that analyzed the code-base of their software product using a tool called NDepend™. Some metrics were identified describing the component and layering structure of the system, the number of .Net namespaces, internal references etc. The big picture showed a system based on one big and extremely complex component or .Net assembly [8]. Secondly, we have collected data through an in-depth group interview with two architects from the four-person architecture team. This group has two main responsibilities, firstly to improve the supporting infrastructure for development (testing, code management, builds etc.) and secondly to improve the architecture of the system, that is, to change the system to make it easier to add and improve features and to ease the deployment of the product. The interview, lasting for 3,5 hours was recorded and transcribed. Further on, the transcription (30 pages of text) was initially analyzed using NVivo™(a tool for tagging fragments of text with information about context, meaning etc) in order to follow an approach on the line of grounded theory [9].

B. *Literature review on code smells and refactoring*

In order to answer our second question, we performed a literature review, following the guidelines suggested by Brereton et al. [10]. Our intention was to explore which tools, methods or knowledge are currently available for code smells analysis and refactoring decisions. We defined the protocol in three stages: Stage 1 was source selection and querying, stage 2 consisted of filtering the result and stage 3 consisted of data extraction and synthesis.

For stage 1, we decided upon the sources and defined the search query based on our review objective. The terms used in the search were: *Code smell*, *Refactoring*, *Method*, *Tool*, *Technique*, *Decision*, *Analysis* and *Maintenance*. The sources used for extracting the primary studies were IEEE Xplore, The ACM Digital Library and ISI Web of knowledge, limited to publications within 2000-2008. For ISI, the following query was used for the search:

```
Topic=(code smell) OR Topic=(refactoring) AND Publication Name=(Maint*) Timespan=2000-2008. Databases=SCI-EXPANDED,
CPCI-S, CPCI-SSH.
```

[1] www.agilemanifesto.org

The query used for IEEE Xplore is presented below:

```
(code <and> smell) <and> (tool <or> method <or> technique) <and> (software <and> ((refactoring <or> maintenance)
<and> (decision <or> analysis))) <in> metadata
```

For ACM, we used several combinations of the selected key words and aggregated the results. The following are the combinations used for performing the search in ACM:

```
"code smell" and "software" and "decision" and "refactoring"
"code smell " and "software" and "analysis" and "refactoring"
"code smell" and "software" and "analysis" and "maintenance"
```

The synonymous used for the term *Code smell* are the following: *Bad smell*, *Structural symptom* and *Smell*. We also looked into the results from previous literature reviews in the area of agile methods such as [11-14] and extracted the references we found relevant for our review objective and added them into the list of sources.

For stage 2, the relevant citations from stage 1 (n = 115) were entered to a spreadsheet and duplicated entries were eliminated, resulting in 80 citations. The following exclusion criteria was used in order to eliminate the immediate citations that were not relevant to our review:

1) A report of a tool/method or concept that was not related to the use of OO programming languages
2) A position paper, an editorial, preface, discussion, article summary, or summary of tutorials, workshops, panels, and poster sessions
3) It was not related to software engineering (e.g., biology)

This resulted on 69 citations, from which titles and abstracts were examined in order to determine the type of contribution and if they met the criterion described by our review objective. The content was also revised in case the title or abstracts were not clear enough. The source was included only if:

1) It reported a method/tool which could be used for code smell analysis/detection or
2) It reported a method/tool which could be used for refactoring/redesign decisions or
3) It reported results from an empirical study useful for code smell analysis/detection or refactoring decisions.

The result was a list of 51 relevant citations for our review. On stage 3, we conducted the data extraction (by using a basic extraction form), by revising the actual studies and by classifying their contribution into the following categories: results from *design research* perspective (*Method*s and *Tools)* and results from *empirical studies*. The methodological contributions were then categorized according to their purpose: *Detection*, *Analysis*, or *Visualization* of Code Smells. Subsequently, we conducted a synthesis of the most relevant aspects of each of the classified groups.

III. Findings

A. Findings from the Case Study

This section presents an overview of the collected data (the NDepend workshop and the group interview) and describes the structure and the complexity issue of the software product being developed by CSoft and the problems this structure imposes.

The system, which has been under constant development for about fourteen years, is based on several technologies that have emerged over the years. Aging solutions from years ago are still part of the system, such as older ASP solutions, COM+ components, VB6 code and other *old* technologies. Today most new code is being developed in C# which by now is spread across approximately 160 .Net assemblies. The total solution is best described as a traditional three-tier system; MS SQL Server in the data layer, a business layer and a presentation layer based on a dozen ASP.Net applications. The separation between the presentation- and the business layer is clean, however the most obvious problem or pattern in the software structure is what the architects refer to as *the Blob*. This is a very large assembly (named *Core*) consisting of approximately 150 K lines of code in 144 namespaces.

The NDepend tool were used to visualize and generate descriptive code metrics of the internal structure of this assembly using what's called a Dependency Structure Matrix and showed an extremely entangled structure where most namespaces refers to most namespaces thus creating lot a of cyclic dependencies.

In the following subsections, we describe the problem areas identified according to the following four aspects: *Analyzability and Learnability, Changeability and Deployability, Testability and Stability* and *Organization and*

Process. These aspects emerged from the coding done on the interview transcripts, which consequently were sorted following the grounded theory [9] approach. The coding and sorting will not be described but the main problems and situations that were associated with the emergent concepts that relate to software entropy and software maintenance.

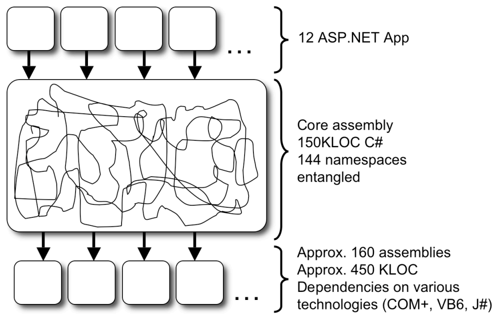

Figure 1: CSoft system's diagrammatic view according to the architects

Analyzability and Learnability. Due to the high complexity of the system, and especially the central core component, it is extremely hard for developers to get an overview of the code and the structure. First of all, the core component is extremely large with a lot of references, making it hard to understand how it really works. It was never designed to be like this, but is a product of years of intense development. New developers joining R&D have a steep learning curve and requires 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. 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 chaotic structure creates what can be called a semantic overload for the developers. One common (unfortunate) way to deal with this is to duplicate code - instead of changing existing, working code, the developers rather separately develops a new piece of code which he or she then has full control over. This off course only makes the problem worse – a self-reinforcing effect.

Changeability and Deployability. As an effect of code duplication, developers does whet the architects refers to as shotgun surgery, meaning that developers that are going to change only small details, e.g. a single line of code, are forced to identify and alter code several other places. Due to these problems, development takes more time compared to an easy-to-follow structure. Besides development and maintenance, deployment of the product also suffers from the excessive complexity. The core component contains features and functionality that is necessary to all configurations of the product and has to be released as a whole even though only a fraction of the functionality is actually needed.

Testability and stability. Due to the size of the code and the many references, there are extremely 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, similar tests run on similar systems may produce initially unexplainable different results. Also, a lot of the existing tests are extremely large, meaning that they are hard to maintain. When tests fail, it often takes a lot of time to fix the identified problem. In sum, the safety net which such tests are supposed to be, is in practice conceived to be non-trustable which leads to a fear or at least reluctance to change existing code – since effects of change are hard to foresee and consequences of errors potentially bad. Yet, regression testing is done, but with a lower than desired quality.

Organization and process. As both the business domain and the system are highly complex, each of the development teams (4-6 developers) has an expert, which is referred to as the guru. This is a person with high technical skills and extensive experience with the code. He or she is vital for the team to solve its tasks. Consequently, this represents a great vulnerability. Loosing just a few of these gurus would have devastating effects on the performance in development. The development process is based on two weeks iterations where the teams are extremely focused on delivering working software by the end of each iteration. However, as the focus is so strong on constantly delivering working software it often happens that the quality of the software suffers, which causes extra work close to the release when the system is thoroughly tested as a whole. In the iterations there is a review at the end, but the high velocity of the process does not give enough time to catch all issues. The development teams are set up to have separate areas of concern where each team is responsible of a part of the total product, for example the reporting solution or the data storage. The idea is to build competence around a well-defined part but the structure of the system does not reflect this as functionality in practice is spread throughout the code. This forces the teams to move outside their area of concern. In sum, these problems have shown to negatively affect the development teams' 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 for improved efficiency.

B. Findings from the Literature Review

Many contributions had a clear focus on methodological aspects but some others additionally presented software implementations of their strategies. Some of the contributions belonged to a more specific domain, such as aspect-oriented software (3 citations), refactoring of test suits (5 citations) and analysis of code smells for defects/performance prediction (3 citations).

From Table 1, we can see that 40 (79%) of the sources consisted of methodological contributions, and only 11 (21%) of the publication reports on empirical studies. This suggests that decision-making support might be limited due to the reduced knowledge in this area. Regarding to the purpose of work, we found that the number of contributions on Analysis (16 citations = 40%) and Detection (15 citations = 37.5%) are quite similar. But considering the fact that most Visualization contributions (9 citations = 22.5%) are also meant to support Analysis, we could conjecture that considerable part of the focus has been on *Analysis* or *Diagnosis* of code smells and refactoring solutions. Due to the space limitations of this report, we will not present a complete summary of the primary set, but concentrate on the sources relevant to the context of the case study. We will present a summary of all the empirical studies (11 citations). Subsequently, we will present a summary of the most relevant methodological contributions.

Table 1: Classification according to type of contribution

Type of research	Contribution	Number	Percent
Design research	Method	22	43
	Tool	10	20
	Method + Tool	8	16
	Subtotal	***40***	***79***
Empirical research	Report of findings	11	21
Total		51	100

From the nine papers with actual empirical studies, we are able to present a summary of findings. Together we have found that the papers basically cover four topics: (a) subjective evaluation of code smells, (b) code refactoring decisions, (c) generalizability of studies based on code metrics and (d) code clones.

Table 2: List of empirical studies

No.	Study
[15]	An Experiment on Subjective Evolvability Evaluation of OO Software: Explaining Factors and Interrater Agreement
[16]	Subjective evaluation of software Evolvability using code smells: An empirical study
[17]	Bad smells - Humans as code critics
[18]	Refactoring test suites versus test behaviour - a TTCN-3 perspective
[19]	Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS
[20]	Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?
[21]	The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph
[22]	Towards Portable Metrics-based Models for Software Maintenance Problems
[23]	An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution
[24]	Assessing the effect of clones on changeability
[25]	How Clones are Maintained: An Empirical Study

Studies about subjective evaluation of code smells. In [16, 17] Mäntylä et al. reports on 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. They found that subjective evaluations with developers were not uniform. However, in cases with a low level of problems, the conformance was higher than cases with high a level of problems. When investigating the demographics of the evaluators they saw that experienced developers were better spotting structural problems in the code than regular developers who could spot problems mainly at the code level (e.g., duplicated code). Also, developers that had worked with the code for a long period of time tended to detect fewer smells than developers with shorter experience with the code being evaluated. 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 correlate with 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 also reports on a student experiment for evaluating subjective evaluation for code smells detection and refactoring decision [15]. 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.

Studies for code refactoring decisions. Counsell et al. investigated refactorings done in seven open-source Java systems to see which types of refactorings that were most common and which effects these had in solving code-smells [19]. The study used fifteen refactorings from the Martin Fowler and Kent Beck classification [26]. 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 (except Rename Field) 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. Refactorings inducing short chains on the other hand, were used frequently. In yet another related study Counsell at al. [21] uses the same refactorings and the same empirical data set to investigate how refactorings affect the testability of a system. That is, to what extent the fifteen refactorings affect the re-usability of a test suite – having to change or update the tests is a spin-off cost of refactoring. The main conclusion from the study is that while semantically preserving refactorings may be ideal for preserving tests sets, they are not necessarily always the right refactorings to choose.

Studies about the generalizability of studies based on code metrics. We identified only one study addressing how well code metrics can be applied across different software systems. Bakota et al. evaluated four software systems [22]: an OSS system vs. a closed source system and an office application vs. a telecommunication system. They found that these four quite different systems could be differentiated from each other pretty well, based on their metric values. However, two metrics "Response For A Class" and "Weighted Methods Per Class", behaved very differently on the analyzed systems. 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.

Studies on Code clones. Two studies addresses problems potentially related to code clones. Aversano et al. [25] studied code clones 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 somewhat to demystify the image of code clones being bad design. In the same line of inquiry, Lozano and Wermelinger [24] report the results from an experiment to investigate the effects of code clones on maintenance. They concluded that the effort seems to increase depending on the percentage of the system affected when the methods that share the clone are modified, but it was not possible to establish any significant relation between cloning and maintenance effort increase.

Overview of methodological contributions. Results from the review showed that the predominant detection methods for code smells are metrics-based, such as [27-30]. But in addition, other methodologies have started to consider using multiple criteria [31] and interrelations between code smells [32], alongside metrics-based analysis. Examples of such work include [33-35]. Furthermore, the results show that the scope for refactoring has been extended to Aspect Oriented Programming (AOP) [36], and testing [37-40]. Analysis of code smells now is also being used to handle issues not only for maintenance purposes, but also for defect predictions [41] and performance evaluations [42]. Approaches more on the side of architectural or high-level design issues can be found in [43, 44]. Table 5 in the appendix section displays the list of methodological contributions categorized by M=Method, T=Tool and B= Both, implying that the contribution is based on a methodology and a tool that implements the methodology.

IV. Discussion

In this section, we will discuss each of the identified problem areas from the findings obtained by the case study. We will discuss their implications in the agile process and suggest potential improvements based on the findings from the literature review.

A. Analyzability and Learnability

In [45], Van Deursen analyses the risks and opportunities that agile practices (specifically XP) may represent in program comprehension, indicating that pair programming, unit testing and refactoring are the most promising practices. One clear gap is that most of the rationale for agile practices assumes that development ends with a release – post-release maintenance is not covered. In our case, the system was already very complex when Evo was adopted. Although agile promotes human-communication over documentation, the lack of adequate documentation holds back the understanding/learning of complex systems. XP states: "code is the documentation", emphasizing on comments and clear code, but if the system was developed using different methods, is not guaranteed that the code will comprise a readable documentation. Also, the limited number of 'experts' in 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 support* 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. Also, the use of *cross cutting concerns* [46] could improve the learnability by "dividing conceptually" the different internal modules of the system. In this way, the newcomers will be able to navigate more intuitively across the code base.

In order to overcome "the fear of change" and cope with the time pressure, *semi-automatic code inspections* [47] comprising visualization and analysis tools such as [33-35, 44] are suggested. This is in order to get a better understanding of non-trivial refactorings, and automate the trivial ones by suitable tools. Some advances in automated refactoring have focused on the elimination of *code clones* [48], although the negative effects of *code clones* are still being investigated [24].

B. Changeability and deployability

According to Martin [49], the dependency problems described in the case study relate largely to two *design smells* called *rigidity* and *immobility,* where a change in the system implies a cascade of changes in other modules, and the inability of the system to entangle components that can be reused because it implies too much effort or risk. If the smells are all over the system, high-level restructuring in order to get rid of unwanted dependencies seems unavoidable. One immediate consequence of dependency issues is the violation of *Interface Segregation Principle* [ibid.], which explains most of the difficulties in the deployment stage. The analysis of modules dependency [50, 51] could represent a feasible strategy for "levelizing the code". In [43], 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. We believe studies in the line of this report are very useful for this context of inquiry.

C. Testability and stability

In agile methods, unit testing is yet another important practice. This enforces a 1-1 relationship between the code and the test code, which at this point of development leads to unmanageable results. To this we have to add the fact that CSoft mainly relies on the customer for external quality checks. Planning of regression, integration and system testing within the iterations are not given enough attention. Recent work has focused on providing methods and techniques for dealing with the refactoring of test suits [37-40], alongside with empirical studies on defects prediction [23].

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-centred indicators could be a feasible solution for focusing on the relevant aspects within a project. Some examples of multiple-criteria approaches can be found in [31, 52]. Another strategy to prioritize areas for refactoring could be to use detectors of defect or performance issues within the system [41, 42].

D. 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 to their context. As mentioned before in the *Analyzability* section, when the systems and organization become too complex, the use of notions as pair programming and team rotation seem 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 areas of concern as the development teams. This points back to the fact that system may have *cross cutting concerns*, which is a common problem in large systems that have been growing for a long period of time. Refactorings addressing AOP [30, 36] could help to restructure the system according to the *areas of concern,* but this area is very incipient and no straightforward solutions are yet available. We also conjecture that a potential reason for delays in the latter stages of iteration or release is due to the lack of adequate information to perform the planning. Better planning of iterations could receive additional information such as from complexity analysis of the involved tasks rather than only estimation activities, e.g., planning poker. In this way, predictors for complexity levels of refactorings and changes could support more accurate estimations by the developers. Nevertheless such predictors may still have limited scope here, since there is not enough empirical evidence on effects of code smells, nor on the impact of different refactorings [21]. Although the lack of empirical information, we could compensate these uncertainties by a continuous monitoring of quality. One strategy could be combining *evolution monitoring* [53-56] and *semi-automatic code reviews,* where we analyze metric-based characterizations and code smells of the system e.g., [44] and tools such as NDepend. Such a combination could be incorporated to the development flow to detect problematic areas and decide upon refactoring strategies. However, we still have the challenge on deciding on the prioritization of refactorings.

V. Critical evaluation of methodlogy used and sources involved

In this section, we analyze critically the methodology used in order to reflect upon the validity of the findings. Subsequently, we analyze the claims from the sources involved and the warrants on which they are based, in order to evaluate their validity. Based on the results from this analysis, we will revise the ideas described in the discussion section and provide a synthesis of the

A. Literature review

Based on the results from our literature review and the identified problems from the case study, we perceive that solutions and methods for refactoring decision and task complexity analysis are lacking. Nevertheless, we are aware that agile is a considerably new area and publications are scattered amongst different sources, so including only three digital libraries and excluding grey literature will not provide a full coverage. We also found that our review query was somewhat limiting the scope due to the use of non-standardized keywords as filters. Although code smells and refactoring catalogues are relatively new, notions related to restructuring the design are not new. Topics such as reengineering, code cloning and aspect oriented programming are highly relevant or closely related to code smells and refactoring and these topics were not included in this review.

B. Case Study

Interviews with Architects. One of the limitations of the case study is that the results from the interviews were not compared to empirical studies on the challenges faced by agile practitioners with respect to software entropy. In our case, we were more interested on investigating a phenomenon on a specific context, and explore potential theories by understanding software entropy in the EVO setting. This is due to two reasons: First, there is relatively low number of studies reporting agile challenges related to software entropy (besides grey-literature) and there was a risk that the context of the studies may not be enough specified, thus leading to external validity issues from the empirical studies. Second, we wanted to follow more of an Action Research [57] perspective since the collaboration with the company CSoft was centered on this specific schema. As such, we had to make sure that the problems identified were specific to the company, but at the same time could give the opportunity to cross-compare and identify commonalities within other agile contexts, so it could become also interesting from a research perspective. Regarding to the validity of the results, and following more of a grounded theory approach, the most important aspect of the quality of the findings is their fit, relevance and workability [58]. From the 'fit' perspective, we can say that the different consequences of software entropy were grouped consistently into four problem areas, which are of common knowledge within software engineering research community. With respect to its relevance, we could say that the problems identified were found relevant for both the practitioner's side and the research community as well (part of the results were published in a peer-reviewed conference in the topic of maintenance [59]).

Workshop with consultant. As for the workshop held with Patrick Smacchia, we could say that there is a slightly vested interest from his part towards the tool used for the analysis of the system. But in the other hand, CSoft already acquired the tool by the time the workshop was held, so this interest didn't play a significant impact. Another aspect to consider is the fact that the information used during the workshop was more focused to get an overview of the technical properties of a system, which suffers from software entropy. From that perspective, this information only can complement the conclusions drawn but do not represent a threat to the validity of the findings.

C. Validity of expert opinion

In order to identify the problems or challenges in the agile development flow related to software entropy, we relied on the results from an interview with the Software Architects of CSoft. Here we are assuming that the Software Architects are experts in their domain and that they have enough insight of the project and the product in order to provide consistent and accurate information.

Some studies within the health domain suggest that decision based on actuarial judgment (which uses mathematical models to evaluate risks, based on data collected from studies, surveys, etc) is superior to clinical judgment [60] (where critical decisions are made on the basis of scientific observations but with the added skill of the expert provided by long experience of similar cases). Nevertheless, recent studies tend to indicate that the reliability of expert judgment highly depends of the nature of the task and the domain in which the judgment is made [61]. Furthermore, a meta analysis on clinical judgment reported in [62] also indicate that even within the domain areas where the clinical judgment seems in disadvantage, there is a significant difference in the performance and accuracy between experienced clinicians and novice. Shanteau proposes the "Theory of Expert Competence", where the competence of the expert is dependent of (1) a sufficient knowledge of the domain, (2) the psychological traits associated with experts, (3) the cognitive skills necessary to make tough decisions, (4) the ability to use appropriate decision strategies, and (5) a task with suitable characteristics.

We consider in this study that that our definition for expert fits Shanteau's definition of expert: "experts are operationally defined as those who have been recognized within their profession as having the necessary skills and abilities to perform at the highest level." All the three architects work closely with their development teams (which makes them aware of the issues from different perspectives going from developer's level problems all the way up to architectural, infrastructural and organizational challenges). From a qualification viewpoint, they are considered by their peers to be best at what they do. At organizational level, they have the highest responsibility at technical level on ensuring the quality of the product.

With respect of the domain area, we consider software engineering has more static objects (e.g., the source code) or things that are relatively constant (e.g., requirements from clients, development process, software tools and methodologies), and would make judgment easier than compared to domains purely based on behavioral characteristics. We could refer back to Shanteau's tasks characteristics and say that although organizational issues involve behavioral characteristics, the mechanisms and objects involved in software development enable repeatability and opportunity for learning/calibrating the knowledge, enabling experts in this domain to have more competent performance. Due to space limitations, we will not develop further all the aspects from Shanteau's theory but state briefly the threats to validity, which are common whenever subjective opinion is used. One major threat is the fact that the interviewees all belong to the same group of experts (Software Architects), which could give biased perspectives upon the criticality of the problems and it could be a possibility that certain problems may escape to the attention of the architects. A better approach would have been to interview different members of the organization (e.g., developers, team leaders) in order to get a more complete and balanced view of the challenges entailed by software entropy in an agile organization.

D. Validity of the selected sources from the literature review

In this section we will briefly analyze the validity and usefulness of the selected sources from the literature review in order to provide a concise synthesis of the recommendations and strategies in order to improve agile practices.

Mantyla's studies [16, 17] report sources from industrial and academic environments, which might represent some external validity issues, but considering that many developers at CSoft are also novices, it is possible to consider their results valid within the CSoft context. The results within the studies looked consistent, and this strengthens the reliability of the findings. One major limitation perceived in his studies is that the definition of code smells that were detected automatically was not reported, which may lead to inconsistencies when compared to other studies using automated code smell detection. Despite this fact, their conclusion still stands (subjective and automated evaluations are not consistent in detecting complex code smells), and their findings point out that a combination of automated and subjective detection of code smell is the most appropriate. In that sense, it should be possible to use data driven from code analysis and let the developers use their experience and intuition for refactoring decisions.

Studies from Counsell [18, 19] do not report enough on the context of the projects from which the data was extracted. As a result, no specific practical advice can be drawn. At most they could provide some hints that certain refactorings are potentially more time consuming because they involve a longer refactoring chain. Moreover, without proper tooling, this information would hardly be used by the developers in a practical setting. With respect to [18], CSoft does not use TTN test suites, which also limits the applicability of the conclusions drawn from the study, although it could provide an idea of which kind of test refactoring are potentially time consuming. With respect to Aspect Oriented tools and methodologies, the lack of industrial cases reported is an indicator that the area is still immature, therefore not possible to applicable it in the current setting. This is the same case with multiple-criteria detection of code smells (reported in [31]), since they do not report a tool to implement the detection within .NET platform, which from implementation viewpoint seems to be a difficult method to use.

With respect to the code clones, although there is some evidence pointing out that they are not so harmful as commonly considered, there is no clear studies indicating which cases represent the critical situations where code clones could have really negative consequences in the productivity and quality of the code. Moreover, considering the case reported in [24] where it was analyzed that calculating the increase (how much the effort increased) between cloned and not cloned periods, 50% of the time was no increase (even minus), but once that it become positive, it would result in a rapid grow up to 900% of increase in the effort. Considering this information and the recurrent comments on code duplication referred by the architects during the case study, code clones should deserve attention, but we could not find any adequate tools in our review for addressing code clones detection in .NET environments.

With respect to study reported by Li and Shatnawi [23], the methodology reported a concise work on hypothesis test validation and the description of the study is replicable. The threads to validity were reported and the period of time that the study covered for its analysis is realistic and it involves an example of a industry relevant piece of software (Eclipse). Although this application is a Java application and CSoft works on C# .NET platform, we do not expect that the subtle differences between these languages have an outstanding effect on the applicability of the results from this study on CSoft's context since the issues addressed in these code smells are more related to purely OO principles rather than technical aspects related to virtual machines or platform differences. As such, they could represent good indicators to be used when planning for testing or refactoring.

Although we couldn't find any visualization tools for .NET environment, we still consider that the examples given in the sources presented are representative of a lightweight option for enhancing high-level understanding of the system, and potentially could be expanded with annotation mechanisms as found in [63]. Nevertheless, they might need some technical adjustments before being useful for CSoft's context. The identification of cross-cutting concerns was also recognized as a potential area for research and potentially useful from the challenges identified in the case study, but more information is required in order to evaluate the different alternatives.

Most of the methodological and tool contributions were still in their development stage. More relevant case studies and better evaluations of the available tools are needed, so practitioners can evaluate the different solutions and adopt the most appropriate ones to their context. Mealy et al. [64] have suggested a set of usability requirements for refactoring tools. In addition, evaluation frameworks like the one suggested by Maletic et al. [65] are needed to assure comparable results. Although the analysis of modules dependencies, high-impact refactorings and evolution monitoring are topics which seem to address the problems identified within the case study, as mentioned before, the reports on cross-evaluation of techniques and their suitability in different contexts is non existent.

VI. SYNTHESIS OF FINDINGS

In this report, we have presented some of the problems agile practitioners face when dealing with big scale projects and software entropy:

1. Difficulties getting the overview of system
2. Steep learning curve for newcomers
3. Unmanageable increase of duplicated code
4. Difficult releases due to complex dependencies between the components
5. Unforeseeable effects from changes in the code
6. Defect corrections demand too much effort
7. Lack of test planning does not enable the handling of issues
8. Knowledge of the system is not evenly distributed (the guru instability)
9. The areas of concern not reflected in the design/implementation of the system

Through the case study, we found that better support is needed for learning, planning and testing activities within the agile flow in order to keep the *agile responsiveness*. We have suggested some basic working strategies for this context and we will describe a study to observe the impact of strategies 3, 5 and 6 initially.

1. Identification of cross cutting concerns
2. High-impact refactoring
3. Semi-automatic code inspections
4. Analysis of modules dependency
5. Use of indicators for identifying error prone modules
6. Evolution monitoring of the system

VII. Design of a study

One of our major findings in this work is that relatively little empirical evidence is available for 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 refactoring should we apply for this code smell?" Knowledge and methods for assessing the cost-benefits of different refactoring is still under-researched. In the following chapter, we describe a study where we could apply some of the empirical evidence and verify their usefulness and accuracy throughout a period of time sufficiently long, which could enable us to observe whole development cycles. This approach could fall into the schema proposed by Dybå et al. [66, 67] where elements such as software process improvement and evaluation of empirical knowledge come along.

A. *Changes proposed in the development process*

As we mentioned in the previous section, we suggest incorporating two activities and observe the results in the project: (1) periodical maintainability assessments and (2) periodical refactorings in the development flow.

Maintainability assessments. We suggest having slightly longer retrospectives in order to incorporate maintainability assessments. Normally, retrospectives are meetings held by a project team at the end of an iteration to discuss what was successful about the time period covered by that retrospective, what could be improved, and how to incorporate the successes and improvements in future iterations or projects. Some organizations have retrospectives that take 2-3 hours, and some have shorter retrospectives (e.g., 30-60 min) and this varies depending of the length of the iterations or the organization itself. In CSoft's case the retrospectives are relatively short (1 to 2 hours), which allows additional space for this activity.

During the maintainability assessment, the teams and the architect(s) will identify the difficulties faced during the iteration as well as the problematic modules, and analyze possible reasons for these difficulties. Data drawn from code analysis done with NDepend can provide input for the discussion. This assessment should produce a maintenance backlog, which will have the function to depict general goals for restructuring and improving the maintainability of the system. The main requirement is that maintenance backlogs need to be enough specific in order to be brake-down into sprint backlogs. A maintenance backlog should resemble a product backlog[2] but focusing on maintainability improvements instead of new functionalities in the product.

Periodical refactoring. In order to incorporate periodical refactoring and constant improvement of design, we suggest an additional iteration at the end of the iterations (which will be called mini-iteration), where the maintenance backlog should be used as input for planning and executing the restructuring/refactoring tasks. Currently, the iterations are two weeks long, so we suggest a mini-iteration of one week. Each of the rules used in normal iterations will apply (e.g., deciding upon the backlog items, planning poker and distribution of tasks amongst the team members). Unit testing, integration testing and system testing should be planned as integral part of the mini-iteration as in a normal iteration.

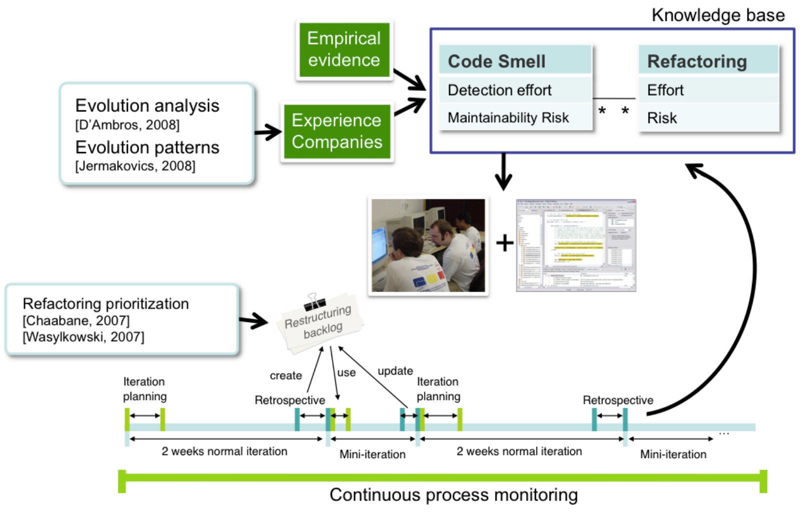

Figure 2: General model for suggested strategies within the development process

[2] According to Paetsh et al. (2003), a *product backlog* can be compared with an incomplete and changing requirement document containing enough information to enable the development during the iteration.

B. *Technical and knowledge framework needed for implementing the changes*

In this section, we present the required framework to guide the maintainability evaluations and the restructuring/refactoring during the mini-iterations. The process will basically rely on two aspects: (1) Tool support given by NDepend and (2) Refactoring knowledge base.

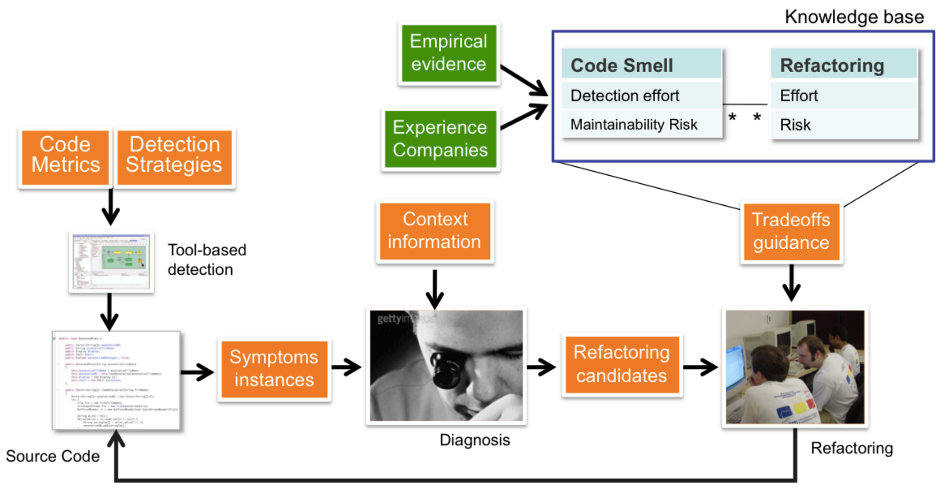

Figure 3: General model describing the technical and knowledge support for the improvement

Tool support. As we mentioned in section, we suggest using semi-automated code inspection. According to [68], code inspection consists of a peer review of any software product by individuals who look for defects using a defined process. In our case, the inspection will try to identify maintainability issues instead of defects, and we will use a set of software design attributes in order to guide the code inspection.

We define software design attributes as quantitative descriptors of potential design issues or flaws in the software. In our case, we define software design attributes as a set of code measures, code smells and design principle violations. Examples of code measures are lines of code (LOC) or Cyclomatic complexity (CC). Examples of OO metrics are Tight Class Cohesion and Number of Children from the work of Chidamber and Kemerer [69]. A comprehensive catalogue of code smells and their corresponding refactoring can be found in [2]. Conversely, design principle violations are somewhat similar to design anti-patterns (see [70] for further reference) and are also associated to the usage of a certain design pattern.

Currently, is possible to calculate several software design attributes by using NDepend. NDepend allows defining rules for searching instances of a given design attribute through a language called CQL or Code Query Language (see [71] for further details). This tool also provides visualization of different characteristics of the design of the code such as: Tree-map of diverse code measures, abstractness vs. instability diagrams, dependencies matrix and dependencies graph. The visualization functionality of NDepend can help detecting circular dependencies and other anomalies in the design of the code. Modules containing high values of code measures that are known to have a negative impact on maintainability; and modules presenting high number of instances of code smells and design violations can be prioritized for code inspection in order to determine how they affected the maintainability during the iteration.

Refactoring knowledge base. The result of the analysis described previously, together with what we call refactoring knowledge base should be used as input for producing the maintenance backlog. We suggest creating a knowledge base containing a list of the code smells and design principle violations (and their respective CQL searching rules, so they can be detected in the source code), which are considered relevant to CSoft's context.

The knowledge base should contain also the corresponding refactoring and restructuring strategies for each of the code smells and design violations (See Figure 4). Each of the code smells or design violations should be assigned a level of Criticality (high or low) depending of the potential negative consequences these may have in the system (as deemed by the architects or developers). Each of the refactoring strategies should be labeled according to their Cost (manual refactoring or automated refactoring) and Risk (high, medium or low risk). Table 3 presents an example of one design attribute (shotgun surgery) and its properties. It is deemed that this information could be useful for deciding which refactorings to do in case modules with this attribute are identified.

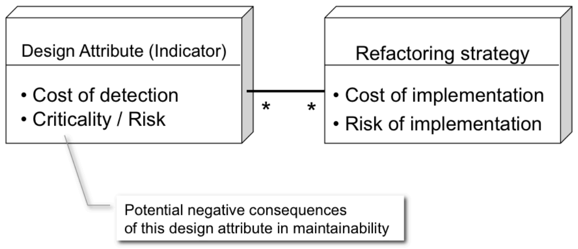

Figure 4: Class diagram to represent the connection between a design attribute and its corresponding refactoring strategy

This knowledge base will be stored in a common repository which all the members from the teams and the architects will have access. A simple format like a Wiki could suffice for this purpose, and it is recommended to pursuit simplicity in order to make the information more accessible to the members with different levels of experience in the team. This knowledge base is meant to support the planning of refactoring strategies. For instance, the prioritization of refactoring tasks could be done according to the potential negative effects of a given code smell or it can be used also for deciding which refactorings to perform. Some refactorings have lower cost (they can be solved automatically by using a tool) compared to others, which demand manual refactoring, so that kind of information should be contained in the knowledge base in order to provide practical information for refactoring decision making.

Table 3: Theoretical example of an item in the refactoring knowledge base with some of the properties of the design attributes and their corresponding refactoring

Design attribute	**Criticality**	**Possible refactoring strategies**	**Cost**	**Risk**
Shotgun surgery	High	Move method	Automated	Medium
		Move field	Automated	Low
		Inline class	Manual	High

C. *Role of the researcher within the study*

This study can be seen as the second part of an Action Research methodology [57], where the improvement or solution is implemented and evaluated (See “Act on Evidence” and “Evaluate results” in **Figure 5**). The role of the researcher would be to participate as a facilitator during the process and provide the technical expertise required to aid in the implementation of the technical and knowledge framework described previously. Although the researcher might provide feedback upon the process itself, he/she won’t participate directly in the activities undertaken during the development process, but rather be an observer.

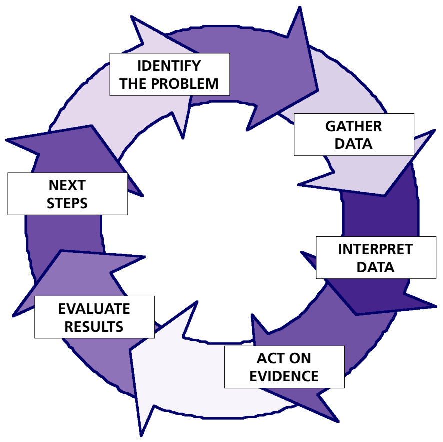

Figure 5: Diagram depicting the Action Research methodology from [72]

D. *Data to be collected during the study*

In order to be able to monitor the evolution of the system and the process, a measurement instrument for the product quality and other factors should be in place. For instance, there should be mechanisms for keeping track of the effort spent in different tasks and also the defects reported by developers and end users. This is very important because these would be the indicators to be used in order to evaluate the impact of the changes within the development process. The effort and defects (potentially including motivational factor amongst the members of the organization) should be registered before, during and after the refactoring iterations and in later development cycles. The researcher should in principle, be able to have access to the following data in order to help the organization to evaluate the cost-benefits of refactoring strategies and the activities incorporated within the development process:

1. Recordings / notes from the poker planning and retrospective meetings
2. Reports of defects found per module in the system per iteration
3. Reports of individual and team effort for given activities (refactoring, implementation, defect correction, etc) per iteration
4. Reports on the results from unit, integration and system level testing (the ones that are available according to the current practices of the organization)
5. The ‘knowledge base’ history (how it is used and how it is being updated)
6. Reports from the versioning system in order to observe for instance how quick new coming developers start committing changes into the system, as an indirect indicator of learning curve

The details regarding to the data analysis will not be covered in this report due to insufficient space, but a matrix depicting the sources and the measurements used for evaluating the method are described. It is assumed however that a consistent descriptive case study that can establish a causal chain of factors and events would provide enough insight in order to evaluate these strategies. Several researchers should handle the analysis of the recordings in order to avoid interpretation bias, and triangulation is suggested at data source level and data analysis technique level in order to ensure construct and internal validities.

Table 4: Example for potential operationalization of process outcome constructs

Process outcomes ➔ / **Data sources 🡻**	Defects introduced during maintenance	Maintenance tasks effort	Difficulty of understanding and analyzing code	Steepness of learning curve
1. Recordings / notes from planning poker meetings and retrospectives	X	X	X	X
2. Time sheet reported by developers and team leads		X		
3. Versioning tool	X			X
4. Test reports	X	X		

E. Period of inquiry

It is suggested initially a minimum of three iterations as the period of inquiry in order to observe any significant indicators of impact from progressive refactoring and the incorporation of semi-automated inspection. After that period, an evaluation following the lines of post-mortem analysis can be conducted in order to assess the effects of the changes in the product and the process. This process is assuming that the researcher(s) involved in the study are analyzing every iteration separately and will put together the findings from all three iterations at the end. After this period, changes in the strategy or the activities incorporated in the process improvement can be calibrated and the organization can continue the second stage for their improvement process. The participation of the researchers could continue, but the interaction level could decrease depending of the results from the initial study.

VIII. APPENDIX

Table 5: List of methodological contributions

M=Method
T= Tool
B= Both
	Contribution

M[73]	Size and Frequency of Class Change from a Refactoring Perspective
M[30]	Bad-Smell Metrics for Aspect-Oriented Software
M[40]	On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test
M[74]	Characterizing the Relative Significance of a Test Smell
M[32]	Leveraging code smell detection with inter-smell relations
M[75]	Towards a Catalogue of Refactorings and Code Smells for AspectJ
M[31]	Multi-criteria detection of bad smells in code with UTA method
M[38]	Refactoring Test Code
M[29]	A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws
M[76]	Model Refactorings through Rule Based Inconsistency Resolution
M[52]	Developing New Approaches for Software Design Quality Improvement Based on Subjective Evaluations
M[77]	Code evaluation using fuzzy logic
M[50]	Discovering Unanticipated Dependency Schemas in Class Hierarchies
M[78]	Restructuring Software Systems Using Clustering
M[79]	Lightweight Risk Mitigation for Software Development Projects Using Repository Mining
M[42]	Poor Performing Patterns of Code: Analysis and Detection
M[56]	Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software
M[27]	Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model
M[46]	A Classification of Crosscutting Concerns
M[51]	Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis
T[48]	Source code enhancement using reduction of duplicated code
T[33]	A Catalogue of Lightweight Visualizations to Support Code Smell Inspection
T[41]	Detecting Object Usage Anomalies
T[55]	Mining Software Repositories with iSPARQL and a Software Evolution Ontology
T[80]	An Interactive Reverse Engineering Environment for Large-Scale C++ Code
T[81]	Metrics Based Refactoring
T[82]	A Feedback Based Quality Assessment to Support Open Source Software Evolution: the GRASS Case Study
T[34]	Using The Meta-Environment for Maintenance and Renovation
T[83]	Ptidej: A Flexible Reverse Engineering Tool Suite
T[53]	Supporting Software Evolution Analysis with Historical Dependencies and Defect Information
T[84]	SOLIDFX: An Integrated Reverse Engineering Environment for C++
T[85]	Experiences in Adapting a Source Code-Based Quality Assessment Technology
T[86]	Metric-Based Selective Representation of UML Diagrams
B[28]	Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code
B[35]	Java quality assurance by detecting code smells
B[54]	Visual Identification of Software Evolution Patterns
B[43]	High-impact Refactoring based on Architecture Violations
B[44]	Towards Automated Restructuring of Object Oriented Systems
B[37]	Refactoring Test Code Safely
B[87]	Towards Experience-based Mentoring of Evolutionary Development

IX. References

1. Lindvall, M., et al., *Agile software development in large organizations.* IEEE Computer, 2004. **37**(12): p. 8.
2. Fowler, M., et al., *Refactoring: Improving the Design of Existing Code.* 2000: Addison-Wesley.
3. Gilb, T., *Competitive Engineering: A handbook for systems engineering, requirements engineering, and software engineering using Planguage*. 2005: Elsevier Butterworth-Heinemann. 480 pages.
4. Fægri, T.E. and G.K. Hanssen, *Collaboration and process fragility in evolutionarily product development.* IEEE Software, 2007. **24**(3): p. 96-104.
5. Schwaber, K., Beedle, M., *Agile Software Development with Scrum*. 2001: Prentice Hall.
6. 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.
7. 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.
8. Smaccia, P. *Getting rid of spaghetti code in the real-world: a Case Study*. 2008 [cited 2009 March]; Available from: http://codebetter.com/blogs/patricksmacchia/archive/2008/09/23/getting-rid-of-spaghetti-code-in-the-real-world.aspx.
9. Martin, P.Y. and B.A. Turner, *Grounded theory and organizational research.* The Journal of Applied Behavioral Science, 1986. **22**(2): p. 141-141.
10. Brererton, P., et al., *Lessons from applying the systematic literature review process within the software engineering domain.* Journal of Systems and Software, 2007. **80**(4): p. 571-583.
11. Abrahamsson, P., et al., *Agile software development methods - Review and analysis*. 2002, VTT Electronics. p. 112.
12. Cohen, D., M. Lindvall, and P. Costa, *An introduction to agile methods*, in *Advances in Computers, Vol 62*. 2004, ELSEVIER ACADEMIC PRESS INC: San Diego. p. 1-66.
13. 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.
14. Erickson, J., L. K., and S. K., *Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research.* Journal of Database Management, 2005. **16**(4): p. 88 (13).
15. Mäntylä, M., *An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement*, in *International Symposium on Empirical Software Engineering*. 2005.
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., J. Vanhanen, and C. Lassenius, *Bad Smells - Humans as Code Critics*, in *IEEE International Conference on Software Maintenance*. 2004, IEEE.
18. Counsell, S. and R.M. Hierons, *Refactoring test suites versus test behaviour: a TTCN-3 perspective*, in *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*. 2007, ACM: Dubrovnik, Croatia.
19. Counsell, S., et al., *Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS*, in *International Symposium on Empirical Software Engineering*. 2006, ACM: Rio de Janeiro.
20. Counsell, S., *Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?*, in *Research Challenges in Information Science*. 2008, IEEE.
21. Counsell, S., et al., *The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph*, in *Testing Academic & Industrial Conference*. 2006.
22. Bakota, T., et al., *Towards Portable Metrics-based Models for Software Maintenance Problems*, in *IEEE International Conference on Software Maintenance*. 2006.
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.* Journal of Systems and Software, 2006. **80**(7): p. 1120-1128.
24. Lozano, A. and M. Wermelinger, *Assessing the effect of clones of changebility*, in *IEEE International Conference of Software Maintenence*. 2008, IEEE.
25. Aversano, L., L. Cerulo, and M. Di Penta, *How Clones rew Maintained: An Empirical Study*, in *European Conference on Software Maintenenace and Reengineering*. 2007.
26. Fowler, M. and K. Beck, *Refactoring: Improving the Design of Existing Code.* 2000: Addison-Wesley. 464 pages.
27. Marinescu, R. and D. Ratiu, *Quantifying the quality of object-oriented design: the factor-strategy model*, in *Working Conference on Reverse Engineering*. 2004. p. 192-201.
28. Munro, M.J., *Product metrics for automatic identification of "bad smell" design problems in Java source-code*, in *IEEE International Symposium on Software Metrics*. 2005.
29. Salehie, M., S. Li, and L. Tahvildari, *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws*, in *International Conference on Program Comprehension*. 2006. p. 159-168.
30. Srivisut, K. and P. Muenchaisri, *Bad-Smell Metrics forAspect-Oriented Software*, in *IEEE/ACIS International Conference on Comouter and Information Science*. 2007. p. 1060-1065.
31. Walter, B. and B. Pietrzak, *Multi-criteria Detection of Nad Smells in Code with UTA Method*, in *Extreme Programming*. 2005. p. 154-161.
32. Pietrzak, B. and B. Walter, *Levaraging Code Smell Detetcion with Inter-smell relations*, in *Extreme Programming*. 2006. p. 75-84.
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.
34. Van den Brand, M.G., et al., *Using The Meta-Environment for Maintenance and Renovation*, in *European Conference on Software Maintenance and Reengineering*. 2007, IEEE Computer Society.
35. Van Emden, E. and K. Moonen, *Java quality assurance by detecting code smells*, in *Working Conference on Reverse Engineering*. 2002. p. 97-106.
36. Monteiro, M.P. and J.M. Fernandes, *Towards a catalog of aspect-oriented Refactorings*, in *International Conference on Aspect-Oriented Software Development* 2005. p. 111-122.
37. Guerra, E.M. and C.T. Fernandes, *Refactoring Test Code Safely*, in *International Conference on Software Engineering Advances*. 2007.
38. van Deursen, A., et al., *Refactoring test code*, in *eXtreme Programming Perspectives*, M. Marchesi, et al., Editors. 2002, Addison-Wesley: Reading, Massachusetts.

39. Van Rompaey, B., B. Du Bois, and S. Demeyer, *Characterizing the Relative Significance of a Test Smell*, in *IEEE International Conference on Software Maintenance*. 2006. p. 24-27.
40. 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.
41. 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.
42. Chaabane, R., *Poor Performing Patterns of Code: Analysis and Detection*, in *International Conference on Software Maintenance*. 2007.
43. Bourqun, F. and R.K. Keller, *High-impact Refactoring Based on Architecture Violations*, in *European Conference on Software Maintenance and Reengineering* 2007. p. 149-158.
44. Trifu, A. and U. Reupke, *Towards Automated Restructuring of Object Oriented Systems*, in *European Conference on Software Maintenance and Reengineering*. 2007. p. 39-48.
45. van Deursen, A., *Program comprehension risks and opportunities in extreme programming*, in *Working Conference on Reverse Engineering*. 2001. p. 176-185.
46. Marin, M., L. Moonen, and A. van Deursen, *A Classification of Crosscutting Concerns*, in *IEEE international Conference on Software Maintenance*. 2005. p. 673-676.
47. Belli, F. and R. Crisan, *Empirical Performance Analysis of Computer-Supported Code-Reviews*, in *International Symposium on Software Reliability Engineering* 1997.
48. 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.
49. Martin, R.C., *Agile Software Development, Principles, Patterns and Practice*. 2002: Prentice Hall.
50. Arevalo, G., S. Ducasse, and O. Nierstrasz, *Discovering Unanticipated Dependency Schemas in Class Hierarchies*, in *Software Maintenance and Reengineering*. 2005. p. 62-71.
51. Leitch, R. and E. Stroulia, *Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis*, in *International Symposium on Software Metrics*. 2003.
52. Mäntylä, M., *Developing New Approaches for Software Design Quality Improvement Based on Subjective Evaluations*, in *International Conference on Software Engineering*. 2004.
53. D'Ambros, M., *Supporting software evolution analysis with historical dependencies and defect information*, in *International Conference on Software Maintenance*. 2008. p. 412-415.
54. 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.
55. Kiefer, C., A. Bernstein, and J. Tappolet, *Mining Software Repositories with iSPAROL and a Software Evolution Ontology*, in *International Workshop on Mining Software Repositories*. 2007.
56. 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.
57. Avison, D.E., et al., *Action research.* Communications of the ACM, 1999. **42**(1): p. 94–97-94–97.
58. Glaser, B.G. and A.L. Strauss, *The discovery of grounded theory: Strategies for qualitative research*. 1967: Aldine Pub. Co.
59. Hanssen, G.K., et al., *Maintenance and Agile Development: Challenges, Opportunities and Future Directions.*
60. Dawes, R.M., D. Faust, and P.E. Meehl, *Clinical versus actuarial judgment.* Science, 1989. **243**(4899): p. 1668–1673-1668–1673.
61. Shanteau, J., *Competence in experts: The role of task characteristics.* Organizational Behavior and Human Decision Processes, 1992. **53**: p. 252–252-252–252.
62. Spengler, P.M., et al., *The meta-analysis of clinical judgment project: Effects of experience on judgment accuracy.* The Counseling Psychologist, 2009. **37**(3): p. 350-350.
63. Storey, M.-A., et al. *TODO or to bug: exploring how task annotations play a role in the work practices of software developers*. in *Proceedings of the 30th international conference on Software engineering*. 2008. Leipzig, Germany: ACM.
64. Mealy, E., *Improving Usability of Software Refactoring Tools*, in *Australian Softw. Eng. Conf. (ASEC)*. 2007. p. 307-318.
65. Maletic, J.I., A. Marcus, and M.L. Collard, *A Task Oriented View of Software Visualization*, in *Intl Ws. Visualizing Softw. For Understanding and Analysis (VISSOFT)*. 2002, IEEE.
66. Dybå, T., B.A. Kitchenham, and M. J\orgensen, *Evidence-based software engineering for practitioners.* IEEE Software, 2005: p. 58–65-58–65.
67. Dybå, T., T. Dingsyr, and N.B. Moe, *Process Improvement in Practice: A Handbook for It Companies (The Kluwer International Series in Software Engineering, 9)*. 2004: Kluwer Academic Publishers Norwell, MA, USA.
68. Fagan, M., *Design and code inspections to reduce errors in program development*, in *Software pioneers: contributions to software engineering*. 2002, Springer-Verlag New York, Inc. p. 575-607.
69. Chidamber, S.R. and C.F. Kemerer, *A metrics suite for object oriented design.* Software Engineering, IEEE Transactions on, 1994. **20**(6): p. 476-493.
70. Gamma, E., et al., *Design Patterns. Elements of Reusable Software*, ed. A. Wesley. 1995.
71. Smaccia, P. *Code Query Language*. 2009; Available from: http://www.ndepend.com/Features.aspx#CQL.
72. Ferrance, E., *Action Research. LAB Northeast and Islands Regional Educational Laboratory at Brown University*. 2000.
73. Counsell, S. and E. Mendes. *Size and Frequency of Class Change from a Refactoring Perspective*. in *Software Evolvability, 2007 Third International IEEE Workshop on*. 2007.
74. Van Rompaey, B., B. Du Bois, and S. Demeyer. *Characterizing the Relative Significance of a Test Smell*. in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*. 2006.
75. Monteiro, M. and J. Fernandes, *Towards a Catalogue of Refactorings and Code Smells for AspectJ*, in *Transactions on Aspect-Oriented Software Development I*. 2006. p. 214-258.
76. Ragnhild Van Der, S. and D.H. Maja, *Model refactorings through rule-based inconsistency resolution*, in *Proceedings of the 2006 ACM symposium on Applied computing*. 2006, ACM: Dijon, France.
77. Zikrija, A., B. Dusanka, and D. Aida, *Code evaluation using fuzzy logic*, in *Proceedings of the 9th WSEAS International Conference on Fuzzy Systems*. 2008, World Scientific and Engineering Academy and Society (WSEAS): Sofia, Bulgaria.
78. Serban, G. and l.G. Czibula. *Restructuring software systems using clustering*. in *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on*. 2007.
79. Masticola, S.P. *Lightweight Risk Mitigation for Software Development Projects Using Repository Mining*. in *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*. 2007.

80. Alexandru, T. and V. Lucian, *An interactive reverse engineering environment for large-scale C++ code*, in *Proceedings of the 4th ACM symposium on Software visualization*. 2008, ACM: Ammersee, Germany.
81. Simon, F., F. Steinbruckner, and C. Lewerentz. *Metrics based refactoring*. in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. 2001.
82. Bouktif, S., et al. *A Feedback Based Quality Assessment to Support Open Source Software Evolution: the GRASS Case Study*. in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*. 2006.
83. Gueheneuc, Y.G. *Ptidej: A Flexible Reverse Engineering Tool Suite*. in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. 2007.
84. Telea, A. and L. Voinea. *SOLIDFX: An Integrated Reverse Engineering Environment for C++*. in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. 2008.
85. Pantos, J., et al. *Experiences in Adapting a Source Code-Based Quality Assessment Technology*. in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. 2008.
86. Kollmann, R. and M. Gogolla. *Metric-based selective representation of UML diagrams*. in *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*. 2002.
87. Xing, Z. and E. Stroulia. *Towards experience-based mentoring of evolutionary development*. in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. 2005.