Cynical Reengineering

Kristoffer Kvam, Daniel Bakkelund and Rodin Lie

Telenor, CRM, Business Logic,
kristoffer.kvam@telenor.com
daniel.bakkelund@telenor.com
rodin.lie@telenor.com

Abstract. This paper presents a solution for saving large systems from increasing entropy. The solution is proven on a large middleware platform giving good results. The method's objective is to rework the system so that reengineering investments pays off. Reaching agile practices is the methods basis. In order to reach the objective the method cynically relies on measurements to find unwanted characteristics of the system. Subjective opinions due to ownership and politics are ignored in the method. An extensive open source tool, the XRadar [1], is given to the community to make these measurements. Various symptoms and measurements are identified and approaches to solutions are discussed.

Keywords: Reengineering, Refactoring, Software Metrics, Open Source

1 Introduction

1.1 Challenge

One of the major challenges we face in software development are the old systems. Systems having reached maturity often have a mysterious tendency to produce highly unexpected errors and maintenance is a pain. Taking inspiration from the Second Law of Thermodynamics some call this phenomenon increasing system entropy: In time a system experiences increasing disorder if not explicitly tended to [2]. This disorder adds unnecessary complexity to the problem domain's inherent complexity, something many organisations experience as their systems mature. A great deal of time is being spent fixing bugs and testing the fixes while development of new functionality is costly, risky and likely to introduce regression errors.

Techniques within the Agile initiatives such as automated acceptance testing, test driven development, continuous integration and refactoring all aim at preventing systems to end up as described above [3]. This idea works well with new systems development, but what about all the existing systems?

One alternative is a complete rewrite, but the cost might not pay off the investment. Rewriting a system is a risky and tricky affair, but not something we shall venture into in this paper.

The other alterative is to facilitate agile practices by reengineering the old system. This process is not free of challenges, whereof some of the more obvious ones are:

- How to get management acceptance?
- Where to start?
- What to prioritise?
- How to reduce risk?

In addition to this you will most likely have very important organizational challenges, such as the process and culture change needed to establish the agile practices. These are not in the scope of this paper.

The following presents a solution to the challenges mentioned above (save the cultural and process related ones). The solution is based on quantitative measurements to make objective reengineering decisions. As a means of obtaining the measurements a new free open source tool, the XRadar [7], is introduced.

1.2 Telenor

Telenor is Norway's largest telecommunications company with numerous international interests.

COS is a middleware system designed to give front-end applications a consistent view across multiple back-end systems. There are more than 20 front end applications serving retail outlets, customer support, large corporate customers and internal functions. The back end systems include Sybase and Oracle databases, network connections and mainframes, all of which are logically interconnected through the use of batch jobs, scripts and database stored procedures.

COS has evolved over 5 years into a large system, composed of many subsystems. After a period of sustained development the problems were manifold. The system state corresponded well with the description of increasing entropy as defined above.

The Pareto project was instigated with solving the problems. The results presented here rely heavily on the work done during the project.

2 Objectives

A project proposal that sounds like "saving the system from the entropy spiral" will likely face scepticism from management. The project's business case must show that the increased life of the system defends the investment. After all, a very sensible alternative from a business point of view is to let the system die a silent death and invest in new development or buying a packaged solution. (In some cases that will also be the better solution.) The following issues must be taken into consideration when deciding the reengineering roadmap.

2.1 Reduced Time to Market

Reducing the time from an idea is created to it is put into production is a critical competitive factor for businesses today. Often the willingness to pay for a change is very high but the organizations ability to implement the changes is limited and time consuming. Hence, reducing the time to production of highly prioritised changes is often the most important factor for the business.

2.2 Increased Flexibility to Change

Having a flexible system architecture usually means being able to introduce large changes into the system at low cost and low risk. Flexibility also increases insight into the systems secrets causing general maintenance costs to decrease. Finding system bottlenecks and optimising performance will also be much easier once the systems architecture is flexible.

2.3 Reduced Number of Critical Errors

Developing zero defect software is one of the observed merits of agile practices such as XP [4]. Unfortunately, this is not the track record of the typical mature system one faces. Even small changes can have catastrophic effects on the stability of the system. Since these mature systems often are heavily depended on, downtimes can amount to painful losses for the business. Consequently, a focus on this area is usually appreciated by the organisation.

3 The Cynical Reengineering Method

A reengineering project is likely to have a mandate corresponding to the objectives mentioned above. In the targeted system there are probably several causes to the problems. Below, typical causes are identified and means of measuring them are presented. Once you have identified the most pressing problems, ways of attacking them are needed. We give you details on that. An unfortunate problem with a reengineering project is that you discover so many issues you want to resolve, but must prioritise the problems that give the greatest benefit for the company. The Prioritisation section summarises and gives insights into that issue.

3.1 Metrics

Object oriented metrics [5] can be used to analyse various characteristics of a system. Unfortunately, in a stand-alone form their value is often limited as a means for architectural decision-making. Questions like what values are acceptable for this system, how can they be combined to add more value and what metrics to emphasize usually arises.

To give true insight you need to see the metrics' dynamical nature and the value of combining them to form a more high-level view of the system. You are interested in how the characteristics of your system changes over time. In this way you can easily see which aspects of the system that are trending negatively and consequently where countermeasures are needed - a dynamic view.

Another issue is the particular metric's value in a reengineering project. Our experience is that most of the symptoms you search for cannot be found by the classical metrics. Instead, knowledge of the system must be combined with various measurements to produce valuable reports. A tool like the XRadar can help you with that.

We worked with many different views of the underlying metrics, but ended up with combining some selected metrics to get at picture of the amount of redundant code, illegal dependencies and code rot.

3.2 Redundant Code

Problem As time goes, old code tends to be forgotten. When changes are done developers forget to (or are not given any time to) clean up the code. Code may be inaccessible or simply never used. This redundant code decreases the maintainability of the system. Our experience with redundant code is that the readability decreases, build cycles go slower and the impression of the system is more complex than what it inherently is. The last issue is important when considering system flexibility. Time is spent analysing regression effects and testing code that is worthless for the business.

Removal of redundant code is one of the easy wins in a reengineering project. It is among the tasks with the lowest risk that you can do in the code base.

Solution We use two techniques to find redundant code: static analysis and statistics from the production environment showing the usage of public access points into the system. When you start removing public entry points, new rounds of static code analysis will most likely report large amounts of internal unused code that can be removed, and so goes the cycle: remove, analyse, remove, analyse. As this process goes on, the code base diminishes and becomes more manageable.

Static analysis of the code base can give valuable insights into the inaccessible parts of the code such as private methods, fields and inner classes that are never accessed. In a similar manner, you can also statically find the unused classes in the system.

Access points typically refer to the publicly available component services of the system. The largest part of redundant code in the system can be found by instrumenting all public access points and detecting which are never used in production (no client applications calling the methods). For a mature system not being maintained with dedication to the redundant parts, the system will most likely have a considerable amount of unused code that may be removed.

Being Cynical Code that is detected as unused through static code analysis may be removed immediately, but some care should be taken to avoid deleting classes that are only accessed through mechanisms such as reflection in Java. When the unused entry points are detected these should be deprecated and allowed to remain in the system for enough time for the clients to report in wrong deprecations (e.g. methods that are used very seldom, but still are important). In this way you will avoid deleting skin-dead code. Still, our experience is that in many cases you need to be ruthless. Clients often report that they have references to the code that you deprecate - references that more often than not originates in client code that also is unused and redundant. Pushing and involving them a little extra on code removal may often create gains for both.

3.3 Illegal Dependencies

Problem Illegal dependencies are dependencies that cause unnecessary entangling of the parts of your system. It is obvious that since there are some dependencies that are "illegal" there must also be some that are "legal". The definition of which are what arises from a defined "dependency graph".

A dependency graph contains a set of sub-systems (that make up your system) and the legal dependencies between these (making up the edges in the graph). For each sub-system it says what code belongs to it (e.g. which java-packages comprises a specific sub-system). The dependency graph should be a directed acyclic graph.

Sources for defining this structure might be found in architecture documents and by doing interviews with system domain experts. Once the work is done you actually have a declared and measurable definition of the vertical and horizontal layering of the system. Based on such a graph it is easy to see whether for example the integration layer framework makes use of customer functionality (which probably is not desirable).

A particularly unwanted problem area are cyclic dependencies [3]. Cyclic dependencies in a system appear when one module calls another module which again directly or indirectly calls the first module. This is a major pain that results in a plethora of problems that all directly negatively affects the objectives of a reengineering project.

The immediate result of these cyclic dependencies is code entangling. Different subsystems with different responsibility depend on each other. This greatly increases the risk for regression errors, and the burden of development and testing changes become large. Hence, flexibility to change and the number of critical errors suffer.

The second problem with cyclic dependencies is that you have to build and deploy all components for each release. It is not possible to develop, build and deploy a single module, and bug fixes are risky since they need to be patched into the system. In sum, such dependencies hinder incremental builds and deployment into production. The end result of that is slow time to market of prioritised functionality.

Solution The advantages gained from removing illegal dependencies between subsystems are numerous, but as in many other circumstances in life; with great gain comes high risk. Mature systems are not likely to have automated acceptance tests that you can use for regression testing. Unit tests are probably not used either. Going into details in this area is not the scope of the paper, but having a clear strategy for both of these testing elements are essential before one starts to do massive restructuring of the code base.

Once the testing strategy has been developed and proved to work for the system, one can start removing the illegal dependencies. There are several ways to remove illegal dependencies and these vary greatly with the nature of the system. Still, some are probably typical:

- The first thing to be done with the illegal dependencies should simply be to optimise the imports in the system. Redundant imports are not part of the code base, but are still references. Several tools exist that can automate that process.
- Second, move methods, classes and modules that are placed in the wrong subsystem. Moves like these often give major results. Of course, one must give attention to clients and interface changes before moving public API related modules.
- Third, standard refactoring and redesign techniques must be used to remove the last illegal dependencies. This part is naturally the most risky, but may result in major gains for the system such as reusable frameworks.

Being Cynical When working on the third and most complex part, our advice is to start at the most fundamental and risky element. Having that part separated out gives confidence, proof of the theory and will probably give the greatest gains for the organisation. The rest of the subsystems will then be much easier to separate out. The XRadar may continually give you the complete picture of how the system compares to the legal dependency graph. When you are done the path is clear to implement an incremental build configuration and you have a highly more flexible and maintainable code base.

3.4 Code Rot

Problem Developers often associate complexity with spaghetti code. Spaghetti code comes in many forms but is often caused by illegal dependencies. Hence, to be precise we define code rot as code with bad smells [6] caused by high complexity. Attacking this problem has been an extensive area of research, and today refactoring [6] is a well accepted practice. In light of that, we focus here on practices for discovering the most immediate problems in a large code base.

Solution Identifying code rot is a task that depends on the nature of the system. Still, some problems are universal, two of these being high method complexity and duplicated (copied and pasted) code. We present our ways of identifying these, all supported by the XRadar.

Here we define a method complex if it has a high McCabe metric for Cyclomatic Complexity [5]. This metric simply is a count of the number of paths a call to a method can go. One typically counts each conditional in a method and out comes the magic number. Usually one sets the threshold on this to 10. All methods with a value above 10 should in time be refactored. The reason to this is that such code has reduced readability and testability. The result is less maintainable code. The metric can be found by static analysis of the code base.

Copying and slightly modifying code are results of quick wins for the short term. Such an act results in duplicated code [2]. For maintenance such code becomes a nightmare, especially if the copying and pasting continues on the originally copied code.

In our experience measuring copied and pasted code can be done in two ways:

- There exists several free tools that can statically analyse the code and find equal sequences of code in the same code base. Such an approach gives fast discoveries.
- The other method is to group all methods in the code base according to the method names and cyclomatic complexity. This measurement is a little more error prone compared to the previous but is often very effective. The theory is that methods with similar names and cyclomatic complexity have been copied. It has been our experience that this is usually true for methods with a CC larger than 10.

Being Cynical After having done analysis of the code, you want to make a choice on which parts to focus on. One typically makes a list of code that is overly conditionally complex and/or copied and pasted. Maybe another code measurement has been crucial for your system, and violating code to this measurement has also been added to the list. What should you start with?

Prioritise the refactoring based on a very important dimension: Historical activity on the code base. Luckily, this measurement is easily obtainable from the source control system. Most of these systems have an API that one can program against to obtain the maintenance metrics on the code files. If, for instance, you have a list of classes in your system where code rot is a critical, prioritise them according to how much source control activity these have been involved in. The assumption is that the most historically maintained classes will likely continue to be maintained. Hence, the greatest benefit is achieved if you refactor and write unit tests on these classes first.

3.5 Prioritisation

When doing a reengineering project, your success is measured. Hence, even though your heart tells you to attack a certain problem you discover, it is not necessarily the right thing to do compared to all the other activities that are lined up. The philosophy should be to attack the problems that give the greatest benefits to the company on both the long and short term. We have used the following approach with success:

- It is strongly recommended to start by deleting obsolete code since the results are easy to obtain and it reduces the amount of code for the rest of the tasks in the project.
- After that, attack the illegal dependencies. You will not be able to save the typical mature and neglected system without having focus on this aspect.
- Last, pure code rot removal should be focused on. The risk will be minimized when you have a clearly defined and followed dependency graph. Furthermore, such work takes an immense effort to produce small results in a large system. From a project perspective, code-refactoring work usually does not defend the investment. The proposal is not to neglect code rot. Analyse

the problems as suggested above and produce a prioritised list of the problems. The list should be attacked as part of continual maintenance after the reengineering project finishes.

Lack of dedicated maintenance was our reason why the system needed to be reengineered. We believe no reengineering projects will succeed if this knowledge is neglected by the organisation. It is time the system starts to continually pay back its technical debt.

4 XRadar

The XRadar is a batch processing application developed as a response to the Pareto project's needs. It gets results from more 8 open source projects and a couple of in house grown projects and presents the results as massive unified html/svg reports. The architecture is based on java, xml and xsl. Presently it only supports Java, but there are plans to produce plug ins for other leading languages. [7] presents the XRadar in detail.

Although developed as a corporate tool, it has been decided to make it open source (at the time of writing, the URL has not been established, but it will be given during the paper presentation). It heavily relies on open source products, and in this way we can give something back.

Measurements As default, the XRadar gives measurements on standard software metrics such as package metrics and dependencies, code size and complexity, coding violations and code-style violations.

Data from unit test metrics and code coverage are also integrated, but must be obtained running the test suites on the system while doing monitoring. We have also integrated even more measurements such as from source control, performance metrics and SQL procedures. Similar plug ins will be made available for the public once we have produced a common generic API.

Reports The XRadar is available in two forms. XRadar Statics gives reports on the current build of the system. The other form, XRadar Dynamics, includes the time dimension and views the historical and present versions along the time axis. The Dynamics version relies on a set of two or more Statics runs of different system releases to work properly.

5 Conclusion

Based on our experience with reengineering a large component based system, we have presented a method for attacking such systems in general. The method is based on software metrics and quantitative measurements. The various symptoms and approaches for their resolution have been discussed in relation to the typical objectives of a reengineering project - objectives with a mandate in a

business case. The methods goal is to get the old system technically ready for typical agile practices. The results of using the method have been very good. A future paper will present the hard results.

As a means of performing such reengineering work we have presented the community a new open source tool, the XRadar, that has been developed in parallel with the methodology. The tool is based solely on other open source projects. It can be used to measure all the various symptoms on a system as discussed.

Our viewpoint here has only had focus on the system challenges. When adapting a system to agile practices it is just as important to focus on the human side, methodology and process. The organization and process involved in maintenance of such a system needs a major shift in order to take out the potential resulting from reengineering. One does not want the system to go back to its old sins.

References

1. XRadar http://xradar.sourceforge.net/
2. Hunt, A., Thomas, D.: Pragmatic Programmer, from Journeyman to Master (2000)
3. Martin, Robert C.: Agile software development, Principles, Patterns, and Practices (2002)
4. Beck, Kent: Extreme Programming Explained: Embrace Change (1999)
5. Fenton, Norman E., Pfleeger, Shari L. : Software Metrics: A Rigorous and Practical Approach, Revised. (1998)
6. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code (1999)
7. Kvam, K., Lie, R., Bakkelund, D.: A Tool for Cynical Reengineering (2004) Presented at the rOOts 2004 conference