Project Report

Process and Product improvement plan for CSoft

Software Product and Process Improvement Course - INF5180

N.N.

[simula . research laboratory]

Table of Contents

1. Introduction

This document consists of a process and product improvement plan to be performed at the company CSoft. The process improvement will consist of incorporating periodical software maintainability assessment and refactoring to the development flow. The product and process improvement will have as goal to increase the maintainability of the code base as new functionality is incorporated to the releases. We suggest incorporating an additional iteration ad the end of each release in order to incorporate code inspection and refactoring activities. We suggest using semi-automated code inspection to support maintainability assessment and refactoring decision. The scope of the semiautomatic code inspection is limited to refactoring decision-making (where and when to refactor?). This implies that the actual refactoring process (how to refactor?) as well as the redesigning of the code structure will not be addressed in this document. The rest of the document is organized as follows: Sect. 2 describes the context of the improvement plan required in order to understand the problem being addressed by the improvement plan. Sect. 3 presents the improvement plan and the expected outcomes from each of the activities encompassed in the plan. Sect. 4 describes the rationale behind the selected approaches in the improvement plan.

1.1. Intended readers

This document is addressed to the software architects of CSoft, as well as researchers interested in software process improvement cases. Also it could be addressed to external software managers and software architects interested on examples of software process/product improvement plans for a given type of context and product. This document assumes that the reader understands basic concepts from software engineering and software development processes, and they should have good command of general terms related to agile development methodologies.

2. Context and problem statement

This section presents the context for the intended process improvement, by characterizing the organization, the software product and the current problems/challenges faced by CSoft. Our intention is to provide enough background information, which will enable to build the rationale for the strategy and the activities suggested in the improvement plan.

2.1. Organization

CSoft (an anonymized name) is a medium-sized Norwegian software company that develops, maintains and markets a product that is used for creating and running 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. There has been a gradual shift from building custom-made applications for the customers to a packaged software product.

2.2. Product properties

CSoft product 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. Some modules are central and are used in any configuration, while the use of the others depends on the situation. 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).

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 system is based on several technologies that have emerged over the years, such as older ASP solutions, COM+ components, VB6 code and other *old* technologies. Most of the *new* code is being developed in C# which by now is spread across approximately 160 .Net assemblies. The separation between the presentation- and the business layer is clear.

2.3. Development process

About five years ago the development process had become too slow and inefficient. Out of necessity CSoft changed to a radically different process – Evo [1] under the guidance of Tom Gilb, which originally defined the process [2]. Evo is an agile method comparable to the better-known Scrum-method [3], 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 [4, 5]. 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.

CSoft have tried TDD (Test Driven Development) but it failed according to the

[1] www.agilemanifesto.org

Architects team due to the high-pace of the development and the complexity of the code, which didn't allow the desirable delivery pace along with the generation of test cases. CSoft manages at some extent a set of unit tests (reaching 60% of code coverage), but relies on integration and system level testing for identifying defects. The system and integration testing is done before a major build and/or before the end of an iteration.

2.4. Identified problems

CSoft is currently suffering a very common phenomenon, which is called by several names such as: *software entropy*, *code decay*, *software rot*, or *software erosion*. This phenomenon consists of an imminent increase of code size and complexity, which results in a decrease of the productivity of the development team, as well as the product quality. The most obvious example of this problem 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. Through a series of interviews with the 'Architect Team', a team composed by 4 members all with good knowledge of the system, the domain and programming expertise, we have identified a series of problem areas: a) analyzability and learnability, b) changeability and deployability, c) testability and stability and d) organization and process.

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. 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. Developers as a result generate duplicate code - instead of changing existing, working code, the developers rather separately develop a new piece of code, which he or she then has full control over.

Changeability and deployability: According to the architects, a result of this code duplication is the shotgun surgery code smell, which manifests when developers are performing small changes, e.g. a single line of code, and they are forced to identify and alter code in 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 according to their coverage criteria is insufficient 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.

Organization and process: As both the business domain and the system are highly complex, each of the 11 development teams (consisting of 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.

2.5. Desired changes in the development process

As part of the discussions with the CSoft architects, we also collected several of their 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. 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. At the horizontal architecture point of view, the system should better separate business and platform related code, eliminating dependency cycles and inverted dependencies.

(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™ [6], by defining CQL[7] rules to detect design flaws and monitor potential problems. The architects believe that this approach would considerably reduce the fear of changing the code.

2.6. Short and mid-term allocation of resources in CSoft

Through 2008, a series of meetings and a case study was conducted in order to understand the context of the company and the challenges they were facing with respect to the maintainability of the product (the results from this case study are reported in sections 2.1 to 2.4). By 2009, the architect team convinced the management to devote 40% of the effort (from the architect team) to improve the maintainability of the systems. Currently, CSoft is planning a preliminary release (called "*rehearsal period*") by August 2009, focusing their resource allocation on improving the maintainability of the system and freezing the development. This means that no significant new functionality will be added to the product and the effort will be put on solving defects, attending special requests from customers who have already acquired the system and improving the maintainability to the product. An official release is planed for January 2010.

2.7. Focus of the improvement plan

The product and process improvement will have as a focus to increase the maintainability of the code throughout the evolution of the product. The main focus we will try to have is to spread the knowledge of the "gurus" among the teams, reduce the "fear to change code" and gradually reduce changeability and deployability issues. This means that we are supposed to perceive a gradual increase on the maintainability of the code as new functionality is added and new releases are created. We suggest for reaching these goals to integrate periodic maintainability assessment and code refactoring into the development flow. These two activities can be supported by an approach we call semi-automated code inspection, which consists of a combination of tool aided static analysis and subjective evaluation. We suggest using NDepend™ to support this approach, since it has already been used by the architect team for detecting circular dependencies and other anomalies in the design of the code. The improvement plan will consist of putting in place these two activities in the development flow involving three different levels in the organization: the architect's team, the development team and individual developers.

2.8. Scope for future improvement plans

We are aware that these two activities could support different kinds of decisions, and this highly depends of the organizational level where the decisions are taken. Some examples are: architectural improvement, redesign decision, choice of implementation of new functionalities for a given release, and refactoring of test sets. The process and information demanded for this type of decisions goes beyond the scope of this plan, and they can be viewed as potential areas for future improvement plans. Some other uses of semi-automated code inspection could be for educational purposes (show the junior programmers examples of code "offending" design principles and have given problems in the past), or for cost-benefit analysis of refactorings through what-if scenarios in order to perform refactoring decision-making.

3. Software Process Improvement Plan

This section presents an overview of the improvement plan and the intended activities or stages in order to conduct it. First, we explain the general model of the process modification suggested in the development flow. Secondly, we provide a description of the technical/knowledge framework required for implementing this change. We present a roadmap of the entire process improvement plan. Consequently, we present the required steps to enable the implementation, evaluation and calibration of the process improvement. For each of these steps, we describe the different activities encompassed, the timeframe, participants and responsible for carrying out the activities.

3.1. Model chosen for the process improvement

Since the way CSoft implemented Evo in practice resembles more to the Scrum methodology, we will use the terminology from Scrum for defining the changes into the development process. As we mentioned in the previous section, we suggest incorporating: (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 (information on how to use NDepend will be provided in section 3.2). 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.

[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.

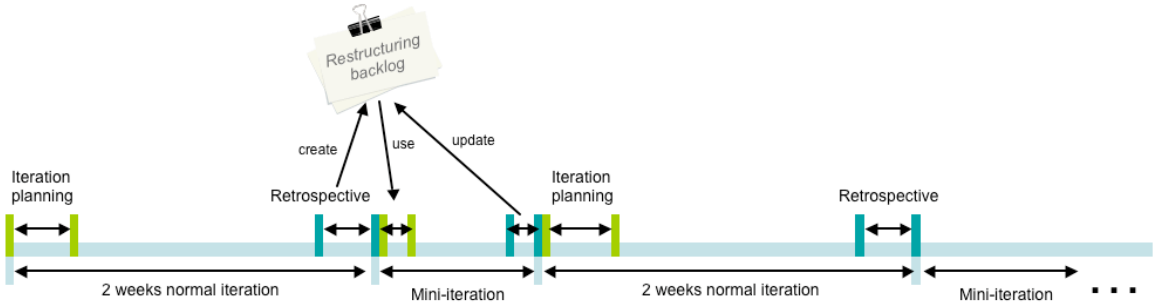

Figure 1: General model for integrating the restructuring mini-iteration in the development flow

3.2. Technical and knowledge support for the improvement

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.

Tool support. As we mentioned in section 2.7, we suggest using semi-automated code inspection. According to [8], 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. Design flaws are commonly associated with maintainability issues as well as other software qualities, such as correctness, reliability or efficiency. 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). These measures apply to any programming language, as opposed to Object Oriented (OO) code measures, which are specific to OO paradigm (e.g., Tight Class Cohesion or TCC). Further details on OO code measures can be found in the work of Chidamber and Kemerer [9]. A *code smell* is a surface indicator (also known as structural symptom by Marinescu [10]) that usually corresponds to a deeper problem in the system. Code smells could be used as guidance for recognizing situations where refactoring is needed. A comprehensive catalogue of code smells and their corresponding refactoring can be found in [11]. Conversely, *design principle violations* are somewhat similar to design anti-patterns (see [12] 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 [7] 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 2). 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 labelled according to their *Cost* (manual refactoring or automated refactoring) and *Risk* (high, medium or low risk). Table 1 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.

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.

The process for assigning the values to the properties of the code smells and refactorings; as well the process for updating the knowledge base will be explained in section 3.3.

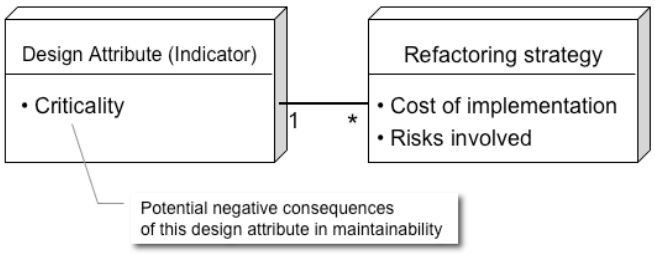

Figure 2: Class diagram to represent the connection between a design attribute and its corresponding refactoring strategy

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

Table 1: Theoretical example of an item in the refactoring knowledge base with some of the properties of the design attributes and their corresponding refactoring

3.3. Overview of the plan and roadmap

The overall plan is to incorporate the suggested activities in a "*small scale*" within the "rehearsal period" (see section 2.6) in an incremental style. Afterwards, we intend to evaluate the results from the small-scale version, and iterate with an adjusted "big-scale" improvement process according to the results from the evaluation. "Small-scale" implies that we will only involve a limited number of teams to adopt the changes in the development process. We expect that this will enable a more concise process to be put in place by the official release period in January 2010. The specific goals with the stages are orthogonal to the steps specified in the PROFES manual [13].

Time scope	Step	Specific goals
Jan 2008 - April 2009	Step 1: Understand	Understanding of the problems
		Formulation and agreement of overall improvement strategy
May - June 2009	Step 2: Propose the plan	Definition of goals and process for measuring the goals
	Step 3: Prepare plan	Planning activities for implementing the process change
July - August 2009	Step 4: Implement plan	Implementation of the improvement plan
	Step 5: Evaluate and adjust	Evaluation of the improvement plan and adjustment of the activities
September-December 2009	Step 4: Implement adjusted plan	Implementation of the adjusted improvement plan
	Step 5: Evaluate and adjust	Evaluation of the improvement plan and adjustment of the activities
January 2010	Step 6: Evaluate the overall change and extend the scope of the improvement plan	Evaluation of overall improvement plan and its impact in the process and product

Table 2: Roadmap of the process improvement plan, steps and expected main outcome from each of the steps in the process improvement plan

3.4. Steps and activities

This section presents a detailed description of the proposed steps (and corresponding activities, their outcome, responsible and participants) for implementing, evaluating and adjusting the process. The researcher in this case would be the process/product improvement facilitator or initiator. The activities in each of the steps are assumed to be sequential. The order of the steps might be cyclical, but this will be depicted clearly in the overall roadmap in section 3.3.

Step 1: Understand the context. In order to understand the context and define a realistic focus for the improvement plan, we suggest performing the following activities: an explorative interview, identification of problem areas and literature review, and motivational screening meeting.

Activity 1 – Interviews with software architects

Description	Responsible	Participants
In-depth interview with the architects from the four-person architecture team. The main outcome from this activity will be the interview transcripts.	Researcher(s)	Architects

Activity 2 – Identification of problem areas and potential solutions

Description	Responsible	Participants
Identification of main problem areas by doing text analysis on the interview transcripts. This analysis should be complemented by a literature review on the current technologies and empirical knowledge on refactoring and code smells detection. The outcome from this activity should be the identification of the maintenance problems and the notion of an overall strategy, which could address these problems (or a subset of the problems).	Researcher(s)	Researcher(s)

Activity 3 – Screening of engagement and plan feasibility

Description	Responsible	Participants
Motivational presentation and in-depth contextual inquiry (in the form of open discussion) should be carried out in order to assess the interest and engagement of the relevant constituency groups (such as architects and managers). The presentation should: (a) provide the evidence for the emergence of maintainability requirements in the code base by describing the identified problems, (b) propose the overall strategy for the improvement plan and (c) conduct a brainstorming session in order to get more specific improvement goals. The outcome from this activity will be an assessment for a "green light" for the improvement plan and the focus/scope for the improvement.	Researcher(s)	Architects

Step 2: Developing the improvement plan. In order to draw a concrete improvement plan with specific goals and enable its implementation, we suggest first presenting an outline of the plan, its focus and scope. Secondly we suggest deriving the specific goals for the process improvement, and finally deciding upon how to evaluate the results of the improvement.

Activity 4 – Definition, presentation and negotiation of improvement plan

Description	Responsible	Participants
After the focus of the process improvement is clear, the researcher should draw a more concrete proposal for the improvement plan and present it to the architects. A discussion should be held in order to "negotiate" or calibrate the plan according to the architects' notion of available resources, time scope, priorities and costs. The result will be an improvement plan comprising a description of steps, activities and time scope of the whole improvement plan.	Researcher(s)	Architects

Activity 5 – Concept mapping session for identifying the improvement goals

Description	Responsible	Participants
The researcher should plan and carry out a concept mapping session in order to identify the goals of the improvement (based on the identified problems in section 2.4) and define measures that could operationalize the goals. The concept mapping session will result in a list goals for the improvement and a list of measures or "health indicators" to determine if the improvement in the process is effective. Some examples of measures or health indicators that might be useful for evaluating the effectiveness of the improvement plan are suggested in Section 5, Appendix A.	Researcher(s)	Architects, Team lead, and some team members

Activity 6 –Plan for measuring the improvement goals (measurement plan)

Description	Responsible	Participants
A separate meeting should be planned and carried out by the architecture team, in order to enable a process for collecting the measurements needed for evaluating the improvement in the process and the product. For each of the measures, the participants should discuss and make sure that: the methods for performing the measurement are practical, the required technical infrastructure is available and it is possible to summarize the measures into higher-level indicators that can be shown to management in the final stages of the improvement process.	Architects	Team leads, Drift

Step 3: Preparing the implementation of the improvement plan. Once the goals of the improvement are set and the means for measuring and evaluating the improvement are defined, it is required to set the technical and knowledge framework we mentioned in section 3.2, and prepare the different participants for the actual implementation of the process/product improvement.

Activity 7 – Setting up the refactoring knowledge base

Description	Responsible	Participants
The researchers and the architects should build together the "refactoring knowledge base". Researchers initially suggest a list of design attributes and a discussion should be held with all the participants about the importance of those and the risks associated to them in the context of CSoft. Active discussion and results from the literature review from activity 2 should be used in order to define the properties of the design attributes (code smells and design violations) as described in section 3.2. It should be decided upon the technical infrastructure for hosting the knowledge base, and a process for updating the knowledge base (e.g., access levels upon architects, researchers, team leads and team members). An example of defining the risk property of a refactoring strategy could be: "Some recent literature in empirical software engineering has determined that for a code smell M, there are two types of possible refactorings (Y and X), where Y is more prone to change the dependency structure in the code, therefore more risky than X.	Researcher(s) Architects	Architects, Drift

Activity 8 –Introduction of improvement plan to team members

Description	Responsible	Participants
The researcher(s) and the architects will describe the improvement plan, responsibilities and changes that will become effective in the development process (including the process measurement activities) will be informed to everyone who is involved in the process.	Researcher(s) Architects	Architects, Drift, Team leads, and team members

Activity 9 – Setting up the NDepend tool and provide mentoring to the team members

Description	Responsible	Participants
The architects should plan and implement informative and training sessions for the team leaders so they can learn how to use the NDepend tool.	Architects	Team leads

Activity 10 – Setting up the NDepend tool and provide mentoring to the team members

Description	Responsible	Participants
The architects should plan and implement informative and training sessions for team leaders and developers to use/update the refactoring knowledge base in their every day work.	Architects	Team lead, and developers

Step 4: Implementation of the plan. The suggested changes in the development process are implemented in this step. The specific activities in this step are: implementation of measurement plan, preparation for retrospective, maintainability assessment during the retrospective, planning meeting for the mini-iteration, implementation of maintenance tasks during the mini-iteration, retrospective of mini-iteration, updating of maintenance backlog and refactoring knowledge base.

Activity 11 –Implementation of measurement plan during normal iterations

Description	Responsible	Participants
During the normal development iteration, the architects, team leads and team members will keep track of the "health indicators" agreed upon from the activity 5 and 6. Architects and team leads are mainly responsible of making sure that the activities arranged for the process measurement are followed.	Architects, Team leads	Architects, Team leads, Team members

Activity 12 –Preparation for extended retrospective

Description	Responsible	Participants
The architects will collect and summarize all the "health indicators", the design attributes from the code as well as the diagrams generated by NDepend so they can be used in the retrospective	Architects Team leads	Architects Team leads

Activity 13 –Extended retrospective/maintainability assessment

Description	Responsible	Participants
The architects will present the data and diagrams generated by NDepend and packages or classes with high values will be identified. The team leads and the architects will present the "health indicators" collected during the iteration (e.g., burn-down charts, defect report summary). Team members will report on the main problems and difficulties faced during the iteration. The discussion process will consist of relating the identified packages, classes or methods to practical issues (e.g. if the package has a high bug rate, if the class is well known for its "unpredictable behaviour", etc). This will enable to relate the design attributes to different types of issues and could help to categorize the design attributes according to the "severity" of their practical consequences[3]. Based on the discussion, cross cutting concerns are identified and high-level preventive maintenance goals are formulated. The outcome from this retrospective will be the Maintenance backlog.	Architects, Team leads, Team members	Architects, Team leads, Team members

Activity 14 –Iteration planning meeting for mini-iteration

Description	Responsible	Participants
Teams plan for their mini-iteration, by using the maintenance backlog. The architect will participate in the prioritization of the maintenance backlog items. Once the backlog items are prioritized for the current mini-iteration, the team will use the refactoring knowledge base and active discussion to breakdown the backlog into a series of atomic refactorings. The choice of refactorings as well as the strategies for high level restructuring will be decided based on discussion and the refactoring knowledge base. Planning poker can be used for estimating time for refactorings.	Team members	Architects, Team leads, Team members

[3] For the next retrospective the "health indicators" from the past iteration will be compared to the latest iteration to observe any meaningful differences.

Activity 15 –Implementation of measurement plan during mini-iteration

Description	Responsible	Participants
Normal rules for development iteration apply. The architect is involved (during initial stage as fostering agents) with the team lead and team members. Team members should keep track of the refactoring effort as well as other "health indicators".	Team members	Architects, Team leads, Team members

Activity 16 –Retrospective of mini-iteration

Description	Responsible	Participants
Retrospective of mini-iteration is performed to assess the effort and risks involved in different refactorings applied during the iteration. Based on the results from the retrospective, the refactoring knowledge base and the maintenance backlog are updated, Also discussion should be held on lessons learned, good practices, bad practices for adjusting the mini-iteration practices.	Team members	Team leads, Team members

Step 5: Evaluating and adjusting the improvement plan.

Activity 17 –Evaluating the implementation of improvement plan

Description	Responsible	Participants
Before starting on the new iteration, a discussion should be help on lessons learned from the normal iteration, the extended retrospective, successful changes, unsuccessful changes, difficulties, perceived benefits, and based on that provide recommendation on continuing the process or adjusting the activities proposed in the improvement plan, in a very similar style to post-mortem analysis. Once the adjustments are decided, the architects and team leads should make them effective by informing the team members of the changes.	Architects, Researcher(s)	Architects, Team leads

Activity 18 –Implementing the adjustments in the improvement process

Description	Responsible	Participants
A new iteration with the changes in the improvement plan is held. In the same way as in activity 11, the architects, team leads and team members will keep track of the "health indicators" agreed upon from the activity 5 and 6. Architects and team leads are mainly responsible of making sure that the team members follow the activities arranged for the process measurement.	Architects, Team leads	Architects, Team leads, Team members

Step 6: Extend the scope for the improvement plan.

Activity 19 –Evaluating the improvement plan – Mini-Retrospective (2nd stage)

Description	Responsible	Participants
At the end of the second mini-retrospective, there should be a meeting where the architects will present lessons learned from the whole process, the successful practices, the unsuccessful practices, results on improvements on "health indicators", recommendation on continuing the process which should be based on evidence (anecdotic and empirical). Based on this information, a discussion will be held in order to decide on extending the scope of the improvement plan, keeping the improvement plan as it is, modifying it or cancelling it.	Architects	Managers, Architects

4. Rationale behind improvement plan

The current section provides the reasoning behind the main strategy, which is the usage of semi-automated code inspection and the integration of maintenance evaluations in the retrospectives, as well as the introduction of mini-iterations for restructuring purposes. The rationale is linked back to the problems identified in section 2 and sources found in the available literature related to software process improvement, agile methods and software design. Additionally to the rationale on the main strategies, we will also describe the reasoning behind the choice of NDepend and the usage of the refactoring knowledge base. The rationale for some other decisions in the plan is explained as well.

Why maintenance evaluations? A structured and repeatable assessment of the attributes of the code and their effects in the development process and product quality could drive the efforts for restructuring and enhancing the design of the code. It also will enable the observation of the effects of the changes in the code from the identified problems (see section 2.4) point of view.

Why semi-automated code inspection? As described in 2.5, the architects have found that automation for improving the quality is necessary in their development flow. Nevertheless, technology for fully automated refactoring is not yet available due to the complexity that is involved in this process. According to Anda [14], a combination of static analysis and subjective judgment is likely to be a feasible option since they both cover different aspects of maintainability and are therefore complementary (this is the notion of "semi-automated"). Quantitative data (design attributes) such as measures of code and code smells could guide the exploration of extensive code thus may facilitate de identification of problematic areas/modules in a reasonable time frame.

Why integrate maintenance evaluation to retrospectives? Retrospectives are good opportunities to evaluate the overall maintainability of the system, because results from static analysis can be related to the experiences from the teams, team leads and the architects during the iteration. Another good aspect of retrospectives is that they are periodical, thus already integrated in the normal working flow. This considerably facilitates the implementation of the maintenance evaluation and at the same time it supports the aim of the architects in section 2.5 where it was stated the need of *continuous monitoring of quality*.

Retrospectives also represent a common space where the different teams integrate after iterations. This is a good opportunity to exchange knowledge on the issues faced by the different teams, potentially leading to the identification of cross cutting concerns by comparing the types of problems and domain objects which were displaying those issues. This may be addressing one of the problems identified in section 2.4 "*Organization and process*".

Why introduce mini-iterations? As mentioned in section 2.6, a major factor for this choice is that the management has given a "period of grace" of six months where new development is virtually "frozen" in order to focus on restructuring and refactoring tasks. Although some development will still performed, there will be enough resources for a dedicated time span for refactoring. There are mainly two options for allocating the time span: one is to incorporate the refactoring activities to the normal workflow, by adding refactoring tasks into the product backlog. This first option has a series of disadvantages: one is that is hard to keep track of refactoring effort measurement, and second is the potential reluctance of developers/team leads/managers to prioritize refactoring activities over development or defect correction (which may have higher priorities). Also, if the refactoring effort is not recorded, then it will be considerably difficult to assess the cost-effectiveness of refactoring and restructuring strategies.

The second option is to have a dedicated mini-sprint for refactoring, with its own maintenance backlog. This second option facilitates the monitoring of the refactoring

process and the planning of the refactoring activities, as opposed to having these activities mixed up with the normal development activities. Also in the initial stage when the refactoring activities are introduced, this may lead to an increase of defects due to the lack of knowledge on effects of different refactorings. A separate iteration will allocate space for integration/regression testing, and this will ensure that defects are identified and corrected after the refactoring changes, resulting in higher confidence of the system quality. This also will provide data on how risky are certain refactorings by comparing the type of refactorings performed versus reports of defects after the refactorings have been implemented.

Having a separate iteration gives space for planning the refactoring. This will enable to define levels of decision for refactoring (who is responsible for what and when?). This is a process, which will be highly difficult to monitor and control without explicit planning. An anecdotic case from CSoft is a developer who had the tendency to refactor "out of control" holding the code for long periods of time, not letting anyone to modify his module.

Having a separate maintenance backlog may avoid the "big bang" restructuring tendency and will enable the refactoring changes to be atomic (through maintenance backlog items), so tests could be run after each backlog item is finished. Nevertheless, is important to notice that not all the refactorings can be atomic, so risk and effort need to be estimated by the responsible of the task.

Nevertheless we do not imply that this model must be kept in the long run. Once refactoring activities become more mature in the development teams and the effort/risk of refactoring activities become more predictable, then it should be possible to have the mini-iteration merged into the normal development iteration.

Why to have a "refactoring knowledge base"? We conjecture that a major reason of the inherent complexity of refactoring tasks is because there is a lack of information (from empirical viewpoint) about the costs and effects of refactorings in the system and also there are not too many automated means for ensuring compliance with "good design principles". This difficults the process of making "smart" refactorings, meaning choosing the refactorings that are less costly, will give the most benefits and will cause no side effects. A starting point here is to use some of the scarce evidence in effects of refactorings and their indicators (code smells and design principle violations) and start building a knowledge base by observing the effects of the implementation of these refactoring in the system. This will enable a learning process, which will support better understanding of the system, more confidence in refactoring decisions and loose the "fear to change" (see section 2.4). The usage of software measures allows the usage of the knowledge derived from empirical studies on measures of software structural attributes, and this we deem it useful for identifying problematic areas in the code.

Why use NDepend? The usage of NDepend is because is already used by the architects and it enables the detection of code smells and design principle violations besides the calculation of the code measures. Besides this functionality, the tool provides different diagrammatic views, which could facilitate considerably the discussion in the teams.

Why incremental approach and gradual improvement? As mentioned in section 3, we suggest *incremental adoption* [15] of the activities by implementing the process change by only involving some teams. Hodgetts [16] recommends incremental integration of agile activities in the development flow, and deems this strategy as a key factor for successful adoption of agile development practices in the industry.

In order to enable the team members to assimilate the process and learn about refactoring by actually "implementing them" (-- the "learning and doing" approach by Hodgetts) an incremental approach is the most recommended.

Conversely, we don't expect high-impact or immediate improvements in the results from the changes implemented in the process. This means that we would expect a gradual improvement in the understandability and performance of the teams with respect to the code and maintenance tasks. Is not easy to determine time span for measuring the impact of an improvement plan, an reason could be that some projects have a long life cycle (e.g. Siemens) where they expect that the improvement will provide improved performance in three-to-five years (see Paulish et al. [17]). We would expect that some aspects might have immediate responses as some other may not, and this could be observed during the retrospectives. Architects need to have "leadership" spirit in order to convince the developers that the changes put in place in the process are the best things to do although no clear/immediate results could be seen after the first iterations. The important thing is to make people aware of the *effort curve* before any benefits can be clearly perceived.

The initial evaluation of the process improvement can also help to determine the scope in terms of the time span expected for perceived benefits (management of expectations) and also the evaluation of the practices so far (calibration of activities and evaluation of the strategy as a whole and the individual activities).

Why involving architects in most of the team activities? Architects already spend some time with junior developers by doing refactoring together; this activity could be incorporated as a form of *refactoring by pair programming* between team leads and team members. The feedback process when planning and implementing refactorings could be a good alternative for loosing the "fear of change" and spreading the knowledge of the system across the members of the team. The fostering of developers by architects is possible because of the current size of the development organization and because each team has a "sponsor" architect.

5. Appendix

Appendix A: Measurements for evaluating the maintainability and other product qualities improvement

1.	Perceived ease for implementing changes (measurement of “mood”)
2.	Perceived ease for learning new modules (by the incoming developers)
3.	Perceived level of interaction between the “areas of concern” per iteration
4.	Test coverage (currently at 60%) per iteration
5.	Number of issues reported from system and integration testing per build per iteration
6.	Errors rate reported per build per iteration
7.	Number of issues reported from customers per iteration
8.	Effort for actual change implementation per iteration, per release
9.	Effort for fixing reported bugs after release
10.	Effort for conducting the refactoring/redesign
11.	Total effort per iteration (including implementation, testing and issue solving)

Appendix B: Design attributes from [18] and [19] deemed useful by the architects in their project

1.	Interface segregation principle or ISP (separation of concerns)
2.	God class (related to ISP)
3.	Shotgun surgery
4.	Unused class/method/parameter/field
5.	Misplaced class (Odd man out)
6.	Feature envy (Anemic domain)
7.	God Method
8.	Single Responsibility Principle

6. References

1. Gilb, T., *Competitive Engineering: A handbook for systems engineering, requirements engineering, and software engineering using Planguage*. 2005: Elsevier Butterworth-Heinemann. 480 pages.
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. Schwaber, K., Beedle, M., *Agile Software Development with Scrum*. 2001: Prentice Hall.
4. 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.
5. 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.
6. Smaccia, P. *Getting rid of spaghetti code in the real-world: a Case Study*. 2008; Available from: http://codebetter.com/blogs/patricksmacchia/archive/2008/09/23/getting-rid-of-spaghetti-code-in-the-real-world.aspx.
7. Smaccia, P. *Code Query Language*. 2009; Available from: http://www.ndepend.com/Features.aspx#CQL.
8. 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.
9. 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.
10. Marinescu, R., *Measurement and quality in object-oriented design*, in *Intl Conf. on Softw. Maintenance (ICSM)*. 2005, IEEE. p. 701-704.
11. Fowler, M., et al., *Refactoring: Improving the Design of Existing Code.* 2000: Addison-Wesley.
12. Gamma, E., et al., *Design Patterns. Elements of Reusable Software*, ed. A. Wesley. 1995.
13. Fraunhofer, *PROFES User Manual*. 2000, Stuttgart: Fraunhofer IRB Verlag.
14. Anda, B. *Assessing Software System Maintainability using Structural Measures and Expert Assessments*. in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. 2007.
15. Orlikowski, W.J. and no, *Radical and incremental innovations in systems development : an empirical investigation of case tools*. 2003, Massachusetts Institute of Technology (MIT), Sloan School of Management.
16. Hodgetts, P. *Refactoring the development process: experiences with the incremental adoption of agile practices*. in *Agile Development Conference, 2004*. 2004.
17. Paulish, D.J. and A.D. Carleton, *Case studies of software-process-improvement measurement.* Computer, 1994. **27**(9): p. 50-57.
18. Benestad, H., B. Anda, and E. Arisholm, *Assessing Software Product Maintainability Based on Class-Level Structural Measures*, in *Product-Focused Software Process Improvement*. 2006. p. 94-111.
19. Marinescu, R., *Measurement and Quality in Object Oriented Design*, in *Department of Computer Science*. 2002, "Politehnica" University of Timisoara.