TOWARDS THE AUTOMATIC IDENTIFICATION OF VIOLATIONS TO THE NORMALIZED SYSTEMS DESIGN THEOREMS

Kris Ven, David Bellens, Philip Huysmans and Dieter Van Nuffel
University of Antwerp, Prinsstraat 13, 2000 Antwerp, Belgium

Keywords: Normalized systems, Software architecture, Modularity, Quality.

Abstract: Contemporary organizations are operating in increasingly volatile environments and must be able to respond quickly to their environment. Given the importance of information technology within organizations, the evolvability of information systems will to a large degree determine how quickly organizations are able to react to changes in their environment. Unfortunately, current information systems struggle to provide the required levels of evolvability. Recently, the Normalized Systems approach has been proposed which aims to address this issue. The Normalized Systems approach is based on the systems theoretic concept of stability to ensure the evolvability of information systems. To this end, the Normalized Systems approach proposes four design theorems that act as constraints on the modular structure of software. In this paper, we explore the feasibility of building a tool that is able to automatically identify manifestations of violations to these Normalized Systems design theorems in the source code of information systems. This would help organizations in identifying limitations to the evolvability of their information systems. We describe how a prototype of such tool was developed, and illustrate how it can help to analyze the source code of an existing application.

1 INTRODUCTION

Contemporary organizations are operating in increasingly volatile environments. Hence, organizations must be able to respond quickly to their environment in order to gain a competitive advantage. Since organizations are becoming increasingly dependent on information technology (IT) to support their operations, the evolvability of the IT infrastructure will determine to a large degree how quickly organizations are able to react to changes in their environment. Unfortunately, current information systems struggle to provide the requested levels of evolvability. One of the challenges that contributes to this issue is the existence of Lehman's Law of Increasing Complexity which states: *"As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.*" (Lehman, 1980, p. 1068). This law implies that over time, the structure of software will become more complex, thereby requiring increasing effort to add new functionality to an existing system.

To address this issue, the Normalized Systems approach starts from the systems theoretic concept of stability to ensure the evolvability of information systems (Mannaert and Verelst, 2009). It argues that the main obstacle to evolvability is the existence of so-called *combinatorial effects*. The Normalized Systems approach defines clear design theorems that impose constraints on the modular structure of software and that eliminate combinatorial effects. Adhering to these theorems therefore results in information systems that exhibit stability and that defy Lehman's law (Mannaert and Verelst, 2009).

Organizations currently have a large number of in-house developed information systems in use. These information systems are likely to contain combinatorial effects that limit their evolvability. Organizations will therefore be looking towards ways to identify these combinatorial effects in their code base, and to devise solutions to improve the evolvability of their information systems. Manually inspecting the source code may be a possibility, but is likely to be a very time-consuming task. Automatic identification therefore seems to be a very interesting alternative. In this paper, we explore the feasibility of building a tool to automatically identify manifestations of violations to the Normalized Systems design theorems. We describe how a prototype of such a tool was developed, and illustrate how it can help to analyze the source code of an existing application.

Ven K., Bellens D., Huysmans P. and Van Nuffel D. (2010).
TOWARDS THE AUTOMATIC IDENTIFICATION OF VIOLATIONS TO THE NORMALIZED SYSTEMS DESIGN THEOREMS.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 23-28
DOI: 10.5220/0003005000230028

2 NORMALIZED SYSTEMS

The basic assumption of the Normalized Systems approach is that information systems should be able to evolve over time, and should therefore be designed to accommodate change. This implies that the software architecture should not only satisfy the current requirements, but should also support future requirements. It is a well-known problem in software engineering that the structure of software degrades and becomes more complex over time as changes are applied to it. This problem has been defined by Manny Lehman as the Law of Increasing Complexity (Lehman and Ramil, 2001; Lehman, 1980). The practical manifestation of this law is that the impact of any given change to an information system will increase over time (Belady and Lehman, 1976; Lehman, 1980; Lehman and Ramil, 2001). This is clearly an important concern for information systems development.

Recently, the Normalized Systems approach has been proposed which aims to address this issue. The Normalized Systems approach uses the systems theoretic concept of *stability* as the basis for developing information systems (Mannaert and Verelst, 2009; Mannaert et al., 2008). In systems theory, stability refers to a system in which a bounded input function results in bounded output values, even as $t \rightarrow \infty$. When applied to information systems, this means that applying a specific change to the information system should always require the same effort, irrespective of the size of the information system or the point in time at which the change is applied. The Normalized Systems approach further relies on the *assumption of unlimited systems evolution* (Mannaert and Verelst, 2009). This means that the system becomes ever larger in the sense that the number of modules—and the number of dependencies between them—become infinite or unbounded as $t \rightarrow \infty$. This may seem an overstated assumption, but actually, it is quite logical as even the introduction of a single module or dependency every twenty years corresponds to an infinite amount for an infinite time period.

Information systems exhibiting stability with respect to a defined set of changes are called *Normalized Systems* (Mannaert and Verelst, 2009). In contrast, when changes do require increasing effort as the system grows, *combinatorial effects* are said to occur (Mannaert and Verelst, 2009). In order to obtain stable information systems, these combinatorial effects should be eliminated. In order to identify and avoid most of these combinatorial effects, a set of four *design theorems* was developed (Mannaert et al., 2008; Mannaert and Verelst, 2009). We will now briefly describe each of these theorems. More details are beyond the scope of this paper and can be found in the literature (Mannaert et al., 2008; Mannaert and Verelst, 2009).

The first theorem, *separation of concerns*, requires that every change driver or concern is separated from other concerns. This theorem allows for the isolation of the impact of each change driver. This principle was informally described by Parnas already in 1972 as what was later called *design for change* (Parnas, 1972). This theorem implies that each module can contain only one submodular task (which is defined as a change driver), but also that workflows should be separated from functional submodular tasks.

The second theorem, *data version transparency*, requires that data is communicated in version transparent ways between components. This requires that this data can be changed (e.g., additional data can be sent between components), without having an impact on the components and their interfaces. This can, for example, be accomplished by appropriate and systematic use of web services instead of using binary transfer of parameters. This also implies that most external APIs cannot be used directly, since they use an enumeration of primitive data types in their interface.

The third theorem, *action version transparency*, requires that a component can be upgraded without impacting the calling components. This can be accomplished by appropriate and systematic use of, for example, polymorphism or a facade pattern.

The fourth theorem, *separation of states*, requires that actions or steps in a workflow are separated from each other in time by keeping state after every action or step. This suggests an asynchronous and stateful way of calling other components. Synchronous calls—resulting in pipelines of objects calling other objects which are typical for object-oriented development—result in combinatorial effects.

3 TOOL DEVELOPMENT

In order to automatically identify manifestations of violations to the Normalized Systems design theorems, a tool prototype was developed. This tool identifies manifestations of violations to the Normalized Systems design theorems at the API level. The tool was developed in an iterative way using the design science methodology (Peffers et al., 2007). In this paper, we are primarily concerned with *building* and *evaluating* an *instantiation* artifact (March and Smith, 1995).

Since each programming language has its own constructs and syntax, different violations are possi-

ble in different programming languages. We decided to focus on the Java programming language since it is a very relevant language within an enterprise context (cf., Java EE). The tool consists of two main components, `NSTVdoclet` and `NSTVdetect`, that have to be run in succession.

The `NSTVdoclet` component is written as a custom doclet to `javadoc`. The `javadoc` tool is part of the Java 2 SDK. By default, it generates documentation in HTML format of the API of a Java application. The `javadoc` tool is, however, easy to extend by creating custom doclets that provide output in an alternate format. It was decided to develop a custom doclet for `javadoc` that would write selected information obtained by `javadoc` to a temporary database. This method has three main advantages. First, it allows us to reuse the source code parsing algorithm of `javadoc`. Second, most Java applications ship with an `ant` build file that allows to automatically generate the API documentation for the application using `javadoc`. In that case, it is quite easy to specify in the build file that our custom `javadoc` doclet must be used. Third, it is more efficient to parse the source code only once, and have the `NSTVdetect` tool make use of the already parsed information that is available in the temporary database. Since we do not require all the information that is obtained by `javadoc`, this information is filtered and only the information required by `NSTVdetect` is written away in a temporary database.

The `NSTVdetect` component processes the information in this database and analyzes it to identify manifestations of violations to the Normalized Systems design theorems. Hence, we needed to define which violations will be detected by this tool. To this end, we determined how Java applications could violate the Normalized Systems design theorems. In this first iteration, we distinguish between three violations that may occur in Java applications. Each of these violations are related to one or more of the Normalized Systems design theorems. In fact, an $N - N$ relationship exists between the violations and the Normalized Systems design theorems: a single design theorem can be violated in several ways, while a single violation can refer to more than one theorem. Although the current list of violations is not exhaustive, it includes common violations against the Normalized Systems design theorems and covers all four design theorems. This list can be further expanded in the future. As such, the current list represents a lower bound of the violations to the Normalized Systems design theorems that exist in Java applications. For each of these violations, a separate `NSTVdetect` module was developed. Each module analyzes the internal representation of the source code for manifestations of a specific violation in Java applications. We will now discuss these violations and how they are detected by each module in more detail.

3.1 Import Multiple Concerns Violation

A class is a basic module in the Java programming language. Java classes can import and use functionality from external technology environments and packages by using the `import` instruction. This may introduce dependencies on these external technologies in an implicit way. The Normalized Systems approach considers the use of an external technology in a class to be a separate concern, since the external technology can evolve differently from the background technology environment of the class (Mannaert and Verelst, 2009). The separation of concerns design theorem requires that each change driver or concern is isolated from other concerns, so that each concern can evolve independently. This means that each module should contain only one change driver. The separation of concerns theorem therefore implies that a class should refer to at most one external technology. Otherwise, combinatorial effects are introduced in the design.

The *Import Multiple Concerns Violation* module determines which concerns are used by each class based on the imported libraries (using the `import` statements in Java). Before running the analysis, the researcher must define which concerns are present in the application, as well as which libraries fall under each concern. Based on this definition, it is determined how many different concerns are combined in each class. As stated before, a class should not address more than one concern. Depending on the application, one concern could, for example, be the use of the Java Swing packages for the graphical user interface, while a second concern could be the use of the Java JDBC packages to support database access. According to the separation of concerns theorem, both concerns should not be combined in a single class. This is consistent with the concept of multi-tier architectures. Another concern could be the use of another application, such as Cocoon to provide a web-based user interface.

3.2 Primitive in Interface Violation

This violation is related to two Normalized Systems design theorems, namely data version transparency and action version transparency. The use of primitive data types or the `java.lang.String` class in the interface of a method constitutes a violation to both design theorems. We illustrate this with an exam-

ple. Consider a method that allows the user to search for a specific string in a set of files and that takes a single parameter of the type `java.lang.String` that specifies the text to search for. Next, assume that the developers want to extend the search functionality in the future by allowing users to make use of regular expressions. In other words, the method should support searching for regular texts as well as regular expressions. In that case, the interface of the method should be extended with a `boolean` variable to indicate whether the string is a regular expression. This will, however, affect all other methods calling the search method. The method is not action version transparent since the interface of the method will change, and it is not possible to upgrade to a new version of the method without having an impact on the rest of the system. The data is not data version transparent either, since its structure does not allow to send additional data without any additional changes to the system. To resolve this issue, it is better to encapsulate the search parameters in a new class `SearchConfiguration` that can be extended with additional fields as new functionality is added to the search method. The default constructor of the `SearchConfiguration` object should assign a neutral value to newly added parameters (e.g., to indicate that a search string is not a regular expression). By calling the appropriate `set` method, the default settings can be overwritten. Future changes would then have no effect on methods calling the search method. This solution would be compliant with the data and action version transparency theorems.

The implication of both theorems is that the interface of methods should not contain any primitive data types (or objects from the `java.lang.String` class). The *Primitive in Interface Violation* module inspects the interface of each non-private method and determines whether the interface includes one or more primitive data types or the `java.lang.String` class.

3.3 Custom Exception Violation

The Java programming language provides the exception mechanism to handle errors that occur during the execution of a method. If an exception is thrown by a method, the calling method must process this error, either by catching and handling the error internally, or by throwing the exception further upward the stack. Unfortunately, this is a violation of the separation of states theorem which requires that state must be kept upon return of a method call. This prohibits that a calling method must be able to react to all possible error states of a method. Instead, this should be handled by a separate and dedicated module (Mannaert and Verelst, 2009). Otherwise, combinatorial effects occur in the design. Consider, for example, a method that is called by N different methods in the application. If the developer working on this method decides to introduce a new error state by having the method to throw a new exception, this has an impact on the N methods that call this method, since they are forced by the Java environment to catch or throw this exception. Hence, the error handling takes place in N different places. Since N becomes unbounded over time, the impact of this change will increase over time, thereby resulting in a combinatorial effect.

The *Custom Exception Violation* module therefore determines which and how many custom exceptions are thrown by all methods. We consider the use of standard Java exceptions (e.g., `java.lang.Exception` and `java.io.IOException`) to be acceptable, since they are related to the background technology being used. Even in this case, the use of these exceptions should be kept to a minimum. The use of custom exceptions should be avoided, since such errors should be handled in a stateful way.

4 CASE STUDY

In order to test this tool, we performed an investigation of *JabRef*.[1] JabRef is a bibliography reference manager that can be used to edit BibTeX files and is written in Java. We focused on this application for a number of reasons. First, it is distributed under an open source license, thereby providing us with access to the source code of the application. Second, the application represents a moderate development effort. It is not too small to be disregarded as a toy example, and is not too large and complex to complicate the evaluation of our tool. Third, the application is quite popular and widely adopted. The first stable version was released in November 2003. The latest stable version of JabRef that was available to us was version 2.5 and consists of 487 classes, 5,988 methods, and 98,982 LOC.

4.1 Import Multiple Concerns Violation

As mentioned in Section 3.1, we must first specify which concerns are present in a given application. For JabRef, we identified 13 different concerns. A list of these concerns and the packages that fall under each concern are displayed in Table 1. All classes and packages belonging to the `net.sf.jabref.*` package were considered part of the application itself and

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

Table 1: List of concerns identified in JabRef.

Concern	Description
Java Swing	`java.awt.*`, `javax.swing.*`
Java Beans	`java.beans.*`
Java IO	`java.io.*`, `java.nio.*`
Java Net	`java.net.*`
Java SQL	`java.sql.*`
Java XML	`javax.xml.*`, `org.w3c.dom.*`, `org.xml.sax.*`
Java Plugin	`org.java.plugin.*`
antlr	`antlr.*`, `org.antlr.*`
glazedlists	`ca.odell.glazedlists.*`
jgoodies	`com.jgoodies.*`
ritopt	`gnu.dtools.ritopt.*`
microba	`com.michaelbaranov.microba.*`
jempbox	`org.jempbox.*`
pdfbox	`org.pdfbox.*`

Table 2: Import Multiple Concerns Violations.

Number of		
Concerns	**Classes**	**Percentage**
0	306	31.2%
1	320	32.7%
2	228	23.3%
3	86	8.8%
4	34	3.5%
5	6	0.6%
Total:	*980*	*100.0%*

were therefore not considered a separate concern. The results of the analysis are shown in Table 2. It shows that 626 out of 980 (63.9%) classes include at most one concern. Consequently, 36.1% of the classes address two or more concerns and therefore represent a manifestation of the *Import Multiple Concerns Violation*. Six classes even combine five different concerns. A more in-depth analysis showed that these classes all combined the glazedlists, Java Swing, Java IO, and Java Net concerns with either the jgoodies or Java Plugin concerns. Although most of these concerns are related to the default Java SDK API, it does create dependencies on different packages within the API. This data also suggests that file system functions (Java IO and Java Net) are combined with user interface functions (Java Swing). This may neglect the concept of multi-tiers and would therefore require attention in a further screening of the source code.

4.2 Primitive in Interface Violation

The results for the *Primitive in Interface* analysis are shown in Table 3. It shows that 4170 out of 5988 (69.6%) methods do not contain any primitive data types or the `java.lang.String` class in their interface. Further analysis showed, however, that 2578 methods do not take any parameters and therefore require no input. In the next step in our analysis, we excluded those methods from the analysis and focused on those methods that do require input parameters. Results showed that 1592 out of 3410 (46.7%) methods only accept objects in their interface. Consequently, 1818 out of 3410 (53.3%) methods represent manifestations of the *Primitive in Interface Violation*.

Table 3: Primitive in Interface Violations.

	All methods		Methods with parameters	
Violations[a]	**n**	**%**	**n**	**%**
0	4170	69.6%	1592	46.7%
1	1334	22.3%	1334	39.1%
2	294	4.9%	294	8.6%
3	132	2.2%	132	3.9%
4	34	0.6%	34	1.0%
5	18	0.3%	18	0.5%
6	6	0.1%	6	0.2%
Total:	*5988*	*100.0%*	*3410*	*100.0%*

[a] Number of primitive and `java.lang.String` data types used in interface of each method

4.3 Custom Exception Violation

As mentioned in Section 3.3, we consider the use of standard Java exceptions to be acceptable, since they represent the background technology being used. This means that all `java.*` and `javax.*` exceptions were ignored in this analysis. The results for the *Custom Exception Violation* module are shown in Table 4. It shows that 5828 out of 5988 (97.3%) methods do not throw any custom exceptions. The other 160 methods throw at least one custom exception. Our data further shows that 5464 methods do not throw any exceptions. If we only consider those methods that actually throw one or more exceptions, it can be seen that only 364 out of 524 (69.5%) methods do not use any custom exceptions, while the other 160 (30.5%) methods do. These 160 methods therefore represent manifestations of the *Custom Exception Violation*.

5 DISCUSSION AND CONCLUSIONS

In this paper, we have explored the feasibility to automatically identify manifestations of violations to the Normalized Systems design theorems. To this end,

Table 4: Custom Exception Violations.

Violations[a]	All methods		Methods with parameters	
	n	%	n	%
0	5828	97.3%	364	69.5%
1	124	2.1%	124	23.7%
2	16	0.3%	16	3.1%
3	20	0.3%	20	3.8%
Total:	*5988*	*100.0%*	*524*	*100.0%*

[a] Number of custom exceptions thrown

we developed a prototype of a tool to analyze Java applications This tool focuses on violations to the Normalized Systems design theorems at the API level. A first contribution of our paper is that we have shown that it is indeed possible to detect manifestations of violations to the Normalized Systems design theorems in an automated manner. A second contribution is that we have identified three violations to the Normalized Systems design theorems that may occur in Java applications.

Since this tool is still a prototype, we acknowledge several limitations with respect to our findings. The results provide a first-cut and rough assessment of violations to the Normalized Systems design theorems. This assessment can increase awareness about—and give a first impression of—the code quality of an application with respect to evolvability. The violations identified by the tool should at the moment be considered a lower bound for the existence of combinatorial effects, since the tool does not analyze the source code for all potential sources of combinatorial effects.

Given the current limitations of our tool, we do not want to make any claims with respect to the quality of JabRef. Instead, the results obtained in the case study will be used to further refine our tool.

In addition, our results suggest that although the automatic identification of manifestations of violations to the Normalized Systems design theorems is feasible, an additional manual inspection of the source code is required. This inspection provides more insight into the seriousness of the issues identified in the analysis. In practice, some trade-off will need to take place to judge whether the additional effort of containing combinatorial effects is warranted by the likelihood that a future change would manifest itself. However, such decisions should be carefully considered and developers should be aware that not adhering to the Normalized Systems design theorems may have a negative impact on the evolvability of the software.

In future research, we intend to further develop this tool to increase its ability to automatically detect manifestations of violations to the Normalized Systems design theorems. To this end, we will further extend our list of violations to identify a larger number of violations to the Normalized Systems design theorems. Our tool will therefore be expanded with additional modules to test for the manifestation of these new violations.

REFERENCES

Belady, L. and Lehman, M. M. (1976). A model of large program development. *IBM Systems Journal*, 15(3):225–252.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Lehman, M. and Ramil, J. (2001). Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11:15–44.

Mannaert, H. and Verelst, J. (2009). *Normalized Systems—Re-creating Information Technology Based on Laws for Software Evolvability*. Koppa, Kermt, Belgium.

Mannaert, H., Verelst, J., and Ven, K. (2008). Exploring the concept of systems theoretic stability as a starting point for a unified theory on software engineering. In Mannaert, H., Ohta, T., Dini, C., and Pellerin, R., editors, *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA 2008)*, Sliema, Malta, October 26-31, 2008, pages 360–366, Los Alamitos, CA. IEEE CS Press.

March, S. T. and Smith, G. F. (1995). Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77.