Defect Frequency and Design Patterns: An Empirical Study of Industrial Code

Marek Vokáč

***Abstract*— Software "Design Patterns" seek to package proven solutions to design problems in a form that makes it possible to find, adapt and reuse them. A common claim is that a design based on properly applied patterns will have fewer defects than more ad hoc solutions.**

This case study analyzes the weekly evolution and maintenance of a large commercial product (C++, 500 000 LOC) over three years, comparing defect rates for classes that participated in selected Design Patterns to the code at large. We found that there are significant differences in defect rates among the Patterns, ranging from 63 % to 154 % of the average rate. We developed a new set of tools able to extract design pattern information at a rate 3×10^6 lines of code per hour, with relatively high precision.

Based on a qualitative analysis of the code and the nature of the Patterns, we conclude that the Observer and Singleton patterns tend to be used in complex parts, and so can serve as indicators of code that requires special attention. Conversely, code designed with the Factory pattern is less complex or less central, and consequently has lower defect rates. Template Method was used in both simple and complex situations, leading to no clear tendency.

***Index Terms*— design patterns, defects, defect frequency, industrial code, case study**

I. Introduction

SOFTWARE Design Patterns as first formalized by Gamma *et al.* [1], have become popular in the object-oriented software community. Some of the patterns have been incorporated into widely used architectures and frameworks. Examples of this are the Observer pattern in event-based user interfaces, and the Factory pattern in Microsoft COM, MFC and J2EE.

Common arguments for the use of Design Patterns often relate to defects—by designing an application with proper use of Design Patterns, we reduce the number of defects, as in [2]. Other claimed positive consequences of using Design Patterns are additional flexibility and easier understanding of the design ([3]–[5], Introductions), and reduced change proneness [6].

In this research, we wish to test the claim of reduced defects by determining the defect rates and Design Pattern usage of a large industrial software product. We performed a case study on a Customer Relationship Management (CRM) product. During the study we developed a tool for fast analysis and extraction of Patterns from C++ code. Having access to the complete history of the product for three years made it possible to analyze the defect rates and correlate them with the usage of Design Patterns. The defects come from both pre-release testing and post-release reporting by users.

The rest of the paper is organized as follows: section II describes the chosen patterns, and how they were chosen for study. Section III lists related work. Our pattern extraction tool is described in section IV, and the studied "CRM5" software product is described in section V. Extraction of patterns from CRM5 is described in section VI. Our statistical model follows in section VII, and it is interpreted in section VIII. Threats to validity are addressed in section IX, and our results are summarized in Section X. Finally, section XI concludes and outlines future work.

Marek Vokáč is with the Simula Research Laboratory, NO-1325 Lysaker, Norway

II. Design Patterns

In this section, we describe the five patterns chosen for investigation, as well as the method by which they were chosen. For each pattern, we list the features that lead us to expect different effects on defect rates.

A. The choice of patterns

The selection of Design Patterns was based on several criteria: the occurrence of pattern names in academic literature; the occurrence of pattern names in relevant Web forums; the occurrence of pattern names on the Web in general; and the structure of the patterns, to test the effects of fundamentally different patterns.

	ISI		IEEE/ACM	
Pattern	Raw	Content	Raw	Content
Composite	4	4	42	15
Observer	8	6	9	8
Factory	5	4	16	9
Decorator	3	4	4	4
Adapter	1	3	5	4
Bridge	3	0	15	5
Singleton	2	2	4	3
Visitor	1	1	4	4
Proxy	1	0	5	4
Façade	3	1	2	2
Template Method	1	1	1	1
Sum	32	26	107	59

TABLE I

Design Pattern ranks in academic literature. The "Raw" column lists the number of hits for the query, while the "Content" column lists the articles that were relevant.

Academic literature was searched via the ISI Web of Knowledge, IEEE and ACM databases, and the results are in Table I. Simply counting the hits was not enough—the results of such searches can be quite imprecise. For instance, the word "'Composite" also matches "compositing" in the IEEE search engine, leading to many false hits in the area of CASE tools and architecture papers. For the Web in general, a series of searches was done using Fast, AltaVista, Google, Yahoo, MSN

and Lycos.[1] The search criteria were "design patterns" software <pattern name> for the candidate patterns listed below. The search target was the web in general—or more precisely, the subset of the Web included by each search engine.

Pattern	Median rank	Rank st.dev	Total hits
Factory	1.0	0.41	24 124
Proxy	3.0	1.55	19 856
Composite	4.0	1.10	17 010
Bridge	5.0	1.72	16 105
Observer	5.0	0.75	15 677
Singleton	6.0	2.32	20 294
Adapter	8.0	0.52	14 484
Template Method	9.0	3.67	13 424
Visitor	9.0	2.58	13 832
Façade	10.0	0.82	9 700
Decorator	11.0	0.00	7 521

TABLE II

DESIGN PATTERN RANKS ON THE WORLD WIDE WEB

The design patterns were then ranked individually for each search engine, and the median rank calculated for each pattern. A low standard deviation for a patterns' rank means that it had similar ranks across all the sampled search engines, leading to a higher confidence in the rank value.

These surveys of the Web and of academic literature indicate that Factory, Composite, Bridge and Observer are patterns that deserve further scrutiny. We added three more patterns: Singleton, because it may be tempting to use it simply as a replacement for global variables; Template Method, because it was known to be extensively used in the code we later wished to analyze; and Decorator, because it can be a deceptively simple pattern that can hide quite complex behaviour. Bridge and Composite were later excluded as their structure is hard to recognize reliably, partly due to the problem of correctly extracting aggregations from C++ code, where a pointer may or may not be an array; other workers in the field have had similar problems [7], [8].

The patterns chosen for investigation were thus:

- Factory (creational)
- Singleton (creational)
- Observer (behavioural)
- Decorator (structural)
- Template Method (structural)

A Design Pattern as described by Gamma *et al.* [1] is a description, using prose and semi-formal diagrams, of a way to structure classes in a program. However, the ultimate expression of Design Patterns is in executable code, whether generated from a model or by hand. It follows that the observed effects, in terms of defect rates or maintainability, are related to the patterns through the code structure. Our analysis of the expected effects is therefore founded on the kind of structure that the selected patterns prescribe.

The effects of some patterns on object-oriented software metrics have been described by Huston [9]. Patterns can influence several kinds of metrics: coupling measures, inheritance depth, and method counts. Huston concludes that the introduction of Design Patterns in general does not lead to "unacceptable" metric scores (i.e., scores associated with high defect rates). This can at least be seen as not contradicting the claim that Design Patterns should lead to lower defect rates.

However, different patterns do have different effects and applicability. In the following sections we describe the five patterns chosen for the case study.

B. *Factory Method—a simple structure*

The Factory Method pattern is used in places where one of several possible subclasses (products) should be created, and the instantiating class (the client) cannot anticipate which subclass it should be. It may be that the knowledge does not belong in the client, or even that it is genuinely unavailable to the client. Instead, it is located in a Factory Method.

By definition, a Factory Method will operate on one single product superclass. It has no knowledge of or dependency on the clients, and also does not depend on the details of the products—it only needs to know of their existence and the criteria for choosing the proper product to create. The clients may be numerous, but their dependency on the Factory Method is simple and opaque. The fundamental structure of the program is not necessarily changed by using Factory Method—objects will still be instantiated, but through a Factory Method call instead of directly calling the `new` operator with a given concrete type.

While there may be situations in which there are stronger dependencies, complicated hierarchies of products and criteria to choose between them, we generally expect this pattern to lead to simple structures and relatively small amounts of code.

C. *Observer—a pattern for central structures*

Observer, often implemented as Publish-Subscribe, is a pattern that specifies how a number of objects—observers—may be informed of changes to one object, the subject. Usually the subject will be some kind of data-carrying object, and the observers will be views or processes that present or react to changes in the data. The classic Model-View-Controller pattern or one of its variants is often implemented using an Observer.

Both Observer and its related patterns (MVC, Publish-Subscribe) fundamentally influence the design of an application. If the subject is the applications' data model, and the observers are user interfaces, we not only expect the implementation of the pattern to include a large amount of code, but we also expect that code to be fairly complex.

The dependencies between a subject and its observers also tend to be fairly strong. The subject informs its observers of changes through some calling mechanism, and the observers then either extract embedded information from the message or make calls back to the subject to obtain more data. The subjects then process the data in some way and make further calls in the process. If the processing leads to changes in the subject, a further round is triggered, possibly recursively. The number of subjects is generally larger than one and determined

[1]Excite, WebCrawler and MetaCrawler were excluded because they limit the number of hits to less than about 100, so they cannot be used for this kind of search

at runtime, making it much harder to arrive at a deterministic model of how the system behaves.

D. Singleton—global access to a single instance

In many ways, Singleton is similar to a global variable. Its scope may be limited by name spaces or similar constructions, but within the scope it is accessible to all. The main responsibility of the Singleton pattern is to assure the existence of exactly one instance of the class in question.

The principles of Singleton are simple. The detailed implementation may be quite complicated, due to particulars of languages and environments. For instance, the order of construction and destruction of static objects in C++ is not always known or controllable. Dependencies between multiple singletons can also complicate matters.

Finally, the Singleton pattern is generally used for objects that should be globally accessible, implying that there are many dependent objects. A change to a Singleton will therefore tend to have a large impact. On the other hand, the presence of globally accessible singletons will often remove the need to pass parameters down long call structures, thereby avoiding needless dependencies.

E. Decorator—adding functionality in layers

This pattern is used to extend the functionality of a base object. Inheritance is appropriate when extensions are done in a hierarchic manner; Decorator is used where more than one extension can be active at any given time, and if extension is to be done dynamically at run time.

Central to the Decorator pattern is a collection of decorator objects, embedded in the base component. Whenever there is a call to the "decorated" method, the base component will call the corresponding method in each decorator object in turn, giving each of them a chance to add its functionality. This is in contrast to inheritance and virtual methods, where a single method instance overrides all earlier definitions.

Decorators can significantly reduce coupling in a system, since they are not visible from the outside of the pattern—clients simply call the method without ever being aware of a possibly large number of decorator objects inside. However, this can also make the system harder to analyze, since the actual call graph (including order of calls) cannot be determined in advance.

F. Template Method—replacing building blocks

Template Method is suited for cases where a high-level algorithm remains constant, but its underlying building blocks are subject to change. The algorithm is written in a method in a base class, and calls virtual methods to perform the building-block steps. By providing alternative implementations of one or more building block methods, a subclass can alter the details of the algorithm, while reusing the algorithm itself and any other building blocks.

The pattern is applicable to both large and small problems. The high-level task may be to fetch and process data from a database, and a building block might be the syntax for one specific DBMS. It could also be a sophisticated routing algorithm where the building blocks are themselves algorithms composed of smaller, replaceable blocks, forming a whole tree of code.

III. Related work

Prechelt and Unger [10], [11] reported on two experiments on Design Patterns. The first experiment sought to test whether the presence of Design Patterns in program documentation had an effect on maintenance; the results were positive in the sense that maintenance was either faster or introduced fewer errors when the software was documented in terms of Design Patterns. Observer, Composite and Visitor were the tested patterns.

In the second experiment maintenance tasks were performed on "equivalent" versions of several programs, with one version containing Design Patterns and the other having a simpler, more straightforward structure. The results varied with the pattern—Visitor was inconclusive, while Observer and Decorator caused less time to be used for the maintenance tasks. The Abstract Factory pattern had no significant effect.

This experiment was replicated by Vokáč [12], with the difference that a real programming environment was used instead of pen and paper as in the original experiment. Paid industry professionals were used as subjects. The results were similar for the Observer, Decorator and Abstract Factory patterns, while the Visitor and to some extent Composite patterns had strongly negative results—code derived from these patterns proved significantly harder to maintain both in terms of time used and the number of errors.

A case study performed by Bieman *et al.* [13] analyzed 39 versions of an evolving object-oriented software system. They found a strong relationship between class size and change frequency. Having taken this into account, they also found that classes that participate in Design Patterns are just as change-prone as other classes, and that change-proneness is positively correlated with reuse. Classes that were the most reused were also the most change-prone. A later, more extensive study [6] mostly confirmed these findings.

A threat to the validity of that study is the size and nature of the data set. The Singleton pattern was identified in 10 instances, Iterator in four and Factory Method and Proxy in only one instance each. The system under study evolved from 24 000 LOC and 199 classes to 32 000 LOC and 227 classes over the study span. Intermediate versions were not evaluated, but the total number of changes per class was counted. All changes were included, regardless of whether they were preventive, adaptive, perfective or corrective.

Further case studies have been conducted by several groups [14]–[16]. The subjects of the studies are industrial systems of up to 30 000 LOC, but they address re-engineering or construction concerns, not maintenance over extended periods.

IV. Identifying Design Patterns in C++ Code

In this chapter we briefly describe our new tool for extracting Design Patterns from C++ code on a large scale.

A. Goals for a recovery tool

Early in the study, we formulated the following goals for a pattern recovery tool:

- It must be possible to describe a structural signature, and extract from a C++ code base the set of classes that correspond to this signature.
- The method must be flexible, since some design patterns have complicated signatures that are not easily expressible as simple rules. It must be reasonably easy to express a structure, so that the specification can be checked, revised and debugged.
- The method must scale extremely well, and be able to handle amounts of code in the 10^8 LOC range with running times on the order of hours or at least within a weekend. Preferably, much of the processing time should be spent on data preparation that is independent of what patterns are being sought, so that addition of new patterns does not force a re-run of the whole process. If possible and necessary, the method should lend itself to optimization using parallel hardware (storage, CPU) or even clustering.
- The input data should be in the form of "untreated" code files, i.e., whatever structure the source project is already in. Output should be in a form that is easily transferable to statistical packages for further analysis.

B. Tool construction

Existing tools found in the literature [2], [7], [17]–[23] have not been documented to possess the combination of speed, recovery and precision needed for our case study, with the possible exception of the work of Balanyi and Ferenc [24]—which only appeared after our study was in progress.

Due to the lack of a satisfactory, off-the-shelf tool, we decided to construct our own. The tool was built using several sub-components to handle different stages of the process. During our case study, it satisfied all the goals, with the exception of parallel processing; this was not pursued due to both a lack of suitable hardware, and a lack of need.

- C++ code is parsed using a commercial parsing tool [25], to create a database of metadata—lists of types, names, variables, methods, etc. and the relations between them, such as "method X calls method Y". The tool is extremely fast and handles million-line projects with no problems.
- The metadata is transferred from the original, proprietary format into a SQL database. This makes it possible to perform analysis using a declarative query language (SQL), instead of through custom programming in C (the API offered by the tool vendor). Relational databases are well suited to handling large amounts of data in this fashion.
- Queries in SQL are derived from the structures of the selected Design Patterns, for each pattern. The queries are used to retrieve objects and relations that match the design patterns. An example is to search for a method that returns an object reference, where the object type is a base class for inheritance, and the method calls one or more constructors of subclasses from the same hierarchy as the return type. With some additional restrictions, this corresponds to an instance of a Factory Method.
- The queries are applied to the database of program metadata to retrieve pattern instances. The task of generating an optimized execution plan for the query is left to the database management system; after some simple index tuning the queries ran in just a few seconds even on large data sets.

In addition, if we wish to examine the evolution of the system over time, data extracted from a Version Control System (VCS) can be added to the metadata. By combining VCS data with the program metadata, the evolution of patterns over time can be traced. More details about the tool can be found in [26].

C. Tool performance: Recovery, precision and scalability

The success of a tool like this can be measured in several ways. This section deals with errors in the recovery of patterns, while the next section addresses scaling and performance.

There are two types of error that should be evaluated: false positive and false negative. A false positive occurs when a pattern is recognized, even if the classes concerned do not really conform to the pattern structure. A false negative is when classes do conform to the pattern, but are not recognized.

False positives are relatively easy to check, by inspecting the code that has been identified by the tool to see if it really forms a pattern. We should note that classes may well conform to a pattern structure even if the original designer did not intend them to do so, or perhaps was unaware of the existence of the pattern. Indeed, in the case study in section VI we shall see that that was sometimes the case.

Depending on the context and degree of match between pattern structure and actual structure, we may classify such an occurrence as either a false positive, or evidence that this structure is a true pattern—a solution repeatedly chosen in the field (e.g., Introduction of [1]).

False negatives are however much more difficult to determine, since we would in principle have to identify all instances of all patterns in the code, and then compare to the output of the tool. For software of any realistic size ($>$10 KLOC) this is not a realistic task. As a substitute for a total analysis, we can look at a random sample of the classes and determine the patterns present; this will give us an indication of the number of false negatives.

Generally, a precise and detailed specification of the structural signature of a pattern will decrease the number of false positives, but increase the risk of false negatives. The exact structural signature is tied to language-specific features, and is also dependent on how stringently we interpret the structure of the pattern in question, wiz. the discussions on inheritance depth and other parameters in [18], [21], [24].

One challenge is caused by the presence of preprocessor macros in the C and C++ languages. It is possible to code complicated declarations and structures in macros, that are then hard to parse and analyze. While the problem is not completely intractable, as shown by Badros and Notkin [8], the parsing tool we use has only limited support. This aspect must be taken into account when validating the use of our tool on any given set of software.

An analysis of the error rates of the pattern recovery tool was performed as part of the case study in section VI. The summarized results were as follows:

Pattern	Instances	False positive	False negative
Factory	51	15.1 %	2.3 %
Singleton	63	0.0 %	2.3 %
Observer	23	15.0 %	3.7 %
Template Method	143	1.2 %	10.3 %
Decorator	9	0.0 %	3.7 %

TABLE III
SUMMARY OF ERROR RATES FROM THE CASE STUDY

To be useful on large projects, a pattern extraction tool needs to have a reasonable base speed, and to scale well with code size. The importance of speed of course depends on the usage. If we only wish to perform a once-off analysis of a system, running time is less important than if we wish to follow a system over time and do repeated scans.

As described in the literature [2], [7], [17]–[24], most existing tools have been tested on code sizes up to 10^4 LOC, and usually less (or not specified). Our tool was designed to support a case study that followed a 500×10^3 LOC system over time, with weekly snapshots for three years. A total of 76×10^6 LOC had to be analyzed, with the realistic assumption that the analysis would have to be performed several times during perfection of the method. This fundamental requirement strongly influenced the choice of tools and technical platform.

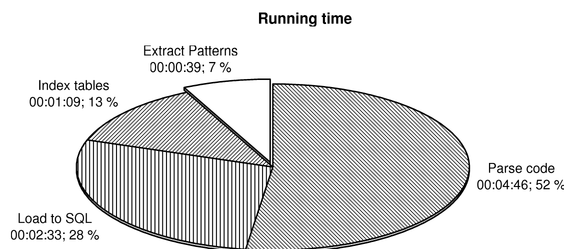

Fig. 1. Length and phase distribution of running times for the Pattern recognition tool, on 500 000 LOC.

The distribution of running time over the different phases of the parsing and extraction process is shown in Figure 1. The actual extraction of the patterns takes place in the final 7% of the run, for all five patterns together. This means that adding another pattern, amending an existing one or experimenting with various structures and statistics can be done very easily and quickly. All the preceding steps parse, transfer and index the metadata in ways that are general and independent of the specific patterns being sought.

V. THE SUPEROFFICE CRM5 PRODUCT

SuperOffice CRM5 is a Customer Relationship Management product—used by companies to keep track of their customers, sales force diaries, activities and sales. It runs on Microsoft Windows and is a classic client/server application. All data is stored in a relational database, while the application and presentation logic are in the client. The company has a heavy emphasis on user friendliness and consequently there is a lot of GUI code. The application is written entirely in C++, and was almost totally rewritten in the year 2000.

The application is sold in one standard version in 11 languages, and is installed at 11 000 customer sites. New versions were released approximately twice a year during the study period (2001-2003).

Metric	Value
Total lines	1 114 092
Lines of code (LOC)	505 367
Number of classes	2 047
Number of files	2 809
Number of methods	30 823
Declarative statements	150 685
Executable statements	194 625

TABLE IV
DESCRIPTIVE METRICS FOR SUPEROFFICE CRM5

The code has been maintained and extended for the past three years by a stable team of developers, about half of whom were involved in originally writing the code. A bug tracking system [27] was used to track reported defects in the code, whether found internally during formal pre-release testing, or at customer sites. The bug tracking system was partially integrated with the Version Control system [28]. Changes to the code were checked into the version control system in transactions generally corresponding to one functional change, whether corrective or otherwise.

Table IV gives basic size metrics for the product, while Table V describes the evolution of the system during the 153-week period covered by the study.

Change type	No.	%	Lines added	changed	deleted
Corrective	1 619	31%	15 598	10 503	6 962
Other	3 562	69%	33 369	21 813	20 202
Total	5 181		48 967	32 316	26 164

TABLE V
SUPEROFFICE CRM5 EVOLUTION

VI. EXTRACTING DEFECTS AND DESIGN PATTERNS FROM THE CRM5 CODE

The goal of the study was to investigate possible correlations between pattern usage and corrective maintenance.

The presence of defects was determined by analysis of the Version Control System (VCS) where all the source code resides. This system was partially integrated with the Defect Tracking System used during the study period. A further textual analysis was made of the comments in the VCS, to recover defects that did not have such direct links. The approach was validated by evaluating a sample drawn at random from the full set of code changes, to check that there were no wrong classifications.

The next step in the study was to obtain weekly snapshots of the complete source code, for the whole study period. Each snapshot reflects the state of the system at midnight between

Saturday and Sunday, starting on February 4th, 2001, when the system consisted of approximately 3 700 files totalling 90 MB of text. The snapshots grew slightly over time as code was added to the system; in sum, they consist of about 500 000 files and 14 GB of text.

The pattern extraction tool was applied to each snapshot, to parse all the code, load the metadata into the SQL Server, index it and extract Design Patterns. Typical running times are shown in Figure 1. Each snapshot had its own, preallocated empty database in the server. The whole process took slightly less than 26 hours on a 2.8 GHz PC with 4 GB of RAM (max. 800 MB actually in use) and standard disk drives.

A. Error rates

Since a Design Pattern is an informal specification of a recommended structure, it will be translated into program code differently in different projects. Any discussion of error rates must therefore be seen in relation to application of the method to a particular set of software.

It is also necessary to define exactly what we mean by an instance of a certain Design Pattern. We consider Factory as a simple example: one Factory class may create one or more Product classes. Should we count every combination of Factory and Product as an instance, or should one Factory class count as a single instance, regardless of the number of products? Similar situations occur in most patterns, since they specify relationships between multiple classes.

In our evaluations, we have adopted the simple definition—one Factory class counts as a single instance. Similarly, we count one instance of the Observer pattern for every Subject; one Template Method for each template method; one Decorator for each Decorator class; and one Singleton for each Singleton class.

It was necessary to adjust the pattern-recovery tool for two particular patterns: Observer and Decorator. The Observer pattern can be implemented in two different ways. The "classic" structure specifies that each Subject should keep track if its Observers directly, using a collection of object references. An alternative approach is to use a generalized message broker to handle the relations between subjects and observers, and this is the approach adopted globally in the CRM5 code. The tool was adapted to this Subject/Broker/Observer structure.

For Decorator, a related problem exists, that of aggregation. The tool was adapted to the kind of aggregation generally used in CRM5 code.

B. False positives

The rate of false positives was determined by manually examining all detected instances of all patterns. The results for all patterns are given in Table VI.

Pattern	Instances	False	Error rate
Factory	53	8	15.1 %
Singleton	45	0	0.0 %
Observer	20	3	15.0 %
Template Method	163	2	1.2 %
Decorator	9	0	0.0 %

TABLE VI
FALSE POSITIVES

Most of the false positives identified for the Factory pattern were classes that had inner (nested) classes used for their implementation. From the outside this looks like a Factory instance, since the outer class creates the inner class and no-one else does. However, this usage does not correspond to the intent of Factory, so it was classed as a false positive. It is quite feasible to add the condition "product class must not be nested within factory class" to the structural signature for Factory in a refined version of the rules.

In the Observer cases, we are dealing with a "loosely coupled" version of Observer, where all notifications are handled by a centralized message broker class. This differs somewhat from the classical, simple Observer pattern where every Subject keeps track of its Observers separately. There are other forms of interaction through the Message Broker than just according to the Observer pattern, and the three false positives are such cases. Since the structure of the pattern is already quite complex, further refinement is difficult without risking higher false negatives.

The structure for Template Method allows for multiple levels of inheritance, and does not require more than one call to an underlying, virtual primitive method to consider the caller a parent method. It is a matter of taste whether one would want to tighten the definition, i.e., to demand that there is more than one implementation of the primitive method, or more than one primitive method for each template method. The number of false positives would most probably decline, but the number of false negatives might increase.

Decorator has a somewhat problematic structure, in that it contains an aggregation (the set of Decorators for one class) that can be implemented in many ways. The signature was adapted to the known kind of aggregation implementation in this code, and so we should not take the zero error rate as being guaranteed in a different setting. Also, false negatives are more probable for this pattern.

Singleton is a pattern that is fairly easy to recognize, and so the low false positive rate is as expected.

C. False negatives

To determine the actual rate of false negatives, we would have to evaluate all classes and find all cases where a class participates in a pattern but has not been detected by the tool. With more than 2 000 classes this is not realistically possible.

Instead, we have chosen to evaluate a random sample of classes, to get an estimate of the false negative rate. From prior knowledge of the code (the author has previously served as a developer and architect on the development team), a low rate of false negatives was expected. To calculate the necessary sample size, the following criteria were set:

Required power: 90%. Hypothesized proportion (false negative rate): 20%. Alternative proportion (rate to be tested for): 10%.

This yielded a required sample size of 109.[2] To guard

[2]MiniTAB [29] version 13.32 was used for all statistical calculations.

against randomly choosing classes that are trivially small or otherwise nonrepresentative, the actual sample size was increased to 125.

Classes were chosen using a uniformly distributed random number generator. The chosen classes covered all major modules of the program. Figure 2 shows the distributions of class sizes, for the full system and the sample.

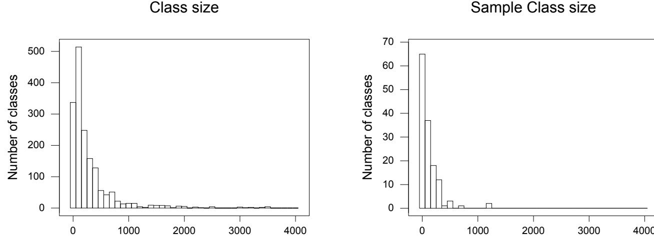

Fig. 2. Histogram of class sizes, of the full system (left pane) and 125-class sample (right pane)

Out of the 125 classes inspected, a total of 9 were false negatives. Of these, one was part of a Decorator, one an Observer and the rest were Template Methods (six of the seven were actually part of the same instance of Template Method, missed due to macros in the code that hid the virtual method declarations).

The bounds for the false negative rate, as derived from these observations, are given in Table VII.

Pattern	N_{false}	95%	P
Factory	0	2.3 %	0.000
Singleton	0	2.3 %	0.000
Observer	1	3.7 %	0.000
Template Method	7	10.3 %	0.000
Decorator	1	3.7 %	0.000

TABLE VII
FALSE NEGATIVES

When interpreting these results, we must keep in mind that they apply to the SuperOffice CRM5 code only. Given the number of possible ways to implement the structure described by a pattern, the validity of the tool must be tested for each new coding style.

VII. Statistical model and Quantitative results

The goal of our case study was to determine if the presence of certain design patterns is correlated with the defect frequency of the code. Our raw data consist of the C++ code, here divided into classes, size metrics for each class, and indicators for whether the class participates in a pattern or not. The amount of code that is not a member of a class (global functions) is so small as to be negligible ($< 0.1\,\%$).

A. *A simple model*

To analyze the data, a binary logistic regression model was chosen [30]. In this model, there are two possible outcomes of an observation, one of them termed an "event" or "success". The logistic equation is

$$G(\text{event}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \cdots \beta_n X_n \tag{1}$$

where G is the link function, that maps the interval $(0, 1)$ onto the real numbers. We use the standard logit function, $G(z) = \frac{1}{1+e^{-z}}$. The X's represent the effects we wish to measure, plus known or suspected confounding factors. An *event* $(z = 1)$ is defined as the occurrence of at least one corrective change to one class within one snapshot (week).

We define one indicator variable X_{Pattern} for each design pattern we wish to test for: $X_{\text{Obs}}, X_{\text{Sing}}, X_{\text{Decor}}, X_{\text{TM}}$ and X_{Fact}. From both previous work [13] and intuition we know that code size is a possible confounding variable; it is reasonable to assume that there are more defects in a large class than in a small one. We also wish to test for any systematic trends in time, e.g., whether the defect rate varies in a linear fashion over time. We thus add the factors X_{Size} and X_{Week} to the model.

The estimates $\hat{\beta}_i$ are generally interpreted in terms of odds ratios, giving the change in odds for an event when the corresponding X_i changes by one unit. We therefore express code size in KLOC; adding one line to a class should not affect the number of defects much, but adding a thousand lines certainly should. The time variable is expressed as a week number, since we have one snapshot of the code per week; the range is $(1 \ldots 153)$. The X_{Pattern} are indicator variables, with 0 denoting the absence and 1 denoting the presence of a pattern for a given class.

Our data set consists of all the classes in the system, over the whole study period. This constitutes a total of 236 876 observations. While these are actually 153 repeated observations of 1550 classes[3], we can still consider them "independent" in the sense that we are looking at the defect frequency relative to design patterns, size and time. Given the size of the system, the possible presence of a few classes with abnormally high defect rates should not have an undue influence.

An alternative way of analysis would have been to perform an Analysis of Variance, on a data set consolidating all 153 snapshots. However, the data material does not contain all combinations of all factors, resulting in empty cells in the model. The logistic regression is less sensitive to this, and also gives results that are easier to interpret.

This gives rise to the following model:

$$\begin{aligned}\frac{1}{1+e^{-z}} &= \beta_0 + \beta_O X_{\text{Obs}} + \beta_S X_{\text{Sing}} + \beta_D X_{\text{Decor}} \\ &\quad + \beta_T X_{\text{TM}} + \beta_F X_{\text{Fact}} \\ &\quad + \beta_K X_{\text{Size}} + \beta_W X_{\text{Week}} \end{aligned} \tag{2}$$

where

$$z = \begin{cases} 1 & \text{if a corrective change occurred} \\ 0 & \text{if no corrective change occurred} \end{cases}$$

Non-corrective changes are not counted in this model. There were 1 619 events out of a total of 236 876 observations. The

[3]The discrepancy between the number of classes cited here and in Table IV is caused by the elimination of undefined classes, templates, classes from standard system libraries and other library code that has not been maintained.

β_i were estimated using MiniTAB Binary Logistic Regression, yielding the following results:

Coefficient		P	Odds Ratio	95% CI Lower	Upper
Constant	(β_0)	0.000			
Factory	(β_F)	0.000	**0.66**	0.54	0.81
Singleton	(β_S)	0.000	**2.69**	2.24	3.24
Observer	(β_O)	0.000	**1.53**	1.33	1.75
Template Method	(β_T)	0.002	**0.61**	0.44	0.83
Decorator	(β_D)	0.159	0.49	0.18	1.32
Size KLOC	(β_K)	0.000	**1.98**	1.85	2.11
Week	(β_W)	0.000	**0.99**	0.99	0.99

Log-Likelihood = −9290.559 Test that all slopes are zero: G = 789.579; DF = 7; P-Value = 0.000

Goodness-of-Fit Tests

Method	χ^2	DF	P
Pearson	97398	10^5	1.000
Deviance	13031	10^5	1.000

TABLE VIII

QUANTITATIVE RESULTS FROM THE FITTING OF THE REGRESSION MODEL IN EQUATION 2 TO THE OBSERVED DATA

The primary results from a logistic regression are the odds ratios, whose interpretation is as follows: given a change of one unit in the underlying factor, and keeping all other factors unchanged, the odds ratio gives the change in probability for the occurrence of an event—in our case, the correction of a defect.

There are several interesting results in this model. First of all, four patterns have strongly significant odds ratios, but not in the same directions. Factory and Observer are correlated with a lower defect frequency, while Observer and especially Singleton are correlated with a higher defect frequency.

Second, the Size effect is highly significant, as expected. Adding 1 000 LOC to a class roughly doubles the probability of a defect, all other circumstances being constant. Finally, there is a very slight downward trend (odds ratio 1:0.99) in the number of defects over time, however this is not considered strong enough to invalidate the other results.

B. A full model including interactions

The model in the previous section only takes into account the main effects (the five patterns under study) and two possible confounders (size and time). However, this is too simple as there are more effects to consider.

First, the participation of classes in patterns in not a simple 1:1 or 1:0 relation. It is possible for a class to participate in more than one pattern; indeed, the occurrences of patterns and pattern combinations in the data material as shown in Table IX indicate that combinations have to be taken into account.

To take this into account, the model is recoded. Instead of using an individual indicator variable for each pattern, the pattern participation of each class is expressed as a combined PATTERN variable. The variable contains an ‘F’ if the class participates in Factory, an ‘S’ if it participates in Singleton, etc. A class that participates in both Factory and Singleton would have ‘FS’ as its PATTERN.

Patterns	Occurrences	%
No Pattern	183634	77.5 %
Factory	20237	8.5 %
Singleton	3331	1.4 %
Observer	16061	6.8 %
Template Method	5381	2.3 %
Decorator	1513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2279	1.0 %
Observer + Singleton	2390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

TABLE IX

FREQUENCIES OF PATTERN OCCURRENCES, AND PERCENTAGE OF CODE COVERED BY THE PATTERNS

The regression is then re-run with Pattern as a factor, i.e., each distinct value is considered a separate coefficient. As before, the baseline is formed by the classes that do not participate in any pattern. We then obtain the results in Table X.

Coefficient	P	Odds Ratio	95% CI Lower	Upper
Constant (No pattern)	0.000			
Week	0.000	**0.99**	0.99	0.99
Size (KLOC)	0.000	**1.90**	1.77	2.03
Factory	0.001	**0.68**	0.53	0.86
Singleton	0.000	**4.30**	3.43	5.40
Observer	0.000	**1.83**	1.57	2.13
Template Method	0.011	**0.63**	0.44	0.90
Decorator	0.170	0.50	0.19	1.34
Factory + Singleton	0.001	**2.00**	1.35	2.96
Factory + Observer	0.500	1.32	0.59	2.96
Observer + Singleton	0.000	**2.08**	1.43	3.04
Observer + Template Method	0.637	1.20	0.57	2.52
Factory + Observer + Singleton	0.145	1.82	0.81	4.09

TABLE X

QUANTITATIVE RESULTS FROM THE FITTING OF THE RECODED MODEL WITH PATTERN COMBINATIONS TO THE OBSERVED DATA.

We observe two combinations yield significant odds ratios: Factory + Singleton and Observer + Singleton. This means that classes that simultaneously participate in both Factory and Singleton are twice as error-prone as classes that do not participate in any pattern (odds ratio 2.00); a similar result (odds ratio 2.08) is seen for the combination of Observer + Singleton.

However, there is another possible confounding factor to be taken into account: the possible interaction between size and pattern. This would be the case if certain patterns are correlated with a higher or lower size than other patterns or the classes in general; such a correlation would imply a collinearity between the pattern and the size coefficients.

We already have an expectation are that some patterns (Observer) will lead to larger classes than others (Factory), wiz. the discussion in sections II-C and II-B. The possibility of an interaction between pattern participation and code size has to be assumed, and we add interaction terms $X_{\mathrm{Pattern}}X_{\mathrm{Size}}$ to the model for each pattern.

The full set of Pattern × Size interactions yields a number of

non-significant coefficients, and their presence disrupts the rest of the model. We therefore eliminate those interaction terms that are not significant; similarly, we eliminate the Pattern × Pattern terms from Table X that were not significant.

We thus obtain the following final model and results:

$$\begin{aligned}\frac{1}{1+e^{-z}} = {} & \beta_0 + \beta_O X_{\text{Obs}} + \beta_S X_{\text{Sing}} + \beta_D X_{\text{Decor}} \\ & + \beta_T X_{\text{TM}} + \beta_F X_{\text{Fact}} \\ & + \beta_K X_{\text{Size}} + \beta_W X_{\text{Week}} \\ & + \beta_{SO} X_{\text{Sing}} X_{\text{Obs}} + \beta_{SK} X_{\text{Sing}} X_{\text{Size}} \\ & + \beta_{OK} X_{\text{Obs}} X_{\text{Size}} \qquad (3)\end{aligned}$$

where

$$z = \begin{cases} 1 & \text{if a corrective change occurred} \\ 0 & \text{if no corrective change occurred} \end{cases}$$

Coefficient		P	Odds Ratio	95% CI Lower	95% CI Upper
Constant	β_0	0.000			
Week	β_W	0.000	**0.99**	0.99	0.99
Size (KLOC)	β_K	0.000	**1.69**	1.53	1.87
Factory	β_F	0.000	**0.63**	0.51	0.77
Singleton	β_S	0.141	1.35	0.91	2.02
Observer	β_O	0.000	**1.55**	1.26	1.91
Template Method	β_T	0.048	**0.72**	0.52	1.00
Decorator	β_D	0.154	0.49	0.18	1.31
Singleton + Observer	β_{SO}	0.000	**0.32**	0.21	0.48
Singleton × Size	β_{SK}	0.000	**13.18**	6.29	27.61
Observer × Size	β_{OK}	0.009	**1.21**	1.05	1.40

Log-Likelihood = -9245.342 Test that all slopes are zero: G = 880.012; DF = 10; P-Value = 0.000

Goodness-of-Fit Tests

Method	χ^2	DF	P
Pearson	98299	10^5	1.000
Deviance	12941	10^5	1.000

TABLE XI

QUANTITATIVE RESULTS FROM THE FITTING OF THE FINAL REGRESSION MODEL IN EQUATION 3 TO THE OBSERVED DATA

VIII. INTERPRETATION OF RESULTS

From the quantitative results in Table XI we see that no significant correlation has been detected for the Decorator and Singleton patterns. The remaining coefficients are significant, and the goodness-of-fit tests indicate that the model fits the data well.

The remaining odds ratios must be interpreted with some care. The presence of a correlation does not by itself guarantee *causality*; for that we must go back to the code and perform a more qualitative analysis. However, even at a quantitative level the results are interesting.

We can see that the systematic evolution in the number of defects with time was very slight, as the Week odds ratio is very close to one—the estimated value of 0.99 indicates a small decrease in the defect ratio over time. Thus, while the defect rate goes down significantly (P=0.000), the size of the effect is negligible. On the other hand, adding 1 000 lines of code to a class increases the probability of defects by two thirds (ratio 1.69)—excluding any simultaneous effects of pattern participation, as these are handled in the interaction terms.

For the Factory pattern, the expectation from sections II-B is supported. The odds of a defect in a class involved in a Factory pattern (both the Factory class itself and its products) are less than two thirds (0.63) of the "background" defect rate. The insignificance of the Factory × Size and Factory × Singleton interactions (both dropped from the final model for lack of significance) increases our confidence that we are indeed looking at a true effect associated with the pattern—or rather, an *indication of the complexity of the situations in which this pattern was used.*

For Decorator, the pattern is not used much (one Window class has 8 different, simple Decorators attached) and therefore the data are too sparse.

Template Method, on the other hand, is used in many different contexts in the code, ranging from small and simple, to relatively deep and complex (3–4 levels inheritance and nested template functions). The wide spread is the cause of the rather weak results seen here, P=0.048. However, in the context of complexity and pattern use, the observation that there is only a weak relation is itself interesting—Template Method is a pattern that can hide a complex system, or be used in simple circumstances. This contradicts the remark made by [24] of Template Method being a trivial pattern. The tendency observed here is that it does tend to lower the defect rate somewhat (an odds ratio of 0.72), so it is more used for the simpler functions.

In the CRM5 system, the Singleton pattern is used for objects that are more or less global in scope, in the form of caches, utilities and repositories of application state. An example is a cache of user preferences, another is a set of interconnected objects that keep track of the GUI state (select panels, current data set, etc.). In the regression model we do not have a significant result (odds ratio 1.35, but P=0.141) for Singleton in isolation.

However, Singleton is a pattern that has two significant results in combination with other factors: Size, and the Observer pattern. Significant interaction factors mean that we cannot interpret the main effect on its own, but rather have to consider it as a function of the factor it interacts with [31]. We follow the interpretation given by Hosmer and Lemeshow [32], and calculate the odds ratio for Singleton given simultaneous participation in Observer as $\widehat{\text{OR}}_{SO} = \exp[\beta_O + \beta_{SO}]$ =0.43.

The explanation is found by a qualitative analysis of the classes in question in the code. A set of 7 small classes form a state machine that controls the global GUI state of the whole application; these classes are Singletons and simultaneously Observers of each other. They were carefully designed early on in the project, and since the underlying requirements haven't changed, they have stayed very stable. Thus their defect rate is also small. This explains why Singleton and Observer on their own tend towards higher defect rates (odds ratios of 1.35 and 1.55 respectively), but together have a significantly lower defect rate. We would not expect this interaction effect to be generally valid for other programs.

Since size in KLOC is a continuous variable, the interpretation of the interaction between a pattern and size yields a set of odds ratios, one for each chosen value of the size. To determine proper size values, we must look at the size distributions of the classes that participate in the Singleton and Observer patterns; the histograms are shown in Figure 3.

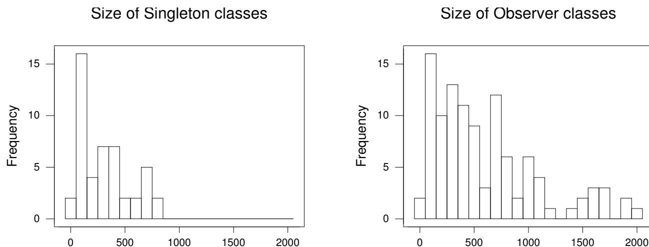

Fig. 3. Histogram of class sizes, of the Sinlgeton classes (left pane) and Observer classes (right pane)

The odds ratio for the expected defect rate, for a combination of a given pattern and a given code size is calculated as $\widehat{\mathrm{OR}} = \exp[\beta_{\mathrm{pattern}} + \beta_{\mathrm{interaction}} \times \mathrm{KLOC}]$. Table XII shows these odds ratios for Singleton and Observer, for code sizes that correspond to the actual classes in the CRM5 system.

KLOC	Singleton OR	Observer OR
0.1	1.75	1.58
0.25	2.58	1.63
0.5	4.91	1.71
1.0	17.82	1.88
1.5	—	2.07
2.0	—	2.28

TABLE XII
ODDS RATIOS FOR SINGLETON AND OBSERVER, GIVEN SOME CLASS SIZES

We can observe that classes that participate in the Singleton pattern are very sensitive to size; this is partly explained by presence of the small, stable classes mentioned above. The same effect applies to the Observer × Size interaction, with the difference that there are more instances of the Observer pattern than the Singleton pattern (6.7 %, vs. 1.8 % of the total code, from Table IX). This is why the interaction is significant, but without invalidating the main effect related to Observer.

The odds ratio for Observer alone is 1.55 (this corresponds to setting the size value to 0), supporting our expectation from Section II-C: Observer is a relatively complicated pattern that is used in situations with coupling between multiple, nontrivial classes. The higher than average defect frequency is what we expected to happen.

IX. THREATS TO VALIDITY

In a study like this, there are multiple threats to validity, both internal and external. Internal validity is concerned with the consistency of the measurements, appropriate use of tools and methods. External validity concerns the degree to which the data and results are transferable outside the particular context of the study.

One threat arises from the use of an automated tool to recover design patterns from code, an inherently imprecise and difficult process. The tool has therefore been tested and validated, and the results from section VI-A indicate that, in the current study, the tool is sufficiently reliable to be used. This means that it has acceptably low false positive and false negative rates, when applied to the code in this case study.

The choice of the patterns to be analyzed was based partly on their known structure and expected effects, and partly on their popularity as determined from surveys of academic literature, discussion groups and the Web in general.

A second threat is related to the software chosen for analysis, in that it should be of sufficient size and complexity that patterns and defects occur often enough to give statistically valid results. As discussed in section VII, the pattern Decorator did not fulfil this criterion. No strictly quantitative conclusions can therefore be drawn regarding Decorator from the current material. The spread of values for Template Method is an interesting and valid result—given the high number of occurrences—in spite of its lack of simple quantitative significance.

The choice of statistical model and its content must also be considered. Given the size of the data set, a detailed analysis and classification of each defect is beyond the scope of the study. With a binary outcome (defect/no defect) and indicator variables for the presence or absence of patterns, logistic regression is a natural model.

In addition to main effects and two confounders (size and time), all pattern × pattern interaction terms that were actually present in the data were evaluated. Further interaction terms for patterns versus size were included, based on intuition and the experience of other workers in the field, especially Bieman *et al.* [6], [13], who found a strong size effect. In our case, a size effect was found for two patterns (Observer and Singleton), linked to a special set of small, stable classes in the code.

The external validity of the study is mainly limited by two factors: the applicability of the tool to other coding styles in other software, and the usage of design patterns elsewhere.

The software itself is a long-lived industrial product of considerable size, thus small size or unrealistically simple design should not be a threat. It was designed by a team that was aware of Design Patterns in general, without being fanatics on the subject. It is to be hoped that this is representative of commercial software designed with patterns.

The studied defects came from both internal pre-release testing, and reports from actual users. The study span of three years ensured that there was in fact time to incorporate user feedback, causing measurable changes to appear in the code.

As argued in sections II-B and II-C, we believe it is the fundamental nature of the Factory and Observer patterns that the former is used in simpler, less tightly connected code than the latter. The observed results support this observation. However, only further observations of other systems will support generalization outside the studied domain.

The applicability of the pattern extraction tool is fairly easy to check for any other software system, by performing a validation similar to the one done in sections VI-B and VI-C.

If no major adjustments have to be done to the recognition algorithms, we may conclude that the tool is more generally applicable. However, the general applicability of the tool—or lack of it—is not a significant factor in the applicability of the results of the present case study, for which the tool was extensively validated.

X. Summary of results

We wished to determine if there is any systematic correlation between the occurrence of certain Design Patterns and defect frequency in an industrial product. To analyze the source code, we designed and implemented a tool capable of extracting information about the presence of selected design patterns from C++ code. The tool analyzed 76×10^6 LOC in less than 26 hours. The patterns were selected based on surveys of the Web and academic literature, as well as the suitability for evaluating the tool.

False positive errors (identification of a pattern where none exists) were determined by inspecting each pattern occurrence identified by the tool, and varied from 0 % to 15 %. False negative errors (missing a pattern) were estimated statistically from a random sample of classes, and varied from 2 % to 10 %. These rates compare favourably with other work in the field. The combination of speed and precision opens the possibility to analyze large software products over time.

The tool was then applied to a Customer Relationship Management product, consisting of more than 500 KLOC. Snapshots were obtained for each week in a three-year period, and analyzed using a logistic regression model. The response variable was the presence of a defect in the code, and the model terms were the presence of the five patterns, size and week number, and interaction terms.

Significant correlations were found for the Factory, Singleton and Observer patterns. Code related to Factory had a lower defect rate (63 %) than the code in general, while Singleton and Observer were correlated with higher defect rates (135 % and 155 %). In the case of Singleton and Observer there were also significant interactions with size, supporting the hypothesis that these patterns tend to be used in complex areas, with more code and higher defect frequencies.

The Decorator pattern did not occur often enough to yield statistically significant results. Template Method occurred many times in many different contexts, so the spread of defect frequencies was large and no strong, single conclusion can be drawn. However, this illustrates the many uses to which this pattern can be put, and the different effects it may have.

XI. Conclusions and future work

The usage of Design Patterns has been cited as a way to make good designs easier to develop, even for less experienced developers. As a corollary, since good design is presumed to lead to lower defect rates and other benefits, the use of patterns is also expected to lead to lower defect rates.

However, we find that the reality—at least in the case of the software studied—is not quite that simple. Well-known Design Patterns have widely different sizes, complexities and applicability, so that the use of patterns by itself is no guarantee of few defects.

In any non-trivial software product, there will be areas that have an irreducible, significant complexity. Unless they are very carefully designed and maintained, such areas will have higher defect rates than the average in the product.

We believe that some patterns—notably Singleton and Observer in our study—tend to be associated with such complexity. Thus, even the "proper" use and implementation of these patterns may not be enough to reduce the defect rate to the general average. But the applicability of these patterns can serve as a useful warning sign to the developers: if Observer is the proper solution, then that area of the software is probably complex and warrants an above-average effort in its design and implementation.

Correspondingly, we found that the Factory and to a certain extent Template Method patterns are associated with lower complexity. While this should not encourage complacency on the part of the designers, it does seem that areas covered by these patterns require less painstaking attention. As good design resources are always at a premium, these conclusions can hopefully help focus them in the most useful place.

Future work is envisioned as an interaction between tool development and code analysis. In its current state the tool is fast enough to be used on large projects, but it is necessary to validate it on different kinds of programs, from different development groups. The Open Source community should be a good source of projects to analyze. This should result in an improved tool in terms of applicability to different coding styles. Extending the number of patterns analyzed is also relevant.

With an improved tool, an analysis similar to the present case study can be performed, to see if our results are more general in nature. The existing material can also be reanalyzed for any new patterns added to the tool.

Further study is possible by calculating relevant metrics for the code, to see how the presence of design patterns correlates with trends in the metrics and defect frequencies.

Acknowledgement

The author would like to extend sincere thanks to SuperOffice ASA and Director of Development Guttorm Nielsen, for granting unrestricted access to the CRM5 source code. Prof. Erik Arisholm made major contributions to the statistical modelling, and Prof. Dag Sjøberg provided valuable comments on the style and structure of the paper.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995, 1995.

[2] Y.-G. Gueheneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, 2001, pp. 296–305.

[3] L. Rising, *The Patterns Handbook*. Cambridge University Press, 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Chichester: Wiley, 1996.

[5] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed. Prentice Hall, 2001.

[6] J. Bieman, G. Straw, H. Wang, P. Munger, and R. Alexander, "Design patterns and change proneness: an examination of five evolving systems," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, 2003, pp. 40–49.

[7] R. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, 1999, pp. 226–235.

[8] G. J. Badros and D. Notkin, "A framework for preprocessor-aware c source code analyses," *Software-Practice & Experience*, vol. 30, no. 8, pp. 907–924, 2000.

[9] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, 2001.

[10] L. Prechelt and B. Unger, "Methodik und ergebnisse einer experimentreihe über entwurfsmuster." Fakultat fur Inf. Karlsruhe Univ. Germany, 1999.

[11] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta., "A controlled experiment in maintenance comparing design patterns to simpler solutions." *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1134–1144, 2001.

[12] M. Vokác, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, 2004.

[13] J. Bieman, D. Jain, and H. Yang, "Oo design patterns, design structure, and program changes: an industrial case study," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, 2001, pp. 580–589.

[14] D. Schmidt and P. Stephenson, "Experience using design patterns to evolve communication software across diverse os platforms," in *Ecoop '95 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, 1995, vol. 952, pp. 399–423.

[15] G. Neumann and U. Zdun, "Pattern-based design and implementation of an xml and rdf parser and interpreter: A case study," in *Ecoop 2002 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, 2002, vol. 2374, pp. 392–414.

[16] W. C. Chu, C. W. Lu, C. P. Shiu, and X. D. He, "Pattern-based software reengineering: a case study," *Journal of Software Maintenance-Research and Practice*, vol. 12, no. 2, pp. 121–141, 2000.

[17] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, 1996, pp. 208–215.

[18] G. Florijn, M. Meijers, and P. van Winsen, "Tool support for object-oriented patterns," in *Ecoop'97: Object-Oriented Programming*, ser. Lecture Notes in Computer Science, 1997, vol. 1241, pp. 472–495.

[19] J. Bansiya, "Automating design-pattern identification." Dept. of Comput. Sci. Alabama Univ. Huntsville AL USA, 1998.

[20] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Using metrics to identify design patterns in object-oriented software," in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, 1998, pp. 23–34.

[21] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, pp. 181–196, 2001.

[22] R. Schauer and R. Keller, "Pattern visualization for software comprehension," in *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, 1998, pp. 4–12.

[23] H. Albin-Amiot, P. Cointe, Y. G. Gueheneuc, and N. Jussien, "Instantiating and detecting design patterns: putting bits and pieces together," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). 26-29 Nov. 2001 San Diego, CA, USA [IEEE Comput. Soc. Tech. Committee on Software Eng.; ACM SIGSoft; ACM SIGArt]*. Ecole des Mines Nantes France, 2001.

[24] Z. Balanyi and R. Ferenc, "Mining design patterns from c++ source code," in *International Conference on Software Maintenance (ICSM'03)*. Amsterdam, The Netherlands: IEEE, 2003, p. 305.

[25] S. T. Inc., "Understand for c++," 2003.

[26] M. Vokác, "A tool for recovering design patterns from c++ code, and its application in a case study," Simula Research Laboratory, Tech. Rep., May 2004 2004.

[27] TechExcel, "Devtrack defect tracking tool," 2004.

[28] Perforce, "Perforce software configuration management system," 2004.

[29] M. Inc, "Minitab 13.32," 2003.

[30] D. G. Kleinbaum, *Logistic regression : a self-learning text*, ser. Statistics in the health sciences. New York: Springer, 1994.

[31] A. DeMaris, "A framework for the interpretation of first-order interaction in logit modeling," *Psychological Bulletin*, vol. 110, no. 3, pp. 557–570, 1991.

[32] D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression*, 2nd ed. New York: John Wiley & Sons Inc., 2000.