Normalized Systems: An Assessment of Evolvability Based on Metrics

Ricardo Coelho
Instituto Superior Técnico, Universidade de Lisboa, Portugal
ricardo.miguel.ferreira.coelho@tecnico.ulisboa.pt

ABSTRACT

Evolvability is a fundamental quality of systems that is valued across all areas of software development. Over the years, techniques have been suggested to achieve this quality, with disputable results mostly because they require a great deal of developer discipline, an appropriate architecture, or because they require embracing complex new frameworks. Normalized Systems (NS) Theory posits that software evolvability rests on four well-defined principles that can be effectively achieved by systematically reusing design and architectural knowledge through code generation. Despite their promising capabilities, NS lack a proper evaluation of their applicability to real software projects, specifically with regards to the evolvability of the applications they produce.

In this work, we overview the concept of evolvability and the existing approaches to achieve it, also identifying the existing gaps in the literature regarding the evolvability assessment of applications developed using NS and other approaches. We then proceed to define a case study consisting in developing a web based information system using NS tools, a traditional approach, and a Model Driven Engineering (MDE) approach. Finally, we evaluate the implementations evolvability using static code analysis in the form of code metrics that measure evolvability characteristics, and compare their results. The results show that NS tools indeed produce highly stable and evolvable applications, even when compared to the remaining approaches. However, they were proven to not be perfect, and several disadvantages are identified.

1. INTRODUCTION

Software evolvability has been defined as the ability of a system to accommodate requirements changes throughout its lifespan at the least possible cost, while maintaining architectural integrity [46]. This quality has become a central concern in the software life cycle [18]. It has been estimated that low software evolvability can increment costs by more than 30%, and increase development effort by 25-36% [35]. Enterprise systems that are expected to have a long lifetime (of 20+ years), will inevitably undergo significant changes to their requirements and functionality, ranging from technology changes to system merging and migration, that will degrade their overall structure and hamper evolvability [11, 50]. Experiments show that about 75% of all the defects identified during code reviews have an impact on evolvability rather than functionality [25]. Therefore, most companies resist to changes in their software, by not implementing the necessary modifications, or by completely scraping the current system and building a new one according to the most recent requirements [12].

Even though evolvability is undeniably essential to the long-term success of a system, it is often neglected during the development process. Indeed, its implementation is complex and costly [7]. In order to achieve evolvability, software has to be built with enough robustness to support a set of anticipated changes [53]. This robustness however represents a big investment up-front, since companies will be designing and implementing structures based on hypothetic future requirements [53]. Since companies seek to maximize the profit of software development, and tend to minimize the effort to implement the required functionality, this results in systems that are not resilient, and quickly their overall structure will degrade once change inevitably needs to be introduced [38]. Degradations will stack up and, eventually, the system will become too complex to manage [28]. This scenario is typical in small outsourcing companies for example [27], that are unsure if they will maintain the system in the long term and, therefore, have little motivation to invest in the intrinsic quality characteristics of the software they produce.

Approaches have been proposed to decrease the investment and time necessary to develop evolvable systems, such as Design Patterns (DP) [54], Software Architecture (SA) [6], or MDE [15]. Unfortunately these techniques still do not reduce evolvability problems effectively, due to human errors, overly complicated or vague procedures, or simply lack of adherence to these approaches [53]. NS Theory argues that evolvable software systems rest on four well-defined design principles for evolvability that are based on the concepts of systems stability [34]. The adherence to these principles generates systems that are extremely fine-grained at the modular level, composed by dozens of structures [45]. This extreme modularity will isolate the various tasks and data, preventing the propagation of changes [12]. However, to achieve evolvability, these principles have to be applied from the start of the software’s development, and a degree of cognitive overload and discipline can potentially lead companies to abandon the approach. With this in mind, the creators of NS propose that the four principles should be obtained by code generation, using tools entitled Normalized Systems Expanders (NSX), in order to spare developers from all the complexity of NS theory, and to eliminate human errors [54].

Even tough NS posits that it can produce highly evolvable systems, there is no evidence in literature that backs up these claims. In fact, there is a scarcity of literature documenting cases of evolvability assessment, with studies only...
focusing on organizational factors such as productivity and effort, or focusing on a subset of qualities and characteristics of evolvability. Furthermore, none of these studies present a concrete and complete evaluation methodology to determine this quality.

To bridge the gap in the study of evolvability, our work consists of (i) defining evolvability as a software quality, presenting a proper decomposition into sub-qualities and characteristics, (ii) studying NS and their tools, analyzing how they function and their principles to achieve evolvability, (iii) developing a controlled case study, defining a sample system complete with domain, requirements, constraints, and extensions, and (iv) developing a complete evaluation methodology, including a robust set of metrics that measure software at various levels and characteristics. The development of an adequate case study and evaluation method is not a trivial task, nor is it the proper definition of evolvability. As a matter of fact, several of these tasks have never been tackled by other researchers, and there are barely any works offering partial guidelines to achieve the goals we proposed.

To assess the evolvability of NS, we derive their evolvability sub-qualities from quantitative data, obtained by analyzing their generated application using a set of code metrics, that enable the measurement of software characteristics. This evaluation is complemented by also repeating the same process using a traditional approach and a modern MDE approach, and comparing the results, as to perceive the place of NS in the current scene.

2. BACKGROUND

This section presents a series of relevant definitions for understanding the problem discussed in this work. Firstly, we will study evolvability on a more detailed level. Secondly, some concepts regarding DP and SA are visited, followed by a study over MDE. Finally key concepts from the NS theory are presented.

2.1 Evolvability

Evolvability is a multi-dimensional software quality defined as the ability of a system to cope with corrective, adaptive, and perfective nature changes in its environment, requirements, and technologies, that may have an impact in terms of structural and functional enhancements [10]. Its importance starts at the beginning of development and extends throughout the software’s life cycle, encompassing long-term, coarse-grained high-level modifications. The evolvability quality can be decomposed into more elemental qualities which help understand the characteristics of evolvable software [9].

According to the definition of evolvability, its two main sub-qualities are maintainability and portability. Maintainability itself is a very broad quality, which can be decomposed into modifiability, analyzability, and testability. Modifiability not only takes into account the capacity of the system to change, but also its capacity to stay stable in face of those changes. Therefore, we can define the following set of elemental qualities of evolvability as Analyzability, Changeability, Portability, Testability, and Stability. There are several characteristics of software that affect its qualities, and thus, affect evolvability [9]. These set of fundamental characteristics are represented in Table 1, along with its effect on the different qualities.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Analyzability</th>
<th>Testability</th>
<th>Portability</th>
<th>Changeability</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Cohesion</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Modularity</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Reusability</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Coupling</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

Table 1: Relationship between the identified evolvability sub-qualities and the characteristics of software that affect them. ● Characteristic increases the quality. ○ Characteristic decreases the quality.

2.2 Design Patterns

Design Patterns identify and abstract key aspects of a common design structure, along with the participating classes and instances, their roles and collaborations, and distribution of responsibilities. DP can be seen as predefined solutions to recurring problems in software engineering, that are: (i) highly reusable, (ii) knowledge preserving, and (iii) enrich system implementations with derivable qualities [49].

The use of DP conveys properties to software that are related with increased evolvability [49, 53], which all stem mainly from the high abstraction and dependency elimination characteristics from DP. Patterns can also be coupled, where structures play roles in more than one DP by referencing common objects and using methods in other patterns [37].

2.3 Software Architecture

Every computing system with software has an architecture. SA is the structure or structures of a system, which comprise software elements, the externally visible properties of those elements, and their relationships [6]. In order to obtain a robust architecture, its construction should take into account the notion of architectural patterns [37, 57], which are a description of elements and relational types, along with a set of useful constraints on how they may be used.

Architecture is also associated with quality attributes, which are properties of the system that affect behavior, design, evolution, and user experience, and combined they dictate the success of an application [50]. Functionality and quality attributes are orthogonal, since the function of a system does not dictate its qualities automatically [6]. The relation between functionality and quality attributes mainly concerns the way functionality interacts with qualities and constrains them.

2.4 Model Driven Engineering

Through models it is possible to analyze a system and plan its construction in an effective and efficient manner, achieving more control over its life cycle, and making it possible to share knowledge more efficiently. Since models contain a big amount of information about systems, it is possible to use them in order to shorten the development time, by using techniques such as code generation or model interpre-
There are two main types of transformations: (i) model-to-text, which are used to produce software elements directly from models [45], and (ii) model-to-model, which translate models into other sets of models that are closer to the solution domain, or that fulfill specific needs [22]. One of the great advantages and promises of MDE is the fact that it provides a systematic means to reuse architectural and design knowledge [10]. MDE enables developers to reuse documented solutions even if they do not directly possess themselves the technical knowledge necessary to apply them. By means of transformations and code generation, these tools can automatically implement patterns that solve specific problems relevant in the context of the application.

### 2.5 Normalized Systems

The NS theory was developed with the intent to provide concrete guidelines for software development to achieve maximum evolvability. The theory spawned from the concept of system stability whereby a system is considered stable, if a bounded set of changes results in a bounded amount of impacts to the system, independently of the system’s operational time period [31, 34].

Whenever effort required to implement changes increases as the system grows, that is an indication that we are in the presence of Combinatorial Effects (CE) [13]. These effects are the consequence of dependencies between modules that should not exist, and result from the failures in encapsulation. Due to these dependencies, changes in a given module will have impacts in other modules whose responsibilities are independent from the original change. As the system is developed, dependencies are introduced in the code increasing the CE thus decreasing the overall stability of the system. A software system is said to be a normalized system if it does not produce CE for a set of anticipated changes.

In NS there are two primitive software entities [53]: (i) data entities, that contain fields and links do other data entities, and (ii) action entities, that represent operations performed over data entities. The later are decomposed into tasks which are a set of instructions that perform a certain functionality, and are distinguished between functional and supporting tasks. These may have different versions, and each task should only contain instructions from its technology environment, allowing each task to evolve independently.

NS are stable for a set of anticipated changes. These changes are regarded as primitive changes, and constitute the fundamental modifications that are possible in an information system [31, 53]. These are the addition of (i) a data attribute or field, (ii) a data entity, (iii) a task, and (iv) an action entity.

Finally, NS postulates that stability rests on four fundamental principles for software evolvability [54, 55], and that by following these principles it is possible to build a system that is resistant to the set of primitive changes. The theorems are as follows:

- **Separation of Concerns**, which states that an action entity can only contain a single task.
- **Data Version Transparency**, which states that data entities need to exhibit version transparency.
- **Action Version Transparency**, stating that action entities need to exhibit version transparency.
- **Separation of State**, stating that the interaction between action entities needs to exhibit state keeping.

### 3. RELATED WORK

To achieve better evolvability in information systems, several approaches have been proposed. Hence, we search and review the current state-of-the-art in the application of these approaches, to understand how they compare between each other, how they are used, how their impacts are measured, and what actual impact they have on the evolvability qualities of the produced systems.

#### 3.1 Evolvability In Architecture And Design

Some studies over DP revealed that not only does the success of the approach depend on the pattern itself, but also on its correct implementation and application [15]. Studies also report that patterns can create instability, especially when coupled to others [5]. However, most patterns seem to be effective in protecting the classes that play their main roles from those instabilities. Some comparative studies also suggest that the use of patterns, when applied to the correct domain, do improve evolvability qualities, especially maintainability [22, 50]. We also identified that well structured architectures do improve evolvability qualities [10, 52]. Regarding specific architectural patterns, there seems to be a preference for layered and component based applications with regards to the qualities of evolvability [17].

#### 3.2 Evolvability In MDE

There is definitely a lack of studies over the software quality benefits of MDE approaches, with surveys referring explicitly the lack of MDE studies in current literature, and even less focusing on qualities related to evolvability [3]. The majority of studies try to understand the impacts of this approach at the organizational and business level rather than the actual effects they have on software [4]. Furthermore, we were not able to find case studies or surveys on comparisons between MDE approaches, only between MDE and traditional ones, and there seems to be no standard in the tools used [55]. This shows there is definitely a need for these types of research, in order to answer these questions. Despite the difficulties mentioned, the studies we found seem to point that MDE has benefits regarding mostly maintainability [20, 25], with one study also reporting an improvement of portability [26].

#### 3.3 Assessment Of Evolvability Qualities

In general, few studies measure all qualities related to evolvability, and even fewer actually mention it. The quality that most papers seem to study is maintainability, and few of them were found presenting conclusions over portability. Furthermore, most of the studies that address maintainability do not actually present a definition. In regards to assessment methods, code metrics are the most used, providing quantitative insight into the code produced, being an evaluation approach less susceptible to cognitive biases than the remaining ones. Despite this, the studies that use them often use a small set of metrics, a set that does not measure all characteristics of evolvability, or code metrics that measure software at only one abstraction level.

#### 3.4 NSX And Cases Of Its Application
NSX are a set of software programs that generate code compliant with the NS theory, in the form of structured software elements. These elements are designed with the four NS theorems in mind, and are meant to be composed with each other as building blocks of an application, being generated using input descriptor text files. The tool will generate the source code by expanding these descriptor files into the design patterns defined for each element [14, 33]. In the current version of the NSX there are five elements that can be generated:

- **Data elements**, separating domain functional data and technical tasks.
- **User connector elements**, linking user interaction with data.
- **Task elements**, representing single encapsulated functional and supporting tasks.
- **Workflow elements**, representing a sequence of tasks triggered by an event.
- **Trigger elements**, providing a way to activate a workflow at a specific time.

Regarding studies about its application, we were only able to find three documented case studies about the tool’s usage and evaluations of its effectiveness [42, 43, 44]. These studies focus simply on development time and perceived developer effort, and none of them ever mention the actual evolvability qualities of the code produced. Therefore, there is a need to fill this research gap by evaluating the actual software produced by NSX in regards to evolvability.

4. **CASE STUDY**

To access the evolvability of an information system developed using NSX, we devised a case study based on the implementation of a web application. The first activity was to define a suitable base system, including its domain and requirements. In order to develop a feasible case study, the system had to be complex enough to exercise a significant amount of features, and had to be somewhat elaborated, while not being overly complicated. The system’s domain and requirements should be easily understood, and its implementation in the various approaches should not be very time consuming and resource intensive, as the system and its extensions would have to be developed more than once. The chosen information system is an application to manage Customers, Loans, Accounts, and Branches. The business rules regarding the information of each entity dictate that a customer number, a customer’s customer number, an account’s account number, a loan’s loan number, and branch’s name, should all be unique. Besides these requirements, we restrict implementations to the main technologies NSX supports, by imposing that the database used should be a relational database, and all the data access, business logic, and server side software, should be implemented using Java.

### 4.1 Requirements And Constraints

This system has a set of requirements that all implementations should abide to, which dictate that the user can create, delete, alter attributes and relationships, list, and view details of Customers, Loans, Accounts, and Branches. Also, to use the application, the user must login, using a combination of **username** and **password**. The business rules regarding the information of each entity dictate that a customer’s customer number, an account’s account number, a loan’s loan number, and branch’s name, should all be unique. Besides these requirements, we restrict implementations to the main technologies NSX supports, by imposing that the database used should be a relational database, and all the data access, business logic, and server side software, should be implemented using Java.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tool must be oriented towards the development of information systems.</td>
<td>Domain</td>
</tr>
<tr>
<td>The tool must be oriented towards the development of web applications.</td>
<td></td>
</tr>
<tr>
<td>The server side of the application must be implemented using Java.</td>
<td>Technology</td>
</tr>
<tr>
<td>The application must persist its data using a relational database.</td>
<td></td>
</tr>
<tr>
<td>The tool must be capable of generating a complete application.</td>
<td>Functionality</td>
</tr>
<tr>
<td>The tool must not generate code based on user-defined templates.</td>
<td></td>
</tr>
<tr>
<td>The tool must have been updated at least once in the last three years.</td>
<td>Relevance</td>
</tr>
<tr>
<td>The tool must not have been discontinued.</td>
<td></td>
</tr>
<tr>
<td>The tool must allow full access to the generated code.</td>
<td>Accessibility</td>
</tr>
<tr>
<td>The tool must have a free version available, or have a free trial that matches the other criteria.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Criteria for choosing a suitable MDE tool for the developed case study, along with their motivation.

### 4.2 Extensions

Besides the described base system, we also devised a set of three extensions to simulate three common modifications to systems. The implementation of each modification should be made branching from the base version, generating three new separate versions of the system, which will then be combined to achieve the final version of the system. The three extensions consist in (i) a new business rule, that dictates the amount property of a Loan should never be negative, and represents the new task anticipated change, (ii) the functionality of searching for a Customer by its customer number property, representing the new action entity anticipated change, and (iii) a new domain entity named Division, containing a name attribute, and one or more Branches, and represents the new data entity anticipated change. We will refer to these extensions as Extension Business Rule (ExBR), Extension Search (ExS), Extension New Entity (ExNE), respectively.

### 4.3 Additional Implementation Approaches

In order to do a more complete evaluation of the feasibility of the NSX tools, we decided to complement the case study with the implementation of the system using a code generation tool based on the MDE paradigm, and a traditional application development paradigm. The use of a second MDE tool allows us to perceive the position of NSX in the current MDE scene, while the use of a traditional approach serves as a baseline for the currently most used method to develop information systems today. The selection process to choose an adequate MDE tool consisted in: (i) defining a set of selection criteria, (ii) finding a group of currently available MDE tools, and (iii) choosing a tool matching the defined criteria. We defined a set of criteria presented in Table 2.
motivated by (i) the tool’s domain, to ensure the chosen tool is suitable for the implementation of the system described, (ii) its technologies, to restrain the chosen tool to the main technologies used by NSX, (iii) its functionality, to level the differences between the features of NSX and the chosen tool, (iv) its relevancy, to ensure it is significantly modern and active, and (v) its accessibility, so we can access and use the source code the tool generates, without restrictions. We searched for MDE development tools, and after applying the criteria, we were left with Generjee, a free tool that allows users to generate full Java Enterprise Edition (JEE) information systems. It is an open source active project, hosted as a web application that allows the creation of systems by describing the domain model of the application using various forms.

5. NSX IMPLEMENTATION

The first step in generating a system using NSX is to translate the existing data model into NS data elements. The only other elements needed are user connector elements, but these elements are generated automatically with data elements, even tough they are distinguished as a different element. The next step is to describe the application using an application descriptor file. Afterwards, we describe the bankcomp component in a separate component descriptor file. Finally we can describe our four data elements. Each element is described in its own data descriptor file. Having described all data elements, the application can be generated through the command line. The architecture of the generated application is depicted in Figure 1. The database tier is generated automatically according to the the domain model. The domain logic tier encompasses the (i) persistence support module, that defines the domain data objects and supports access to the database, (ii) transaction support module, that defines business logic and controls transactions on the access of data, and (iii) remoting support module, that allows the communication between the domain logic tier and the presentation tier. The presentation tier implements concerns regarding the user connector element, and is composed of (i) remoting support module, implementing the communication with the domain logic tier, (ii) session support module, containing the implementation of user sessions, and (iii) user support module, implementing aspects related to user interaction and communication with the client tier. The client tier resides on the web browser of the end user, and is implemented using a Model-View-ViewModel (MVVM) pattern by means of the user connector element. The view provides several page fragments that together compose a web page, and it interacts with the rest of the application through the view model.

Each data element implies the generation of a group of these modules. All these modules make use of the NSX own internal framework to mediate the implementation of the module and the technologies it uses, so that the classes only depend on the NSX internal framework and general specifications.

The implementation of the business rules has to be done manually, by altering the domain logic tier, and inserting code between the anchor tags that are generated along with the application. The ExBR extension was implemented by altering the LoanBean class of the domain logic layer. The ExS extension was implemented by adding a line containing findByCustomerNumberEq between the attributes and options lines of the Customer data element descriptor, and regenerating the application. The ExNE extension involved adding a new data element using a descriptor file.

6. MANUAL IMPLEMENTATION

The manual implementation uses a three tiered architecture conventional in web applications, depicted in Figure 2, and is composed by: (i) database tier, which is a relational database serving as the persistent storage of data, (ii) server tier, which can manipulate data fetched from the database, and provides web services that can be used by clients, and (iii) client tier, which presents content to the user, allowing him to manipulate the system. The server uses a layered architecture, with three layers, which are (i) data access layer, which communicates with the database, (ii) business layer, which performs business logic operations, and (iii) web service layer, which provides Representational State Transfer (REST) web services.

To effectively access the database, we employ an implementation of the Data Access Object (DAO) pattern, which abstracts the access from the upper layers. To implement this pattern we defined an abstract, generic DAO that takes care of basic operations. Then, for every domain object, we define a specific DAO for that object, which extends from the generic one, implementing specific tasks. Next, we define

---

Footnote:

1 http://www.generjee.com/

Figure 1: The domain model of the bank information system used in the case study.
DAO interfaces for each domain entity. Finally, we use the Factory Method pattern to delegate the creation of DAO objects to a Repository. This interface provides a method to create DAO just by receiving its domain class as a parameter, while also enabling the manipulation of database transactions.

The business layer is where the business logic lies, allowing the manipulation of domain entities while enforcing business rules. This layer receives input from the web service layer, and uses that input by issuing database transactions, and accessing data from the data access layer, to perform operations on that data.

The web service layer encompasses infrastructure necessary to expose the server Application Programming Interface (API) to the client, by declaring a series of stateless REST web services. The services are determined by the domain entity in which they are based on, and the plurality of the operations.

The client tier uses the MVVM pattern, where the view is the interface exposed to the user, the view model exposes the data and command objects that the view needs, and the model is the server.

The ExBR extension was implemented by adding the business rule to the business layer methods that manipulate the loan domain entity. To implement the ExS extension, we added a search form into the customer listing view of the client, and modified the services implementation in order to allow the use of a query when fetching the list of customers. The implementation of the ExNE extension consisted in adding a new table to the database, a new domain object class and DAO, and new set of web services related to the entity.

7. GENERJEE IMPLEMENTATION

The first step in the implementation process is to describe the application and its domain using the forms in the Generjee web application. We started by describing the application we wanted to generate, following with the description of the domain model of the application. Having described all the entities, the following action is to generate the application.

The architecture of the generated application is depicted in Figure 4 and is based on JEE having three main tiers, which are the (i) database tier, containing the application’s database, (ii) server tier, containing all server side implementation, and (iii) client tier, containing the browser portion of the application.

The database tier is automatically generated from the
Java Persistence API (JPA) specification of the domain model. The server tier is implemented according to a Model-View-Controller (MVC) architecture, where the model is composed by a data module, containing data access support, and a business module, that consists of the business logic and actions invoked by the controller. The data module allows abstracted access to the database through the use of the DAO pattern, having a generic DAO for the common operations, and specific DAO for queries unique to the data entity. The view portion of the architecture is implemented by pages using PrimeFaces components and themes, while the mediation between the views and the model is made by the Faces Servlet controller. The client tier is composed of pages that are served based on the parsing of the view pages in the server, and interacts with the it through calls to the controller.

In order to implement the business rules defined, we altered the business module, by inserting methods that validate the rules, and calls to that method before saving or updating an entity. To implement the ExBR extension we insert a validation in the LoanBean class before saving or updating a Loan. The ExS extension was implemented by importing the JavaScript Object Notation (JSON) model generated with the application into the Generjee tool, and describing the new entity, regenerating the application afterwards.

8. Evaluation

To evaluate the evolvability of the produced systems, we employ methods of static code analysis, using code metric suites capable of measuring the characteristics of evolvable systems. The purpose of this analysis is to provide quantitative data about the qualities of the produced systems in order to understand if a system developed using the NSX tools is highly evolvable, and how it compares to the other systems.

8.1 Methodology

To evaluate the implementations, for each application we employ the evaluation process in Figure 5. To evaluate the produced information systems, we use two code metric suites designed to analyze systems both at class level and project level. One of the metric suites used in this evaluation is the Chidamber & Kemerer’s class metric suite [17], composed of:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number Of Children (NOC)
- Coupling Between Objects (CBO)
- Response For Class (RFC)
- Lack of Cohesion of Methods (LCOM)

The version known as LCOM4, proposed by Hitz & Montazeri [25], is the one used in this evaluation. The other suite is Brito & Abreu’s Metrics for Object Oriented Design (MOOD) project metric suite [2, 3], composed of:

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Coupling Factor (COF)
- Polymorphism Factor (POF)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Complexity</th>
<th>Cohesion</th>
<th>Monotonicity</th>
<th>Reusability</th>
<th>Coupling</th>
<th>Designability</th>
<th>Acceptable Interval</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>[0, 24, 31]</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td></td>
<td></td>
<td>[0, 30]</td>
<td>lower</td>
</tr>
<tr>
<td>DIT</td>
<td>[1, 4]</td>
<td>middle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOC</td>
<td>[0, 4]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBO</td>
<td>[0, 5]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RFC</td>
<td>[0, 40]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCOM</td>
<td>[1, 45]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MHF</td>
<td>[1, 20]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[12.7, 21.8]</td>
<td>higher</td>
</tr>
<tr>
<td>AIF</td>
<td>[0, 20]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[31, 100]</td>
<td>higher</td>
</tr>
<tr>
<td>MIF</td>
<td>[0, 40]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[20, 80]</td>
<td>lower</td>
</tr>
<tr>
<td>COF</td>
<td>[0, 11.2]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POF</td>
<td>[2, 7.96]</td>
<td>lower</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Relationship between the chosen metrics and characteristics of evolvability. Metric value increases characteristic. Metric value decreases characteristic.

The version known as LCOM4, proposed by Hitz & Montazeri [25], is the one used in this evaluation. The other suite is Brito & Abreu’s Metrics for Object Oriented Design (MOOD) project metric suite [2, 3], composed of:

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Coupling Factor (COF)
- Polymorphism Factor (POF)
Table 4: Comparison between the characteristics of evolvability of each system implementation. ● Good. ○ Tolerable. ◦ Dangerous.

These metrics have a series of correlations with the characteristics that affect the evolvability of systems. Table 4 summarizes the aspects discussed, indicating the relationships between the metrics defined and evolvability characteristics, along with the recommended interval of values where they should lie, and the limit to which an implementation should try to aim at.

8.2 Results

The results obtained allow us to infer the characteristics of the general systems. Using these characteristics and their relationships to the evolvability qualities, we can compare the values of the various qualities for each system using Table 4.

The NSX generated code has indeed a good evolvability. It also shows that, as declared by the NS theory, stability is a core quality for NS, since they clearly try to take coupling to a minimum in the majority of the system, and put great effort into the encapsulation of their classes, as to have clearly defined and stable interfaces. They also go so far as to almost completely avoid inheritance, one of the main mechanisms of Object Oriented Programming (OOP), to avoid CE through it. However, there is clear evidence that they rely on the existence of a handful of centralized, large classes, with high complexity and coupling. Therefore, evolutions that involve these classes have to be taken very seriously, as they can compromise a significant part of the system.

Regarding the other systems, the manual implementation shows great contrast with NSX, as it focuses on other methods to achieve evolvability, mainly reusability by inheritance. Essentially, the manual version suffers mostly from problems related to encapsulation and coupling, and shows less stability than the NSX version. However, it does not rely on monolithic classes. Generjee follows an approach more akin to the manual one, but being more careful with its approach, mainly regarding inheritance and coupling relationships. It achieves satisfactory values in most characteristics, and it tries to balance the various qualities. This implementation has good evolvability, regarding the majority of the system, the NSX implementation is still more evolvable. However, if the intent is to make several customizations that might involve the monolithic classes, Generjee may take the upper hand, since it does not have that issue, and its proximity to the manual method might make it more familiar to developers.

9. CONCLUSIONS

Evolvability is a software quality that plays a major role in a system’s success, but current approaches to achieve it are still far from ideal. NS Theory and its NSX tools were proposed, defending that the solution lied in four well-defined theorems and in knowledge reuse through code generation. Although promising, NS lack a proper evaluation of their capabilities and applicability. With a concrete and comparative validation of the evolvability of applications generated with NSX, both developers and organizations could achieve more informed decisions about the practices they employ to produce long-lasting information systems. This in mind, we set out to properly define evolvability, and to devise a case study and evaluation methodology that could effectively measure the evolvability of information systems built using NS, and to compare that evolvability with the one resulting from other approaches, namely, a traditional approach, and an MDE approach using modern tools.

As our review of literature revealed, there is not enough work made in this field, and the current research scene lacks studies over evolvability and its assessment. Besides, no studies that prove the capability of NSX to produce evolvable systems could be found, nor did there exist studies that compared MDE tools in relation to this quality. Furthermore, there is no systematic and complete way to assess the evolvability of a system, regardless of its implementa-
tion approach. To solve this problem, we defined a case study using a sample, controlled system, adequate for this task. We clearly defined its domain, requirements, and constraints, taking into account the inner workings and purpose of the NSX tools, in order to maximize its validity. As a complement to the base system, a set of extensions was devised, to observe their impact on the systems’ evolvability. The selection of an adequate MDE tool to use as one of the approaches was also taken seriously, by carefully reviewing existing tools according to a robust set of criteria, motivated by (i) accessibility, (ii) relevance, (iii) technology, (iv) functionality, and (v) domain, in order to achieve a fair comparison. The implementations of the system were attentively carried out, by adhering to best practices, conventions, and development guides as to attain good evolvability.

In order to evaluate the implementations, we employed static code analysis methods, defining a robust set of code metrics based on existing literature, identifying the specifics of their measurements, the characteristics they measured, and the values recommended by researchers. We then employed that methodology, obtaining a set of results that indicated NSX are indeed capable of generating evolvable applications, with focus on stability. By comparing the results, we also perceived that the traditional approach focuses on different characteristics to achieve evolvability, especially reusability, and employs strong use of inheritance mechanisms. The utilized MDE tool, Generjeye, was revealed to also produce satisfiable levels of evolvability, residing between the other two approaches, achieving a solid balance of characteristics. Nonetheless, NSX were revealed to not be perfect, with the results revealing the existence of a handful of monolithic classes, heavy on complexity and coupling. A surprising, almost complete, lack of inheritance use was also perceivable from our evaluation. We can conclude that although the majority of the NSX generated system displays high levels of evolvability, mainly due to its stability, customizations that can affect the monolithic classes can turn out to be significantly difficult and dangerous, possibly hampering the stability on which NS is founded.

REFERENCES


