Improving Code Maintainability: A Case Study on the Impact of Refactoring

Michael Wahler, Uwe Drofenik
ABB Corporate Research
Baden-Dättwil, Switzerland
firstname.lastname@ch.abb.com

Will Snipes
ABB Corporate Research
Raleigh, NC, USA
firstname.lastname@us.abb.com

***Abstract*—It is a fact that a lot of software is written by people without a formal education in software engineering. As an example, material scientists often capture their knowledge in the form of simulation software that contains sophisticated algorithms representing complex physical concepts. Since software engineering is typically not a core skill of these scientists, there is a risk that their software becomes unmaintainable once it reaches a substantial size or structural complexity.**

This paper reports on a case study in which software engineers consulted magnetics researchers in refactoring their simulation software. This software had grown to 30 kloc of Java and was considered unmaintainable by the stakeholders of the research project. The case study describes the process of refactoring a system under the guidance of a software engineer with results supported by static analysis and software metrics. It shows how software engineers evaluated and selected refactorings to apply to the system using their expert judgment with input from static analysis tools and discusses the outcome of refactoring as evaluated by code owners and reported via static analysis metrics.

I. Introduction

There is a growing amount of software that is written by people without specific training in software engineering: managers write Excel macros in VBA, scientists write simulations in Python, and web designers create interactive sites with Javascript. Modern programming languages and tools enable developers with no formal software engineering training to create runnable programs in a short time [1]. Many programs that are created this way have a short life cycle, while some of these programs survive for a decade or longer.

These long-lived programs require maintenance during their life time because they are often extended, reused, or ported onto new platforms [2]. We have observed that programs written by software engineering laymen are often hard to maintain once they reach a critical size and complexity. Low maintainability results from a lack of separation of concerns, the use of common anti-patterns (e.g., monolithic classes), or code duplication [3], [4].

In order to make such programs maintainable, it may be required to refactor them under the supervision of an experienced software engineer. This engineer needs to decide on a refactoring strategy: which parts of the code should be refactored, what is the starting point of the refactoring, and how should the success of the refactoring activity be measured?

In this paper, we report on a case study in which software engineers were consulted to help to refactor DID, a physics simulation program that is continuously extended as part of a research project. DID is the result of several person years of effort and amounts to 30 kloc of Java. With growing size and complexity of its data structures over the many years of its development, enhancing DID had become too costly for the available resources.

In order to measure the results of the refactoring process, we decided to use a combination of static analysis and software metrics as an objective measure of software maintainability. At the same time, we relate the results of these measurements to the developer's subjective impression of maintainability.

The contributions of this paper are

- a comparison of subjective code analysis by a developer with the results from a static analysis tool;
- a guideline how these results help to prioritize tasks in the software refactoring process; and
- results and lessons learned of using the above approach in a case study.

This paper is structured as follows. Section II presents our approach to assessing the maintainability of a software project. Section III introduces the case study and its challenges in maintainability. Section IV shows how the results of such an assessment can guide developers through refactoring activities. Section V presents results and lessons learned in this project. Section VI presents related work and Section VII concludes this paper.

II. Assessing Maintainability

Maintainability is "the ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment" [5]. Maintainability can be measured as the developer's effort for making changes to a software program [6]. When the effort to make changes in the software increases, the maintainability rating for software correspondingly decreases. In this case study the maintainability of DID was considered very low as developers began to avoid making changes to the program.

It is typically easy for experienced software engineers to detect low maintainability in a given piece of software. What can be challenging is to pinpoint the exact reasons for low maintainability. This, however, is important for defining a refactoring strategy for the software.

In order to reduce the dependency on the subjective impression of the software engineer and obtain a systematic foundation for further activities, objective criteria are needed to assess the maintainability of a code base. We discuss two approaches for obtaining such criteria.

A. Subjective Assessment

Developers can identify code that has low maintainability by using their expert judgment. In this case the developers refactoring the code are experts in the code base and familiar with the maintainability problems that have been experienced (see Section III). The developers evaluated the code base and identified areas of the code that required refactoring. They created a prioritized list of issues to address in the code. The high-priority issues were architectural in nature requiring significant changes to code structure. We discuss the improved architecture and resulting maintainability improvements in Section V.

B. Automatic Analysis

For the quantitative analysis we chose to apply static analysis tools to the code base. We used both tools engineered to find defects and tools that generate metrics of the code. The Findbugs[1] tool was run on code prior to refactoring to identify potential defects in the code base [7]. Then Findbugs was run after refactoring and the results compared. We compared the warnings pre-refactoring to post-refactoring to determine the change in the number of warnings for each module.

Another analysis tool, PMD detected all instances of code clones in DID. We analyzed the number of duplicates and number of duplicated lines before and after refactoring.

III. The DID Case Study

We use the DID software as a case study throughout this paper because it is a common example of a software project that starts very small, but grows fast and becomes difficult to maintain. The development of DID was started by our co-author Uwe in 2011. Initially, DID was a small tool for designing magnetic components to optimize a specific transformer design without any graphical user interface (GUI). Back then, the DID software was not an official deliverable of the project. In subsequent magnetics research projects, newly created magnetic component models were implemented in DID, and collaborating researchers started to employ DID unofficially.

During its lifetime, DID has grown to around 30 kloc of Java code. The code is organized in seven packages and 61 classes. The DID software comprises a GUI, code to read or write persistent data from/to files, transformations to different output formats, and the actual simulation functions. The DID developers use NetBeans 8 as their IDE with the Java 1.7 libraries.

[1]http://findbugs.sourceforge.net/

DID was continuously optimized for calculation speed and accuracy, but it was not structured at all with respect to software architecture. As a result, around 2014, adding new components became more and more demanding, and later that year adding new components was stopped. At the same time more and more researchers employed DID in their projects to help them design the already-implemented components.

Transfer from the research department to the business units was desired, but proved impossible because of concerns about missing code documentation, quality of graphical user interface (GUI), missing versioning, difficult structure, and maintainability of the code. Therefore, a collaboration between software engineers (co-authors Michael and Will) and Uwe (co-author of this paper and original author of DID) was initiated in 2015.

At the start of this collaboration, the software engineers wanted to know what contributed to this perception of low maintainability. To this end, we asked Uwe to extend DID with a nontrivial case and report on the obstacles of this task (Section III-A). In parallel, in order to get an objective view of the maintainability and the general quality of the code base, we ran automatic analyses and interpreted the results (Section III-B). The goal of both activities is to eventually pinpoint the reasons for the assumed low maintainability and define the starting point for refactoring activities.

A. Subjective Assessment of DID

For the subjective analysis, Uwe carried out several maintenance tasks on the initial version of DID. In this subsection he reports on the results.

1) Adding a New Magnetic Component: Adding a new magnetic component requires at least 59 changes in 17 already existing classes, in most cases adding already existing code by copy/paste, partly with minor changes. A class `Calculator` with 850 lines of code, originally designed for managing the component optimization algorithm (300 lines), also contains GUI-related code (550 lines) for text formatting, and requires changes and/or additions in 10 different locations of the code. Additionally, two other classes for GUI formatting require changes at 10 and 15 different locations, respectively. With the existing software structure, adding a new magnetic component does not result in adding a new class, but forces already existing classes to grow.

As another example, model improvements and/or bug fixing (e.g., in the model of the winding geometry of the component) requires changes in typically 7 classes, with up to 12 nearly identical code blocks (about 40-50 lines of code per block) to be changed for each of the 7 classes.

From the originally 7 implemented magnetic components, three have become obsolete (for practical reasons concerning manufacturability of the components), but because their code showed up at so many different locations, the corresponding code and/or magnetic component implementation could not be deleted. Although there was a clear desire from the engineers to employ the tool extensively, adding new magnetic

TABLE I: Analysis results of the initial DID version

Tool	Issues reported
FindBugs	2853 potential issues
PMD Copy/Paste Detector	373 duplicates, 7763 lines

components to these four valid ones became so difficult and error-prone that developers avoided doing so.

Another problem with adding components was that the physical component parts were not simplified which resulted in a large number of possible combinations on the sub-component-level, directly resulting in a large amount of redundant code.

For example, the developers originally implemented four types of windings configurations, with each winding configuration representing the assembly of one coil. With e.g., two coils per component there are four different coil sets with identical winding configurations implemented in the code. At this point, any changes in the winding configuration required 22 changes in 14 files. Later, an attempt to add another three configurations required so many changes that the implementation failed. To address this requirement, it was decided to implement an option to employ a post-processing algorithm, which modified the result in order to estimate the effect of the different winding. This post-processing algorithm added significant complexity to the code.

2) Electrical Excitation and Loss Calculation: Excitation of the magnetic component can be defined via ideal voltage- and/or current-waveforms, or via stepwise numerically simulated voltage- and/or current-waveforms. The waveforms are imported via the GUI (2050 lines of code alone), but part of the pre-processing of the waveforms is already performed inside the GUI adding another 600 lines of code. The remaining part of the pre-processing is spread over three classes (830, 540 and 620 lines of code) with some redundancies. The pre-processed excitation is accessed from 63 code lines distributed over 15 classes.

The main challenge is that the code structure of the pre-processing grew over time with different requirements from different design projects, and did not follow any model-related logic. Adding certain pre-processing would be highly desirable, but has not been tried due to the complexity of the already existing code.

3) Goals of the Refactoring: The goal of the intended code refactoring is to create modular and maintainable code, which allows for quick and easy creation of additional magnetic components whenever required. All mathematical models and calculation algorithms should remain unchanged and ideally copied in large parts into the new class structure, which is expected to be quick and simple.

B. Automatic Analysis

For a quantitative statement of the maintainability of DID, we ran several analyses that the NetBeans IDE offers either built-in or in the form of downloadable plug-ins. Table I shows the analysis results of the original DID version.

The numbers shown in Table I appear intimidating. There are roughly 3'000 issues in 30 kloc; does that meant that there is an issue in every 10^{th} line of code? Where shall we start fixing the code given such a high number of reported issues?

First of all, the issues that are reported by the tools are *potential* issues. As an example, FindBugs[2] reports any dereferentiation that may potentially be null, which causes a runtime exception if not caught. In typical use cases for the software, the object reference in question may never be null. It goes without saying that production-quality code should cater for as many use cases as possible and thus, take into account erroneous input, check whether references are null, and properly handle cases in which they are. For research prototypes such as DID there is typically no budget to elevate the software to production quality. As a result, the software may crash occasionally, but this is not a big deal.

Second, FindBugs groups its reported issues into several categories. Based on the requirements of the software projects, some of these categories can be excluded. For DID, FindBugs reports 188 performance issues, which we decide to ignore because the performance of DID is generally considered as good. Also, FindBugs reports 27 internationalization issues, which we also ignore because DID will only be used by English-speaking users. 1'157 issues found are related to missing JavaDoc annotations, which may have a negative influence on the maintainability of the software.

The issues reported by PMD Copy/Paste Detector[3] have to be taken very seriously. It is known that duplicate code has a negative impact on the maintainability of software because it is easy to introduce inconsistencies if changes in a code snippet are not applied to all of its clones [8].

IV. Refactoring Strategy

We have shown that the original version of DID exposes multiple deficiencies in terms of maintainability. With the numerous warnings from different automatic analyses and the subjective assessment of the developer, we need to devise a refactoring strategy that improves the maintainability of the code and prioritizes refactoring actions such that we achieve the best outcome with the least effort.

Together with the project stakeholders we have devised a list of tasks and assigned priorities to them. In the following, we present a list of tasks, grouped by priority into subsections. Adding missing JavaDoc annotations was considered an orthogonal task to be carried out while working on the other tasks.

[2]Netbeans FindBugs Integration plug-in 1.32

[3]http://plugins.netbeans.org/plugin/1529/pmd-s-copy-paste-detector

A. High-priority Tasks

1) Separate Concerns: We have seen that the initial version of DID often mixes several concerns in the same place. As an example, there are classes that perform file I/O operations, provide parts of the GUI, and contain algorithms that operate on the data model. This is a huge impediment for maintainability: mixing concerns requires developers to change the code in many places when only one of the concerns is changed. In contrast, separating concerns reduces the side effects of changing one piece of code and allows developers to change one aspect of the software (e.g., the GUI) without interfering with other parts.

2) Define Data Model: In DID, there is a coarse-grained data model, which defines important domain concepts such as *transformer* or *core material* as classes. The main drawback of such coarse-grained model is that each class hosts a variety of concepts that are not explicitly modeled such as insulation, wire material, or windings. As a result, many classes are fairly large and complex, which makes them difficult to maintain. Therefore, we decided that defining a fine-grained data model in which all physical concepts are represented by proper classes has high priority. We also pushed for a data model that is agnostic of GUI and file I/O, striving for a model-view-controller architecture.

3) Introduce Multi-Level Testing: Testing in the initial DID version was carried out through a small set of system tests that need to be run manually. We assigned a high priority to the creation of a solid set of multi-level tests (unit tests, integration tests, system tests) for two reasons. First, automatic tests enable regression testing of the software during our refactoring activities. Thus, bugs that are introduced due to the applied refactorings can be found automatically and fixed early. Second, the availability of automatic tests help developers in the maintenance phase of the software: bugs can be found early, and software tests can represent an important part of the documentation.

B. Medium-Priority Tasks

1) Eliminate Duplicate Code: Duplicate code was indicated by the DID developers as a challenge for maintainability, and our automatic analyses showed multiple occurrences of duplicate code. We found different patterns in the duplicate code:

- Code was copied between similar concepts. As an example, the initial DID version specifies several types of transformers as subclasses of the generic transformer class. However, a lot of code that is common to these subclasses was not moved to the transformer class, but instead copied for each subclass. In our refactoring activities we focused on introducing proper OO design using inheritance and polymorphism to eliminate most of the duplicate code.
- The original DID version has several algorithms that each provide conditional control flow depending on the concrete type of the input object. However, many branches in the code flow share code that is simply copied from one branch to another. In our refactoring activities we handled this issue by removing these conditional flows altogether, see Subsection IV-C1.

As an example, this code with duplicate lines

```
class C1 extends C {
  public double f () {
    double x = 2.5 * this.a; // duplicate
    return x + this.b;
  }
}

class C2 extends C {
  public double f () {
    double x = 2.5 * this.a; // duplicate
    return x + 0.5 * this.b;
  }
}
```

can be refactored into

```
class C {
  public double f0 () {
    return 2.5 * this.a;
  }
}
class C1 extends C {
  public double f () {
    return f0 + this.b;
  }
}

class C2 extends C {
  public double f () {
    return f0 + 0.5 * this.b;
  }
}
```

2) Properly Represent Physical Constants: DID uses many domain-specific physical constants, e.g., the density of core materials. It also uses constant arrays to represent values that were precomputed with other tools such as Matlab. Some constants are defined within the code (e.g., as static constant fields), other constants (or sets of material properties) are defined in ASCII files that are loaded at startup.

The definitions of these constants pose several problems with respect to maintainability. First, some classes have become very large because they contain too many methods and fields. This class of problems was reported by the Source Code Metrics plugin, which defines thresholds of how large classes should be at most. Second, there is a complicated mechanism for loading ASCII files with constant values and representing them as types in the code (e.g., a material defined in an ASCII file has a unique name and a unique integer type ID). Third, changing precomputed values that are stored as constants in the code requires recompilation of the whole project when the values need to be updated.

To solve these problems we took the following actions. First, we limited the usage of constants in the code to those that a) will not change and b) will not be extended. As an example, we defined a class to represent wire materials, which provides the details of the possible materials in a static map.

```
public abstract class WireMaterial {
   private static final Map<String,
   WireMaterial> ALL_MATERIALS = new HashMap();

   static {
      ALL_MATERIALS.put("Cu",
         new WireMaterial("Cu",
            0.0039, // alpha
            5.81e7, // sigma293
            380,    // k
            8940,   // rho
            1.0)    // cost
      {
      });
   // aluminium, etc.
   }
};
```

Precomputed values or physical properties that can change over the lifetime of the software (e.g., new insulation materials) will be stored in ASCII files and dynamically loaded at startup. This allows developers to extend the software by providing new definition files, leaving the code untouched.

As an additional measure, we moved *public static* fields and methods that cluttered large classes to auxiliary classes, grouped by their coherence (e.g., all fields and methods related to thermal computations are moved to the same class).

3) Eliminate Platform Dependencies: Despite Java being generally considered as a platform-independent programming language, there are some pitfalls in practice. For DID we experienced several problems when we first tried to execute it on a different machine:

1) DID uses hard-coded file names with a mix of Windows-specific and Unix-specific path separators ("/" and "\"), which caused problems; and
2) DID uses relative path names, assuming that the working path of the program is a specific directory. As a result, some constant definitions could not be found when DID is started from the wrong directory.

For the former problem, we replaced any occurrences of slashes or backslashes with the platform-independent constant `File.separator` offered by Java. For the latter problem, we used the System library to determine the working path (`System.getProperty("user.dir")`) and construct absolute filenames based on this path.

C. Low-Priority Tasks

1) Refactor Anti-Patterns: DID exposes a few anti-patterns that were neither reported by the developers nor the automatic code analysis. In contrast, these patterns are usually detected by experienced software engineers, although they could also be automatically defined by the right static analysis rules.

A commonly used anti-pattern in DID is a `switch` statement with type-dependent cases as in this example.

```
switch (transfomer.TYPE_ID) {
 case Transformer.TYPE_1:
   do_x ();
   do_y ();
   break;
 case Transformer.TYPE_2:
   do_y ();
   do_z ();
   break;
}
```

We observe two problems here: first, the `Transformer` class defines its own type system instead of relying on the type system of Java and using subtyping. Second, there is duplicate code in some or all of the `case` statements.

As a solution to this problem, we model the different transformer types as subclasses of Transformer, eliminate the type ID, and move the calculations that are carried out in the above switch construct to a polymorphic method of class Transformer which is overridden accordingly in its subclasses.

D. Improve Field/Method Visibility

Another observation made by the automatic analyses was that the visibility of many class fields and methods in DID `public`, which is generally considered bad design because it opens the door for inadvertent or malicious misuse of the code. In order to reduce the attack surface, we restricted the visibility of fields and methods to a minimum (using the `private` and `protected` modifiers).

V. Results

This section summarizes the results and lessons learned from our refactoring activities for DID. We first present a subjective assessment of the refactored version, followed by the results from automatic analyses and an account of the lessons learned during this exercise.

A. Status of the Project

At the time of writing, the new data model of DID was complete and the computation and simulation algorithms from the initial version of DID were transferred and adapted to the new data model. Extensive test cases were written to ensure that the refactoring did not introduce any logical errors or bugs.

The GUI and export-to-file mechanisms of DID were not implemented at the time of writing. These tasks were considered low priority to ensure that enough effort is spent on a data model with clearly separated concerns.

B. Subjective Assessment of the Refactored DID

Although the new version DID is not yet complete, it is already obvious that adding additional magnetic components is surprisingly easy, both in terms of time effort and code simplicity. While in the original DID it became difficult to maintain just four magnetic components, it required no extra effort to implement 10 components in the refactored DID within a couple of days.

Another example is the cooling model of the magnetic component, which in the initial version of DID was extremely complex, very difficult to use (confusing GUI, no documentation), and, therefore did not support the implementation of additional different important cooling methods. In the restructured DID currently eight practically highly relevant

TABLE II: Subjective assessment of maintainability

Adding a ...	**Initial version**	**Refactored version**	**Improvement**
new magnetic component	6–8 weeks	1–2 weeks	75 %
new cooling method	8–12 weeks	1–2 weeks	83 %

TABLE III: Analysis results of the initial and refactored DID version (*see Section V-C for detailed explanation)

Tool	**Initial version***	**Refactored version***	**Improvement***
FindBugs	1450 potential issues	374 potential issues	23 %
PMD Copy/Paste Detector	244 code duplicates, 5647 lines	17 code duplicates, 338 lines	82 %

cooling methods have already been implemented without too much effort. Implementing the *single* general thermal model in the initial DID version took two months. Implementing the *eight* new thermal models in the new version of DID will take about the same time, and adding additional thermal models afterwards should be a minor effort.

While the initial version of DID has been very useful, but limited mainly to the R&D department, the refactored DID will be transferrable to business units because the specific magnetic components (including the specific insulation and cooling methods) of their business interests can be implemented quickly.

Table II summarizes the developer's subjective assessment of the maintainability of the initial and the refactored version of DID. To this end, it compares the estimated effort for two typical maintenance tasks in each version. It can be seen that the time required for these tasks in the refactored version is substantially lower.

C. Automatic Analysis

For an objective assessment of the improvements that we achieved during the refactoring we re-ran the automatic analyses on the refactored code. Since at the time of writing, the refactored code did not comprise all the features of the original version yet, we adapted the code bases such that a valid comparison is possible:

- The code for GUI and file I/O was removed from the initial version of DID and the analyses were re-run on the stripped code.
- The documentation of the refactored version of DID is ongoing. Therefore, we removed all warnings about missing JavaDoc from *both* versions of DID.

The resulting code base comprised 15 kloc for the initial version and 5 kloc for the refactored version. Table III shows the analysis results for both versions in absolute numbers and an improvement value in percent, which is normalized over the code volume according to this formula:

$$\text{Improvement} = \frac{\frac{issues(v1)}{kloc(v1)} - \frac{issues(v2)}{kloc(v2)}}{\frac{issues(v1)}{kloc(v1)}}$$

Regarding FindBugs, we have achieved an improvement of 23 %. The majority of issues that are still reported concern serialization warnings (55), which is caused by the fact several classes implement the interface `java.io.Serializable`, but the implementation is not complete yet at the time of writing. As an example, many classes do not yet define a `serialVersionUID`. There are also 60 warnings that fields can be final, which we intend to fix once the data model has proven stable enough. Another 21 warnings relate to the fact that field names do not start with a lower case latter, which is usually caused by fields corresponding to physical units such as `T` for temperature.

In terms of code duplication, 82 % of duplicate lines were eliminated, which corresponds to around 30 % of the total code for the data model and calculations. This is a big improvement for software maintenance because developers need to take care of 5'000 lines less in the refactored version of DID.

D. Lessons Learned

1) Data models should be extensible: Magnetic components are physical objects, which are composed of a small number of components: magnetic core, winding, insulation. These might differ in material properties and geometry, but, generally, every magnetic component can be built employing just these three basic building blocks. The refactored DID version maps all these physical components into classes which results in a set of building blocks which, equal to reality, allow one to assemble any kind of magnetic component as a mathematical model. It seems obvious to structure the modeling software exactly like this from the very beginning, but it is often not the case if the software starts with a design project focusing on a single magnetic component design. Later, with a partly different project team in another project focusing on a different component design, the code structure is not refactored from the single case, and copies are made of the original code to support the new component. This pattern then continues through the evolution of the code base as new components are required.

2) Little consultation helps a lot: The restructuring effort is performed by the original developer who works on the topic of magnetic design, but is closely monitored (in regular short meetings and by employing tools) by software engineering experts. The software engineering experts spent an average of around three hours per week during three months on consulting the magnetics researchers. This time was spent on analyzing the structure of the source code, making suggestions for the software architecture, and answering the questions of the developer. The actual refactorings were carried out independently by the developers in absence of the SE experts. This shows that significant improvements in the maintainability of software projects can be achieved with little effort by SE experts. As a consequence, we suggest that SE experts accompany even small development projects by non-experts from the beginning.

3) SE experts must understand the domain: The data model of DID comprises many classes organized in a complex class model. This is due to the fact that the physical model is very complex, comprising a magnetic core in different geometries and materials, windings, wire, insulation material etc. In order to make sound suggestions for the organization of the data model, the SE experts needed to obtain a basic understanding of the domain. As an example, when devising a data model, one needs to take into account that it is unlikely that new wire types (currently, *litz* and *solid*) will be used, whereas it is likely that new core geometries will be designed. This has an impact on the tradeoff between simplicity and flexibility of the data model.

4) Automatic analyses should be used early: We have seen that automatic code analysis helps to identify many issues. Therefore, we suggest that automatic code analysis be used from the beginning in even in small software development projects to detect and fix issues earlier. One impediment is that the results reported by most tools often need interpretation and priorization by an SE expert. It would be helpful for non-SE-experts to obtain analysis results in a language they can easily understand, together with suggestions on how to fix them.

VI. Related Work

We understand from Parnas that aging as a result software evolution leads to low maintainability of software. Parnas discusses causes and factors to consider about software aging and surveys potential areas to investigate and ways to address software aging problem. Relevant ideas are that modifications to software over time are a cause of decreased maintainability and reliability [2].

The decay of software maintainability over time is described in two laws of software evolution by Lehman [9]. The law of increasing complexity anticipates that the system's inherent complexity will increase as changes are made to it unless specific effort is expended to refactor the code. The law of declining quality indicates the system will become more defect prone as further changes are attempted to it. Both these laws are relevant to this software project's evolution path.

Measuring and drawing inferences from maintainability of software has been a research topic for many years. The commonly used Maintainability Index was defined by Coleman et al. in the early 1990's as a model combining size, complexity and comment volume metrics [10].

Evaluations of metrics to assess maintainability include a focus on object oriented software metrics. Li et al. evaluate object oriented software metrics for their correlation with maintenance effort in ADA software [11]. It finds that 8 metrics are correlated providing more information than size in the population. Basili et al. find some of the Chidamber and Kemerer (CK) metrics suite are correlated with defect-proneness in a sample of eight information system application that were developed from identical requirements [12]. They found correlations with defect-proneness in five of the six C-K metrics including Weighted Methods per Class and Depth of Inheritance Tree. Tang et al. evaluate the CK metrics suite as a predictor of defect prone modules. They find that Weighted Methods for Class and Response For Class are good indicators of fault-proneness [13]. They propose additional metrics which are also related to fault-proneness that could be used for test prioritization. Dagpinar et al. create a set of object oriented metrics and validates them as relating to maintainability of object oriented software [14]. Metrics selected were Total Number Of Statements (TNOS), Non-inheritance class-method import coupling (NICMIC), Non-inheritance method-method import coupling (NIMMIC) and Non-inheritance import coupling (NIIC). Dubey et al. review the body of literature on the CK metrics suite and conclude that CK metrics are useful for predicting fault-proneness and maintainability of object-oriented software systems [15]. In this paper we assess the impact of refactoring on number of duplicated code lines and number of static analysis warnings.

More recent studies have looked at relationships between maintenance effort and measures of maintainability. Bijlsma et al. performed a study to determine what static measures of maintainability are drivers of bug or enhancement resolution speed [16]. Resolution speed is defined by the authors as the calendar time from the point the item was opened until it was closed. They found that maintainability, size and complexity are correlated with resolution speed. Sjøberg et al. conduct an experiment with four identical software systems to determine whether maintenance effort is correlated with quantitative measures of maintainability [17]. They looked at Size, Maintainability Index, Coupling, Cohesion, Weighted-Methods for Class, and Depth of Inheritance Tree. They also looked at measures of code smells of Feature Envy and God Class. Results show that system size is the most relevant indicator of maintenance effort and that metrics such as Maintainability Index are not correlated with maintenance effort. Our focus for improving maintenance effort included reducing the number of duplicated code lines as well as providing a better structure for the code.

Studies on refactoring have focused on quantifying the impact of refactoring on measurements of software structure. For example, Sangal et al. describe a Dependency Structure Matrix and provide a case study applying a new tool for DSM creation to Haystack, an information retrieval system [18]. They apply the DSM to restructure the Haystack code removing undesired dependencies from the code structure. The refactoring results in a cleaner software architecture and architecture rules that prevent degrading the architecture in future updates. Du Bois et al. study the impact of refactoring on coupling and cohesion measurements of source code. They assessed the change in coupling and cohesion measurements when specific refactoring methods were applied [19]. They recommend applying Replace Method with Method Object, Replace Data Value with Object and Extract Class refactoring methods to improve cohesion. Then applying the move method to localize dependencies and move method to separate concerns refactorings to improve coupling. Their case study showed that applying these refactoring methods indeed improved coupling and cohesion measurements in the Apache Tomcat[4] open source system. Kataoka et al. also study the impact of refactoring on coupling metrics and find that coupling measures effectively quantify the refactoring effect [20]. In this paper we evaluate the results of refactoring on reducing duplicate lines of code and reducing static analysis warnings.

Looking specifically at the impact of refactoring on programmer effort, Wilking et al. [6] present an evaluation of refactoring in a controlled experiment. They measure maintainability as the time required to fix randomly seeded bugs in the code base. The find that refactoring increases the maintenance effort, but decreases cyclomatic complexity and reduces the program's memory requirements. The most frequently applied refactoring methods were extract method, rename method and add comments. Moser et al. [21] find refactoring has a short-term positive impact on productivity in a limited industrial setting. Moser also reports a decrease in coupling related software metrics. In this paper we observed the highest impact of refactoring was reduced duplicated code followed by reduced static analysis warnings.

VII. Conclusions

We have presented a case study on refactoring a design tool for magnetic components in order to increase its maintainability. For prioritizing maintenance tasks, we have combined the subjective assessment of the original developer with the results from automatic code analyses. Such combined assessment was also used to measure the success of our refactoring activities.

The results are encouraging. The amount of duplicate lines of code was reduced by 82 %. This number is affirmed by the subjective assessment of the original developer, who can confirm that maintenance tasks can be carried out more easily on the refactored version of the tool than on the initial version. Furthermore, the number of potential issues found by FindBugs was reduced by 23 %. While this is already quite good, we want to investigate further the root causes of the still high number of reported issues and work towards eliminating them.

We believe that the approach presented in this paper can be used in similar refactoring projects. We encourage our colleagues to pay extra attention to the lessons learned on which we have reported in this paper.

A. Future Work

We have seen that static analysis reports a lot of potential issues, but many of these issues have low relevance for software maintainability. For this project we manually analyzed and prioritized the results. In future work it would be useful to have developers report a ranking of modules from low to high maintainability. This ranking would serve as a basis for comparing static metrics results to the developer opinion of maintainability. For future projects it would be desirable to have the static analysis results automatically ordered based on their importance for maintainability. As an example, results that hint at violations of the separation-of-concerns principle should be ranked higher than results hinting at potential performance impediments. Also, the results should be represented in a way that allows non-SE-experts to understand and fix the respective issues.

[4]http://tomcat.apache.org/

References

[1] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089733.1089734

[2] D. L. Parnas, "Software Aging," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287. [Online]. Available: http://portal.acm.org/citation.cfm?id=257788

[3] ——, "On the Criteria to Be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972. [Online]. Available: http://dx.doi.org/10.1145/361598.361623

[4] W. H. Brown, R. C. Malveau, H. W. "Skip" McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. New York, NY, USA: John Wiley &amp; Sons, Inc., 1998. [Online]. Available: http://portal.acm.org/citation.cfm?id=280487

[5] ISO/IEC, "Systems and software Quality Requirements and Evaluation (SQuaRE) ISO/IEC 25000:2014," Tech. Rep., 2014.

[6] D. Wilking, U. F. Khan, and S. Kowalewski, "An Empirical Evaluation of Refactoring," *e-Informatica Software Engineering Journal*, vol. 1, no. 1, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.2442

[7] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1145/1052883.1052895

[8] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of cloned code on software maintainability: A replicated developer study." in *WCRE*, 2013, pp. 112–121.

[9] M. M. Lehman, "Laws of software evolution revisited," in *Software Process Technology*, ser. Lecture Notes in Computer Science, C. Montangero, Ed. Berlin/Heidelberg: Springer Berlin Heidelberg, 1996, vol. 1149, ch. 12, pp. 108–124. [Online]. Available: http://dx.doi.org/10.1007/bfb0017737

[10] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994. [Online]. Available: http://dx.doi.org/10.1109/2.303623

[11] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, Nov. 1993. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(93)90077-b

[12] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1109/32.544352

[13] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*. IEEE, 1999, pp. 242–249. [Online]. Available: http://dx.doi.org/10.1109/metric.1999.809745

[14] M. Dagpinar and J. H. Jahnke, "Predicting maintainability with object-oriented metrics-an empirical comparison," in *Working Conference on Reverse Engineering (WCRE)*, 2003, pp. 155–164.

[15] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 1–7, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1145/2020976.2020983

[16] D. Bijlsma, M. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software Quality Journal*, vol. 20, no. 2, pp. 265–285, May 2012. [Online]. Available: http://dx.doi.org/10.1007/s11219-011-9140-0

[17] D. I. K. Sjøberg, B. Anda, and A. Mockus, "Questioning Software Maintenance Metrics: A Comparative Case Study," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '12. New York, NY, USA: ACM, 2012, pp. 107–110. [Online]. Available: http://dx.doi.org/10.1145/2372251.2372269

[18] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, vol. 40, no. 10. New York, NY, USA: ACM, Oct. 2005, pp. 167–176. [Online]. Available: http://dx.doi.org/10.1145/1094811.1094824

[19] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, Nov. 2004, pp. 144–151. [Online]. Available: http://dx.doi.org/10.1109/wcre.2004.33

[20] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 576–585. [Online]. Available: http://dx.doi.org/10.1109/icsm.2002.1167822

[21] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," in *Balancing Agility and Formalism in Software Engineering*, ser. Lecture Notes in Computer Science, B. Meyer, J. Nawrocki, and B. Walter, Eds. Springer Berlin Heidelberg, 2008, vol. 5082, pp. 252–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85279-7_20