TOWARDS MEASURING SOFTWARE REQUIREMENTS VOLATILITY: A RETROSPECTIVE ANALYSIS

Shouki A. Ebad[1]

[1]Faculty of Computing & Information Technology,
Northern Border University,
Saudi Arabia

Email: shouki.abbad@nbu.edu.sa[1]

ABSTRACT

Requirement management (RM) is a fundamental activity in requirements engineering. It keeps track of all the requirements changes that would cause errors or resulted in software delays or cost overruns. When requirements have many changes over time, they have a tendency to be highly volatile. This volatility depends on several factors such as organizational complexity, process maturity of the company, and development phase. Managing the requirements quantitatively by metrics is a good way to understand whether RM is efficient or not. In this paper, we propose a new metric to measure the requirements volatility of object-oriented systems in terms of use cases; we use retrospective analysis that examines the amount of change applied in successive versions of a software product. We theoretically validated our metric through a set of prominent mathematical properties. We also empirically validated our metrics using three versions of an open source project, JHotDraw. Measurements of the metric were shown to be consistent with previous measurements of the JHotDraw versions conducted at the architecture design level. The study results in a foundation for further empirical retrospective studies of the requirements properties.

Keywords: requirements engineering, requirements volatility, requirements management, software metrics, use cases, software evolution.

1.0 INTRODUCTION

Software requirements describe the services provided by the system and its operational constraints [1][2]. These requirements reflect the needs of system stakeholders (including paying customers, users and developers). The process of finding out, analyzing, documenting, and checking these services and constraints is called requirements engineering (RE) [1][2][3][4]. This process deals with specific requirements activities, elicitation, analysis, specification, communication, validation and evolving of the requirements [2][5][6]. RE is a critical stage of the software process as errors at this stage inevitably lead to problems later in the system design and implementation; the cost of fixing these errors in initial stages is lower than fixing them in the later stages of software development [1][2][4][7]. Many of these errors are caused by changes in the requirements. Requirements are subject to change to reflect changing stakeholders' needs or changing environment, business plans, and laws [5][8][9][10]. Successful software systems evolve as the environment in which these systems operate changes and stakeholder requirements change [6]. Thus, changes to requirements documentation need to be managed; the process that keeps track of all requirements changes and configurations is called requirements management (RM) [2][3][11]. It is clear that RM is a fundamental activity in RE. However, it has become a challenge because requirements change has been reported as one of the main factors affecting the project delays or project cost overruns, if not causing complete failure [5]][9][12][13][14][15].

Typical changes to requirements include, adding, deleting or modifying requirements, and fixing errors [16][17]. When requirements have many changes over time, they tend to be highly volatile [18][11]. From an evolution perspective, volatile requirements are likely to change after the system has been become operational or during the system development process. An example, is the requirements resulting from government healthcare policies [2]. These requirements must evolve to reflect the changed view of the system in development. Furthermore, the

volatility on requirements depends on several factors. For instance, one might be organizational complexity, the process maturity of the company, the phase of the life cycle, the volatility of the market and so on [1][3][18][19]. As a result of an increased number of interacting components, the more complex the system or product developed becomes, the higher is the volatility [11]. In many cases, requirements changes might produce conflicts, which can be interpreted as changes and errors that need measurement. Measures give a status of the software project to the development team, which increases the probability of producing a satisfactory result in the end product [1][2][11]. Thus, managing the requirements quantitatively by metrics is a good way. We can understand whether requirements management is efficient from the requirements metrics result [10].

Although the object-oriented (OO) paradigm has grown in influence and use over the last few years, most of the OO measures extensively focused on internal source code attributes such as size, cost prediction, cohesion, coupling and structure [20]. Other attributes, like those related to the product requirements, have not received much needed attention. To the best of our knowledge, there is no metric defined in terms of use cases (UCs), which were first introduced by Jacobson [20]. UCs are a scenario-based technique for requirements elicitation; they describe how a user interacts with the system by defining the steps required to accomplish a specific goal. The set of UCs represents all of the possible interactions to be represented in the system requirements [2][20]. They have now become a fundamental feature for describing OO systems models. Herein, UCs can be used to measure the requirements volatility during RM activity. According to Jacobson [20], the functionality (i.e. services) that users require of the system is documented in UCs. The objective of this research is to propose a new metric to measure the requirements volatility of OO systems in terms of UCs that considers more factors than what have been used in the existing measures. To evaluate requirements volatility and provide information about the requirements and its evolution, enabling the monitoring of trends in software evolution, retrospective analysis is used by examining the amount of change applied in successive versions of a software product. Compared to the existing volatility metrics, our metric is validated theoretically against theoretical properties and empirically using an open source project. The rest of this paper is organized as follows. Section 2 presents a review of existing requirements volatility measures. Section 3 describes the retrospective analysis compared with the predictive one. Section 4 introduces our new metric for measuring requirements volatility. The validation of the proposed metric against theoretical properties is discussed in Section 5. Section 6 discusses the empirical validation and the application to a real-world case study. Section 7 presents possible threats to the validity of this study. Finally, Section 8 summarizes the paper and gives plans for future work.

2.0 LITERATURE REVIEW

This section presents, in chronological order, a review of the research works that have been done in the area of measuring software requirements volatility. Sherif [[16]] proposed a risk management metric that deals with the stability of requirements throughout phases of software development. By example, the metric gave the incremental risk for every development phase and also the total cumulative risk as the project progresses from phase to phase. Malaiya and Denton [7] analyzed the influence of changes in a program when testing has been initiated. They examined the effect of replacing a component with another component of the same size, as well as general cases when software is added, deleted, and modified. All the results show that changes have a greater impact on defect density when they occur closer to the end of the testing effort (i.e. close to the version date). Wang and Lai [10] discussed the method of requirements management for the increment development model, the goal of the management and the structure of data collection. They also presented a metric to measure the stability of requirements; the metric depends on the statistical process control (SPC) technique often used in monitoring, controlling and improving process stability [22]. Nurmuliai et al. [9] presented a qualitative method to characterize and evaluate the requirements change problems throughout the system development process. They also developed a taxonomy to classify requirements change and the causes of these changes. Their findings revealed that the main causes of requirements volatility were changes in customer needs (or market demands), developers' increased understanding of the products, and changes in the organization policy. They discovered that the rate of requirements volatility was high at the time of requirements specification completion and while functional specification reviews were conducted. Selby [23] investigated the principles for measurement-driven dashboards for development and management of large-scale projects. He focused on software requirements metrics from such dashboards where the number of requirements was defined to be the number of "shall" statements in the requirements specification documents. Loconsole and Börstler [13] quantified requirements volatility through changes to the UC models in a case study in the automotive industry.

In that research, changes were measured as changes to the UC diagram. Their paper is based on a single project comprising fourteen UCs; this led the authors to consider their results as preliminary. The goal of the study in Ali [[1]] was to analyze requirements metrics that measure traceability, completeness, volatility, and size. He also studied the existing automated requirements tools, including Automated Requirements Measurement Tool – ARM, IBM Rational Requisite Pro, Dynamic Object Oriented Requirements System, and Requirements Use Case Tool. Kulk and Verhoef [24] proposed a mathematical model to identify the requirements volatility danger zone of IT projects. With that model, it is possible to calculate a project's tolerance for volatility based on size estimates at different moments in time and the duration between them. They also derived two volatility ratios from this model to express how close the volatility of a project has approached the danger zone when requirements volatility reaches a critical failure rate. Hou [11], proposed a set of RM metrics, but he implemented three because of the time factor, requirements growth (size) that indicates the amount of requirements, requirements acceptance that indicates the acceptance degree by measuring review results as projects are running, and requirements volatility, which means the rate of change of requirements. Besides, they did not validate the metrics theoretically; they only employed the model presented by Canfora and Cerulo [37] to present the metrics proposed. Markopoulos et al. [25] introduced a new project management model based on requirements management tracking using a set of metrics that analyze the requirements evolutional behavior against weighted project implementation phases, weighted project functionality, and weighted project goals and expectations. This weighted requirement based project tracking process is supported by a project tracking analysis model combining a number of metrics that result in the identification of a single volume indicating the progress of the project. The work [14] aimed to evaluate impacts to the systems engineering effort, measured in terms of labor hours, by investigating the causes and effects of requirements volatility in large-scale systems using an extension to COSYSMO (Constructive Systems Engineering Cost Model), a generally-available parametric systems engineering cost model. Obana and Hanakawa [26], proposed a new metric for meeting quality on a software requirement analysis phase. Their original basic idea was "high quality meetings lead high quality software requirements". A feature of the metric is to measure when and who speaks at meetings. Contexts of speaking at meetings are not a target of the metric. As a result, the authors automatically extracted vague discussion and suspicious discussion in quantitative analysis using the metric. However, if topics did not appear in meetings, software faults using the metric cannot be predicted. Peña and Valerdi [19] discussed a number of factors (technical, organizational and contextual) that could lead to requirements volatility being identified, ranked, and depicted in a causal model diagram. Besides, survey results and workshop discussions resulted in a set of observations regarding the expected level and impact of requirements volatility across the system life cycle. Abd Elwahab et al. [18] described several aspects of requirements volatility including causes, measurements, and impact of requirement volatility on software life cycle development. To manage requirements volatility, they proposed a framework that contains four major phases (elicitation and analysis, specifications validation, requirements volatility and change management).

In conclusion, our literature survey reveals that the existing requirements volatility measures and evaluation approaches have some limitations. For instance, metrics proposed by Selby [23] oversimplifies the requirements as it is based on counting the number of "shall" statements. The methods used in works by Kulk and Verhoef [24] and Qazi et al. [14] are not simple; they are based on size estimates, project cost and duration, and financial background. In Loconsole and Börstler [13], some exiting measures related to the requirements volatility were applied on a very small-sized project comprising a few UCs. The used measures were not related to requirements artifacts like changes in source lines of code (SLOC), change error, change effort and so on. The metric of Hou was applied to meet the goals of the Volvo Group. The source code artifacts (not requirements artifacts) played an important role in other studies, Kulk and Verhoef [24] and Loconsole and Börstler [13]. Obana and Hanakawa are closer to the time management technique than requirements metric. Also, some observations with regard to the validity of the proposed volatility metrics were revealed. Although the definition of valid measures requires that they be theoretically validated [18][28], the designers of the metrics proposed in all surveyed works did not validate their proposed metrics against theoretical or mathematical properties and few of them validated their metrics empirically. To address the above issues, we use retrospective analysis to develop a new metric to measure the requirements volatility based on UCs. We theoretically validated our metric against applicable theoretical properties and experimentally against a real-world open source system; each system version has tens of UCs.

3.0 RETROSPECTIVE ANALYSIS

From a software evolution perspective, artifacts (e.g., requirements documents, architecture models, design diagrams, source code, etc.) can be evaluated by using either predictive analysis or retrospective analysis [27]. Predictive analysis tries to anticipate how well a system artifact (in our case, requirements) will perform in the future. It provides prediction of the target property and not the actual value. Such an analysis is useful because it helps save cost and time by concentrating on the key aspects of the user and systems requirements in advance. However, performing this analysis is not easy because the possible changes that will be made to the system's requirements are not known beforehand. Retrospective approach looks at successive versions of the system artifact to analyze how smoothly the evolution took place. Intuitively, our goal is to see if the system's requirements remained intact throughout the evolution of the system, that is, through successive versions of the software. We call this intuitive idea "requirements stability". Usually, the approach relies on comparing properties from one version of the system to the next. This implies that some requirements information must be kept for each version. For example, we might compare the "shall" statement in successive versions. If such statements remain substantially unchanged, we can conclude that it was stable requirements that supported evolution well. The general advantages of this analysis including (1) to evaluate the requirements volatility empirically, (2) to calibrate the outcomes of predictive analysis and (3) to predict trends in the system's evolution. Such predictions can be valuable for planning the future development of the system. As an example, a software project manager may use previous evolution data for anticipation of the resources needed for the next version of the system, identification of the components most likely to require attention, identification of the components needing replacement, or deciding if it is time to retire the system entirely.

4.0 MEASURING REQUIREMENTS VOLATILITY

In this section, we first identify types of change that affect requirements volatility, and based on these types, we then propose a requirements volatility metric.

4.1 Identification of Volatility Aspects

According to Sherif [16] and Stark et al. [17], the change types on requirements are three:

1. Requirement addition: adding a requirement to make up for the omission or meet the customer's requirement.
2. Requirement deletion: deleting or removing existing requirements from the business strategy or the requirements redundancy.
3. Requirement modification: modifying requirement owing to technical restriction or design improvement.

In fact, a requirements volatility problem appears when any of the above types occur after the basic set has been agreed to by both clients and developers. An accurate measurement is useful for preventive and controlling requirements volatility [12]. To this end, we propose a new metric to measure the volatility in terms of UCs of the system. UCs are more suitable to be used in capturing the software requirements than using other techniques such as counting "shall" statements as in some previous works. UCs can be seen as one way of specifying the services that users require of the OO system [[20]]. Moreover, they are highly accepted in industry. In this research, the number of requirements is defined to be the number of UCs in the requirements specification documents, i.e. each UC would count as one requirement. In this context, there are three types of change that can be observed on UCs when two versions of the same software are compared:

1. *New*: A UC that did not exist in version i has been added to version $i + 1$.
2. *Removed*: A UC that exists in version i has been removed in version $i + 1$.
3. *Modified*: A UC that exists in version i has been modified in version $i + 1$.

As our aim is to measure the requirements volatility, such a measure would depend on the changed types and find volatility by counting the changed UCs between version $i + 1$ and version i. We consider the requirements of subsequent version (version $i + 1$) completely volatile if all of its UCs have been changed and completely

non-volatile if none of its UCs has been changed with respect to version *i*. The extent of volatility of the subsequent version is then the percentage of changed UCs to the total UCs.

The impact of requirements volatility varies depending on the type of change [18][19]. Sherif reported that the modification type of change is the least risky with the least effort especially with mission critical requirements [16]. Stark et al. also reported the modification type is the least common form of changes in requirements compared with additions and deletions [17]. Accordingly, our study focused on the most relevant information, i.e. the deletion and addition types, besides the original ones. A UC is *original* if it exists in version *i*.

4.2 Calculation of Requirements Volatility

Retrospective analysis is primarily concerned with the study of successive system versions itself [19][28] It is a quantitative measure that can be used to study the requirements evolution and, in turn, the software evolution. Consider two software versions v_1 and v_2 with two sets of UCs: UC_1 and UC_2, Let N be the number new UCs in v_2, R be the number of removed UCs from v_1 and O be the number of initial UCs that exist in v_1. Therefore, $N + R$ represents the number of changed requirements (i.e. UCs) while $O + N + R$ represents the total number of requirements (UCs). As R is included in O, then $O + N + R$ could be simplified as $O + N$. We define requirements volatility between $v1$ and $v2$, $RV(v_i, v_{i+1})$, as the amount of changes to UCs with respect to vi and v_{i+1}. Operationally, $RV(v_i, v_{i+1})$ is then calculated as in Eq. (1):

$$VR(v_i, v_{i+1}) = \frac{N + R}{N + O} \qquad (1)$$

Where O > 0, R ≥ 0, N ≥ 0, R ≤ O.

The *RV* value varies from 0 to 1 where *RV* value of 0 indicates the lowest possible amount of change between v_i and v_{i+1} (i.e. volatile requirements) and *RV* value of 1 indicates the highest possible amount of change between vi and $vi+1$ (i.e. non-volatile requirements). As an example, consider two software versions v_1 and v_2 with three UCs in v_1 (UC_1, UC_2 and UC_3) and four UCs in v_2 (UC_1, UC_2, UC_4 and UC_5). In this case, O = 3 (i.e., UC_1, UC_2, and UC_3), R = 1 (i.e., UC_3), and N = 2 (i.e., UC_4 and UC_5). The requirements volatility between v_1 and v_2, $RV(v_1, v_2)$, is (2+1)/(3+2) = 0.6.

5.0 THEORETICAL VALIDATION

To date, no research looks at the mathematical properties required to theoretically validate the requirements volatility metrics. Therefore, we validate our metric using four cohesion properties proposed by Briand et al. [28][29]. Because these properties were general, they are applied to validate several types of software metrics such as software packaging and architectural stability so that any well-defined metric should satisfy these properties [28][29][30][31]. In his theoretical validation for stability metric at architecture level, Hassan [32] added three more properties: transitivity, package cohesion impact and change impact. While our proposed metric (Eq. 1) is not meant to be solely architecture stability metric nor cohesion, the rationale behind Briand and Hassan's properties is still applicable to our metric. The theoretical validation of RV based on these properties (except the unrelated property, package cohesion impact) is as follows.

1. Non-Negativity

This property holds that the metric value is greater than or equal to zero. It is worth noting here that the non-negativity attribute of software metrics (among other properties) is proposed in the literature and has been widely adopted as a formal property to evaluate software metrics [33]. The RV metric possesses this property because the variables used in it are the number of corresponding UCs, and such numbers cannot be negative.

2. Normalization

This property holds that the metric value belongs to a bounded interval. Normalization can generally provide support for meaningful comparisons between the requirements volatility of different software versions as they all belong to the same interval [28]. We opted to bind the measurements of RV to the interval [0, 1]. A "0" means there has not been any requirements change from the UCs' perspective across two versions. A "1" means all UCs have changed across two versions. The RV value is bounded between 0 and 1 as the denominator in it is always greater than or equal to the numerator; i.e. RV cannot be greater than 1 in value.

3. Null Value

This property holds that metric value is equal to zero if there is no change. RV metric has the null value when there is no requirement removed from version *i* or added to version *i+1*. This is done formally when R + N = 0.

4. Maximum Value

This property holds that the metric value is Max if there is no a common requirement between version *i* and version *i+1*. This property is satisfied with RV metric when both R and O are equal, i.e. all UCs in version *i* are removed from version *i+1*.

5. Transitivity

Consider three metric measurements such that the first measurement is less than the second and the second is less than the third, and then the first measure should be less than the third one. Version *j* is more volatile than *i* if there is a greater number of removed or new UCs in *j* than in *i*. RV will always show that a software version with a greater number of removed or new UCs has higher requirements volatility than a version with fewer removed or new UCs. Let *i*, *j*, *k* and *m* represent four versions; RV_j is the RV of *j* relative to its previous release *i*, RV_k the RV of *k* relative to its previous release *j*, and RV_m is the RV of *m* relative to its previous release *k*. If $RV_j < RV_k$ and $RV_k < RV_m$, it implies that $RV_j < RV_m$, which means the summation of both R and N in *k* and *m,* respectively, is more than those in *i* and *j*. Therefore, RV satisfies the transitivity property.

6. Change Impact

This property holds that if the number of changed UCs in release *j* relative to release *i* is less than that in release *k* relative to release *j,* then the volatility of *j* relative to *i* will be less than the volatility of *k* relative to *j* provided that the total UCs in *j* relative to *i* is not less than that in *k* relative to *j*. With RV metric, the number of changed UCs is formally represented by the numerator value, i.e. R + N. Thus, if the numerator value of *j* is less than that of *k*, then RV of *j* is higher than RV of *k*.

The above properties are theoretical concepts, so they don't necessarily apply in requirements engineering. For example, when RV actually reaches "maximum value", then it is no longer a version; it is new software entirely.

6.0 EMPIRICAL VALIDATION

Because of the common problem in software engineering of the lack of empirical data early in the software life cycle [34], we were not able to find relevant requirements artifacts where the RV metric could be applied. For this reason, we have reverse-engineered source code of JHotDraw open source project as explained in the next sections.

6.1 UC Extraction

According to Jacobson [20], the services of the system that user requires are documented in UCs. These UCs describe the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used. Each UC is realized by one or more sequence diagrams (SDs) that depict how the objects

interact and work together to provide services [35]. Each individual object provides only a small element of the functionality – its particular responsibilities – but when they work together, objects can produce services that people can use. Even though there are some tools such as Enterprise Architect[1], Together[2] and AltovaUModel[3] that can generate SDs from source code, the generated SDs are based on runtime behavior. In other words, when one specifies a particular method, the runtime system can generate the behavior of this method across the entire system. Source code of real software projects usually contains many statements representing the runtime environment such as "for, while, if and, switch" statements. A transient task should, therefore, be taking place after generating the SDs; it is a filtering process. Filtering is a means to eliminate all runtime variables and messages shown in UML-SDs. It then makes the generated "runtime" SDs mimic the "functional" SDs that describe the functional behavior of UCs through objects and messages starting from the pre-condition to the post-condition. Data have been calculated using computerized tools, including AltovaUModel, XMI2UC and Microsoft Excel, and are therefore reliable, except the outcomes of the filtering process (filtered SDs in Fig. 1) because manual judgment was involved. Moreover, we have done filtering carefully to keep the filtering results as valid and accurate as possible. The filtering step could then cause additional validity threats.

As we discussed in Section 3, the goal for RV is to be used in retrospective analysis of requirements. Requirements are commonly documented using CASE tools because XMI (XML Metadata Interchange) is the de-facto standard for storing artifacts by prominent CASE tools. As a result of data scarcity issues, we were not able to find requirements artifacts available in the literature as we pointed out in the paper. Accordingly, we had to reverse engineer the source code to XMI using AltovaUModel[4] tool (Enterprise Edition Version 2012 sp1) in generating the SDs from the system code. Because the SDs generated by AltovalUModel are runtime SDs, they are filtered to reflect the functional behavior of UCs, as we have explained. We then use an XMI to UCs transformation tool (XMI2UC) to extract UCs from OO source code via XMI documents generated by AltovaUModel. While filtering is performed, the whole system is exported in XMI document to be used as input to the XMI2UC tool. XMI2UC produces three outputs: (1) Package Information Document, (2) Class Information Document and (3) UC Information Document. The first two documents list details of all OO packages and classes, respectively. The UC in the third output is a sequence of related messages passing, i.e. method callings. A detailed discussion on the XMI2UC tool and its use to generate UCs can be found in a previous work [[36]]. Using this third document of any two different versions, we find all aspects related to the volatility, i.e. original UCs of the first version, and new and removed UCs of the second version, and finally calculate the RV value. Because the third document was in the text format, each UC sequence is put in one text line, and we used Microsoft Word to compare among the considered documents to find the original, new and removed UCs. The RV metric calculation of the obtained values related to RV metric was made using Microsoft Excel. The above reverse engineering process could be modeled through the DFD (Data Flow Diagram) shown in Fig. 1.

[1] http://www.sparxsystems.com/products/ea/

[2] http://www.borland.com/us/products/together/

[3] http://www.altova.com/umodel.html/

[4] http://www.altova.com/umodel.html/

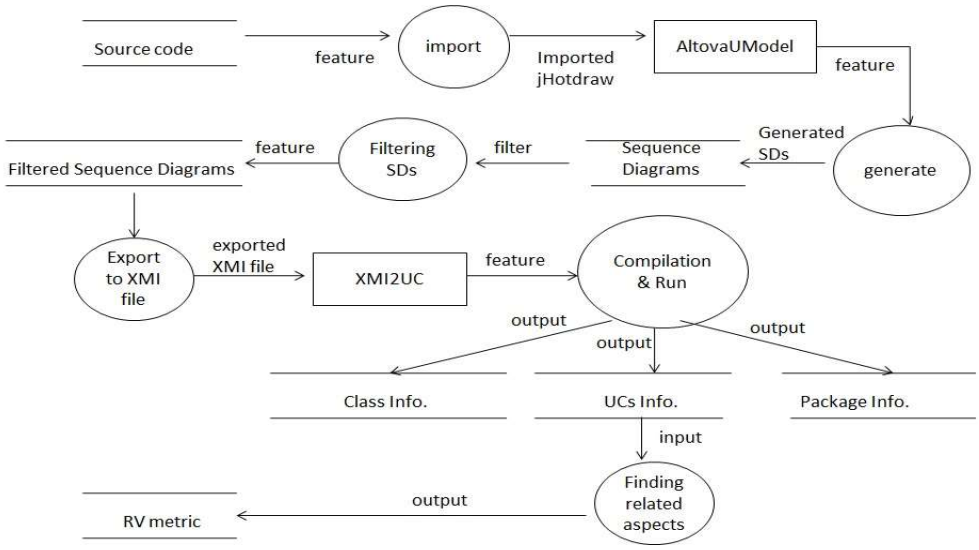


Fig. 1: Modeling the reverse engineering process through DFD

6.2 JHotDraw Case Study

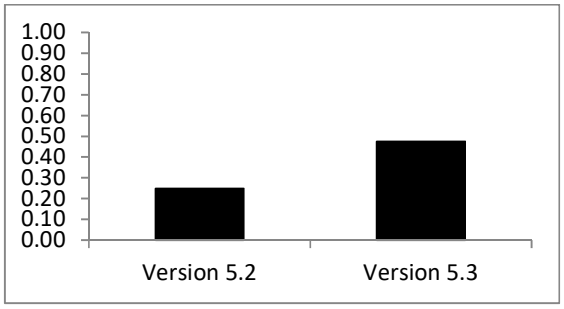
Using the above mentioned procedure of reverse engineering, we applied our proposed metric on a case study, JHotDraw. This section presents the obtained results.

JHotDraw[5] is an open source, two-dimensional graphics framework for structured drawing editors that are written in Java. Because a wide amount of historical data of JHotDraw is available in its CVS repository, several researchers used this system as a case study in their research [30][31][36][37][38][39]]. As a result of space limitations, we list the required aspects of the first two versions of JHotDraw: 5.1 and 5.2 obtained by the UC Information Document. In particular, UCs removed from version 5.1 and 5.2 (i.e. R values) and the new UCs added to version 5.2 and 5.3 (i.e. N values) are available in Table A.1 through A.4 in Appendix. Fig. 2 shows the statistics described in the previous section for the three versions of the JHotDraw project 5.1, 5.2 and 5.3. In terms of RV, it is clear that version 5.3 is more volatile than version 5.2 with a difference of 22%.

Intuitively, the result above is expected because of adding new features in version 5.3, which in turn leads directly to an increase in the value of RV metric. The undo functionality, for instance, is added to JHotDraw system starting from version 5.3. According to Canfora and Cerulo [37], JHotDraw has more than 20 commands that can be undone, causing the undo feature to be spread over many classes [37]. However, we did not find a previous work to compare against our result except the experiment conducted by [31]. They proposed a metric to measure the architecture stability of JHotDraw versions, in which their metric works on inter-package connections (IPCs) level. They also found the amount of changes at the source code level using the BeyondCompare tool[6], i.e. in terms of SLOC. Both of the results, at architecture level, and at source code level, were consistent with each other, so that stability value for version 5.2 is higher than that of version 5.3. Consistently, our results, which are at requirements level, confirm the main finding of their experiments, which indicates that the architecture of version 5.3 is less stable than that of version 5.2; RV metric can be viewed as an indicator of how stable requirements are in the system.

[5] www.JHotDraw.org

[6] http://www.scootersoftware.com/



Versions	O	R	N	RV Value
5.1	130	18	N/A	N/A
5.2	131	29	19	(18+19)/(130+19) = 0.25
5.3	165	N/A	63	(29+63)/(131+63) = 0.47

Fig. 2: Requirements volatility of JHotDraw versions in terms of UCs

7.0 THREATS TO VALIDITY

As with any experimental study, there are some threats to the validity of the study's results. We conclude this section by considering internal, construct and external threats [40]:

1. Threats to internal validity are influences that can affect the independent variable with respect to causality without the researcher's knowledge. Accordingly, the most important concern is the use of different tools for data collection. This threat is not expected to be a serious concern because we used existing commercial tools such as Altova. Although the other tool, XMI2UC, is developed for experimentation purposes only, several previous studies relied on it, and we reported sufficient information about it. To remedy the problem of data scarcity, we used both tools: Altova and XMI2UC to reverse engineer JHotDraw to identify UCs. Another factor that may lead to internal threat is the selection of types of requirements change (new and removed UCs), e.g. ignoring other change types such as UC modification. Although we ignore this type because of its low risk, this type of change across different software versions remains to be investigated.
2. Threats to construct validity concerns generalizing the result of the experiment to the concept or theory behind the experiment. In our study, part of the change we observed in the JHotDraw system could be caused by bug fixes rather than requirement changes.
3. Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. Accordingly, the JHotDraw case study does not show a size limitation for the study because of its large size. However, we validate RV empirically by looking at a single system only, JHotDraw, which is a threat to the external validity of our findings.

8.0 CONCLUSION AND FUTURE WORK

Volatile requirements tend to increase maintenance cost. Use cases (UCs) in OO systems could be used to capture requirements. Therefore, it is important to measure volatile UCs to obtain stable requirements. In this paper, we propose a requirements volatility (RV) metric to measure the volatility of the UCs. We validated RV against theoretical properties and also experimentally using an open source project, three versions of JHotDraw. Our results are a foundation for further empirical retrospective studies of the requirements properties. Retrospective measurements can be helpful in improving the predictive measures. The results obtained from these retrospective measurements can be studied against the predictive measurement, and the results can be used to further optimize and fine-tune the predictive measurement techniques. Any of the predictive approaches can be used for the purpose of relating a predictive approach with the retrospective approach. The goal would be to

use such predictors early during the requirements management to predict the volatility of the requirements. More work is required on empirical validation of RV metric using data from additional industrial projects. Correlating requirements volatility with external quality attributes such as defect detection or maintainability would also be investigated for more confidence on the metric validation.

REFERENCES

[1] M. Ali, "Metrics for Requirements Engineering", *MS Thesis*, Department of Computing Science, Umeå University, Umeå, Sweden, 2006.

[2] I. Sommerville, *Software Engineering*, 10th ed., Pearson, 2015. Chapter 4.

[3] A. Loconsole, "Definition and Validation of Requirements Management Measures", *PhD Thesis*, Umeå University, Umeå, Sweden, 2007.

[4] W. Lewis, *Software Testing and Continuous Quality Improvement*, 3rd ed., CRC Press, 2009.

[5] M. Christel, and K. Kang, "Issues in Requirements Elicitation", Technical Report, Software Engineering Institute, Sep 1992, No. CMU/SEI-92-TR-012, ESCTR-92-012.

[6] B. Nuseibeh, and S. Easterbrook, "Requirements Engineering: A Roadmap", in *Proceedings of The Conference on The Future of Software Engineering*, ACM Press, 2000, pp. 35-46.

[7] YK. Malaiya, and J. Denton, "Requirements Volatility and Defect Density", in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, Boca Raton, FL., 1-4 Nov 1998, pp. 285–294.

[8] L. Li, He. Shuguang, Q. Er-shi, "On Software Requirement Metrics Based on Six-Sigma", in *Symposium on Advanced Management of Information for Globalized Enterprises*, Tianjin, 28-29 Sep 2008, pp. 1-3.

[9] N. Nurmuliani, D. Zowghi, and S. Fowell, "Analysis of Requirements Volatility during Software Development Life Cycle", in *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC '04)*, Melbourne, Australia, 13-16 Apr 2004, pp. 28-37.

[10] Q. Wang, and X. Lai, "Requirements Management for The Incremental Development Model", in *Proceedings of the 2nd Asia-Pacific Conference on Quality Software*, 10-11 Dec 2001, Hong Kong, pp. 295-301.

[11] K. Hou, "Requirements Engineering and Management: A Development of A Requirements Management Metrics Portal", *MS Thesis*, Chalmers University of Technology, Göteborg, Sweden, 2009.

[12] D. Kavitha, and A. Sheshasaayee, "Requirements Volatility in Software Maintenance", In *Proceedings of the 2nd International Conference on Computer Science and Information Technology (CCSIT)*, Bangalore, India. 2-4 Jan 2012. In book: Advances in Computer Science and Information Technology. Computer Science and Information Technology, pp.142-150.

[13] A. Loconsole, and J. Börstler, "An Industrial Case Study on Requirements Volatility Measures", in *Proceedings of The 12th Asia-Pacific Software Engineering Conference, APSEC'05*, IEEE CS Press (2005), Taiwan, 15-17 Dec 2005, pp. 249–256.

[14] A. Qazi, K. B. S. Syed, R. G. Raj, E. Cambria, M. Tahir, D. Alghazzawi, "A concept-level approach to the analysis of online review helpfulness", *Computers in Human Behavior*, Vol. 58, May 2016, PP. 75-81, ISSN 0747-5632, http://dx.doi.org/10.1016/j.chb.2015.12.028.

[15] A. Qazi, R. G. Raj, M. Tahir, M. Waheed, S. U. R. Khan, and A. Abraham, "A Preliminary Investigation of User Perception and Behavioral Intention for Different Review Types: Customers and Designers Perspective," *The Scientific World Journal*, vol. 2014, Article ID 872929, 8 pages, 2014. doi:10.1155/2014/872929.

[16] J. Sherif, "Metrics for Software Risk Management", in *Proceedings of WESCON/96*, Anaheim, CA, USA, 22-24 Oct 1996, pp. 507-513.

[17] G. Stark, P. Oman, A. Skillicorn, R. Ameele, "An Examination of The Effects of Requirements Changes on Software Maintenance Releases", *Journal of Software Maintenance Research and Practice*, Vol. 11, No. 5, 1999, pp. 293–309.

[18] K. Abd Elwahab, M. Abd Elatif, S. Kholeif, "Identify and Manage The Software Requirements Volatility", *International Journal of Advanced Computer Science and Applications (IJACSA)*, Vol. 7, No. 5, 2016, pp. 64-71.

[19] M. Peña, and R. Valerdi, "Characterizing The Impact of Requirements Volatility on Systems Engineering Effort", *Systems Engineering*, Vol. 18, No. 1, 2015, pp. 59-70.

[20] N. Fenton, and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. 3rd edition, CRC Press, 2014. Chapters: 7 and 8.

[21] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. 1st ed., Addison-Wesley, 1992. Chapter 6 and 7.

[22] E. Weller, "Practical Applications of Statistical Process Control", *IEEE Software*, Vol. 17, No. 3, 2000, pp. 48-55.

[23] R. Selby, "Measurement-Driven Dashboards Enable Leading Indicators for Requirements and Design of Large-Scale Systems", in *Proceedings of The 11th International Software Metrics Symposium*, Como, Italy, 19-22 Sep 2005.

[24] G. Kulk, and C. Verhoef, "Quantifying Requirements Volatility Effects", *Science of Computer Programming*, 72(3) : 136-175, 2008.

[25] E. Markopoulos, G. Alexopoulos, and N. Bouzoukou, J. Bilbao, "Software Project Tracking Metrics Analysis Model Based on Project Requirements", In *Proceedings of The 11th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering (MMACTEE '09*), Athens, 2009, pp. 627-632.

[26] M. Obana, and N. Hanakawa, "Process Evaluation Based on Meeting Quality of Requirement Analysis Phase in Software Development Projects", *Journal of Software Engineering and Applications*, No. 7, No. 10, 2014, pp. 828-843.

[27] T. Mens, S. Demeyer, "Future Trends in Software Evolution Metrics", in *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*, Vienna, Austria, 10-14 Sep 2001, pp. 83-86.

[28] L. Briand, S. Morasca, and V. Basili, "Defining and Validating Measures for Object-Based High-Level Design", *IEEE Transactions Software Engineering*, Vol. 25, No. 5, 1999, pp. 722 – 743.

[29] L. Briand, J. Daly, J. Wuest, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", *Empirical Software Engineering*, Vol. 3, No. 1, 1998, pp. 65–117.

[30] S. Ebad, and M. Ahmed, "Functionality-Based Software Packaging Using Sequence Diagrams", *Software Quality Journal*, Vol. 23, No. 3, 2015, pp. 453-481.

[31] S. Ebad, and M. Ahmed, "Measuring Stability of Object Oriented Software Architectures", *IET Software*, Vol. 9, No. 3, 2015, pp. 76-82.

[32] Y. Hassan, "Measuring Software Architectural Stability Using Retrospective Analysis", *MS Thesis*, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, 2007.

[33] S. Ebad, and M. Ahmed, "Review and Evaluation of Cohesion and Coupling Metrics at Package and Subsystem Level", *Journal of Software (JSW)*, Vol. 11, No. 6, 2016, pp. 598-605.

[34] G. Boetticher, "Using Machine Learning to Predict Project Effort: Empirical Case Studies in Data-Starved Domains", in *Proceedings of the First International Workshop on Model-Based Requirements Engineering*, San Diego, 26-29 Nov 2001, pp. 17-27.

[35] D. Bell, "UML basics: the sequence diagram". Available at http://www.ibm.com/developerworks/rational/library/3101.html, 16 Feb, 2004. [accessed at 1 Oct 2016]

[36] S. Ebad and M. Ahmed, "XMI2UC: An automatic tool to extract use cases from object-oriented source code", *International Journal of Future Computer and Communication (IJFCC)*, Vol. 1, No. 2, 2012, pp. 193-195.

[37] G. Canfora, and L. Cerulo, "How Crosscutting Concerns Evolve in JHotDraw", in *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, Budapest, Hungary, 24-25 Sep 2005, pp.65-73.

[38] O. Seng, M. Bauer, M. Biehl, G. Pache, "Search-Based Improvement of Subsystem Decompositions", in *Proceedings Conference on Genetic and Evolutionary Computation (GECCO '05)*, Washington, DC, USA, June 2005, pp. 1045-1051.

[39] O. Seng, J. Stammel, D. Burkhart, "Search Search-Based Determination of Refactorings for Improving The Class Structure of Object Oriented Systems", in *Proceedings Conference on Genetic and Evolutionary Computation (GECCO '06)*, Seattle, WA, USA, July 2006, pp. 1909-1916.

[40] C. Wohlin, P. Runeson, M. H¨ost, M. C. Ohlsson, B. Regnell, A. Wessl´en, *Experimentation in Software Engineering*, Springer, 2012.

APPENDIX

Below are four tables A.1 up to A.4 to list the removed UCs and new UCs for the considered JHotDraw version. This information is obtained by the UC Information Document produced by the XMI2UC tool. The notation x$y means Method x in Class y. This notation is used by XMI2UC to avoid ambiguity in naming, i.e. methods having the same name located at different classes.

Table A.1: List of the UCs removed from version 5.1 (i.e. R values)

UC#	UC Path
UC_{11}	mouseMove$PolygonTool → pointCount$PolygonFigure → mouseMove$PolygonTool → setPointAt$PolygonFigure
UC_{18}	draw$AttributeFigure → hasDefined$FigureAttributes → draw$AttributeFigure → get$FigureAttributes → draw$AttributeFigure → set$FigureAttributes → draw$AttributeFigure → isTransparent$ColorMap
UC_{19}	getAttribute$AttributeFigure → hasDefined$FigureAttributes → getAttribute$AttributeFigure → get$FigureAttributes → getAttribute$AttributeFigure → set$FigureAttributes
UC_{20}	getFrameColor$AttributeFigure → hasDefined$FigureAttributes → getFrameColor$AttributeFigure → get$FigureAttributes → getFrameColor$AttributeFigure → set$FigureAttributes
UC_{21}	initializeAttributes$AttributeFigure → set$FigureAttributes
UC_{22}	read$AttributeFigure→ readString$StorableInput → read$AttributeFigure → read$FigureAttributes → readString$StorableInput → read$FigureAttributes→ readInt$StorableInput → read$FigureAttributes → readStorable$StorableInput → read$Storable
UC_{23}	setAttribute$AttributeFigure → set$FigureAttributes
UC_{24}	write$AttributeFigure→ writeString$StorableOutput → write$AttributeFigure → write$FigureAttributes → writeString$StorableOutput → write$FigureAttributes → writeInt$StorableOutput → write$FigureAttributes → writeStorable$StorableOutput → write$Storable
UC_{34}	handles$ImageFigure→ addHandles$BoxHandleKit
UC_{33}	execute$GroupCommand → addAll$CompositeFigure
UC_{79}	beginEdit$TextTool → getText$FloatingTextField → beginEdit$TextTool → setText$TextHolder → beginEdit$TextTool → endOverlay$FloatingTextField → beginEdit$TextTool →getFont$TextHolder → beginEdit$TextTool → createOverlay$FloatingTextField → beginEdit$TextTool → textDisplayBox$TextHolder → beginEdit$TextTool → overlayColumns$TextHolder → beginEdit$TextTool → getPreferredSize$FloatingTextField → beginEdit$TextTool → getText$TextHolder → beginEdit$TextTool → setBounds$FloatingTextField
UC_{80}	deactivate$TextTool → getText$FloatingTextField → deactivate$TextTool → setText$TextHolder → deactivate$TextTool→ endOverlay$FloatingTextField
UC_{81}	endEdit$TextTool → getText$FloatingTextField → endEdit$TextTool → setText$TextHolder endEdit$TextTool → endOverlay$FloatingTextField
UC_{82}	fieldBounds$TextTool → textDisplayBox$TextHolder → fieldBounds$TextTool → overlayColumns$TextHolder → fieldBounds$TextTool →getPreferredSize$FloatingTextField
UC_{83}	mouseDown$TextTool → acceptsTyping$TextHolder → mouseDown$TextTool → getText$FloatingTextField → mouseDown$TextTool → setText$TextHolder → mouseDown$TextTool → endOverlay$FloatingTextField → mouseDown$TextTool → getFont$TextHolder → mouseDown$TextTool → createOverlay$FloatingTextField → mouseDown$TextTool → textDisplayBox$TextHolder→ mouseDown$TextTool → overlayColumns$TextHolder → mouseDown$TextTool → getPreferredSize$FloatingTextField → mouseDown$TextTool → getText$TextHolder → mouseDown$TextTool → setBounds$FloatingTextField
UC_{85}	findPoint$AbstractConnector → center$Geom
UC_{119}	paintNormal$ToolButton → normal$PaletteIcon
UC_{120}	paintPressed$ToolButton → pressed$PaletteIcon

Table A.2: List of the new UCs added to version 5.2 (i.e. N values)

UC#	UC Path
UC_1	mouseDown$CustomSelectionTool → setSelectedFigure$PopupMenuFigureSelection
UC_2	mouseUp$CustomSelectionTool → setSelectedFigure$PopupMenuFigureSelection
UC_3	showPopupMenu$CustomSelectionTool → setSelectedFigure$PopupMenuFigureSelection
UC_5	basicDisplayBox$GraphicalCompositeFigure → layout$Layouter
UC_6	layout$GraphicalCompositeFigure → calculateLayout$Layouter → layout$GraphicalCompositeFigure → layout$Layouter
UC_7	read$GraphicalCompositeFigure → readInt$StorableInput → read$GraphicalCompositeFigure→ readStorable$StorableInput → read$Storable
UC_8	update$GraphicalCompositeFigure → calculateLayout$Layouter → update$GraphicalCompositeFigure → layout$Layouter
UC_9	write$GraphicalCompositeFigure → writeInt$StorableOutput → write$GraphicalCompositeFigure → writeStorable$StorableOutput → write$Storable
UC_{25}	handlePopupMenu$CustomSelectionTool → setSelectedFigure$PopupMenuFigureSelection
UC_{26}	read$ArrowTip → readDouble$StorableInput → read$ArrowTip → readString$StorableInput → read$ArrowTip → readColor$FigureAttributes → readInt$StorableInput
UC_{27}	write$ArrowTip → writeDouble$StorableOutput → write$ArrowTip → writeColor$FigureAttributes → writeString$StorableOutput → writeColor$FigureAttributes → writeInt$StorableOutput → writeColor$FigureAttributes → write$ArrowTip → writeString$StorableOutput
UC_{36}	handles$GroupFigure → northWest$RelativeLocator → handles$GroupFigure → northEast$RelativeLocator → handles$GroupFigure → southWest$RelativeLocator → handles$GroupFigure → southEast$RelativeLocator
UC_{43}	write$LineConnection → writeInt$StorableOutput → write$LineConnection → writeStorable$StorableOutput → write$Storable → writeStorable$StorableOutput → write$LineConnection → writeColor$StorableOutput → write$LineConnection → writeStorable$StorableOutput → write$Storable
UC_{70}	changed$TextFigure → locate$OffsetLocator
UC_{115}	getMaximumSize$ToolButton → getWidth$PaletteIcon → getMaximumSize$ToolButton → getHeight$PaletteIcon
UC_{118}	paint$ToolButton → selected$PaletteIcon → paint$ToolButton → selected$PaletteIcon
UC_{129}	restore$StandardStorageFormat → readStorable$StorableInput → read$Storable
UC_{130}	findStorageFormat$StorageFormatManager → getFileFilter$StorageFormat
UC_{131}	registerFileFilters$StorageFormatManager → getFileFilter$StorageFormat

Table A.3: List of the UCs removed from version 5.2 (i.e. R values)

UC#	UC Path
UC_9	write$GraphicalCompositeFigure → writeInt$StorableOutput → write$GraphicalCompositeFigure → writeStorable$StorableOutput → write$Storable
UC_{11}	chop$PolygonFigure → length2$Geom
UC_{12}	distanceFromLine$PolygonFigure → length$Geom
UC_{28}	chop$ChopEllipseConnector → pointToAngle$Geom → chop$ChopEllipseConnector → ovalAngleToPoint$Geom
UC_{29}	mouseDown$ConnectedTextTool → acceptsTyping$TextHolder → mouseDown$ConnectedTextTool → getText$FloatingTextField → mouseDown$ConnectedTextTool → setText$TextHolder → mouseDown$ConnectedTextTool → endOverlay$FloatingTextField → mouseDown$ConnectedTextTool → getFont$TextHolder → mouseDown$ConnectedTextTool → createOverlay$FloatingTextField → mouseDown$ConnectedTextTool → textDisplayBox$TextHolder → mouseDown$ConnectedTextTool → overlayColumns$TextHolder → mouseDown$ConnectedTextTool → getPreferredSize$FloatingTextField → mouseDown$ConnectedTextTool → getText$TextHolder → mouseDown$ConnectedTextTool → setBounds$FloatingTextField → mouseDown$ConnectedTextTool → connect$TextHolder
UC_{36}	handles$GroupFigure → northWest$RelativeLocator → handles$GroupFigure → northEast$RelativeLocator → handles$GroupFigure → southWest$RelativeLocator → handles$GroupFigure → southEast$RelativeLocator
UC_{40}	execute$InsertImageCommand → instance$Iconkit → execute$InsertImageCommand → registerAndLoadImage$Iconkit
UC_{45}	containsPoint$PolyLineFigure→ lineContainsPoint$Geom
UC_{53}	draw$RadiusHandle→ getArc$RoundRectangleFigure→ draw$RadiusHandle → displayBox$RoundRectangleFigure
UC_{54}	invokeStart$RadiusHandle→ getArc$RoundRectangleFigure
UC_{55}	invokeStep$RadiusHandle → displayBox$RoundRectangleFigure→ invokeStep$RadiusHandle → range$Geom → invokeStep$RadiusHandle → setArc$RoundRectangleFigure
UC_{56}	locate$RadiusHandle→ getArc$RoundRectangleFigure→ locate$RadiusHandle → displayBox$RoundRectangleFigure
UC_{64}	mouseDown$ScribbleTool → addPoint$PolyLineFigure
UC_{77}	read$TextFigure→ readString$StorableInput → read$TextFigure → read$FigureAttributes → readString$StorableInput → read$FigureAttributes → readInt$StorableInput → read$FigureAttributes → readStorable$StorableInput → read$Storable → readStorable$StorableInput → read$FigureAttributes → read$TextFigure → readInt$StorableInput → read$TextFigure → readBoolean$StorableInput → read$TextFigure → readStorable$StorableInput → read$Storable
UC_{78}	setAttribute$TextFigure→ locate$OffsetLocator → setAttribute$TextFigure → locate$OffsetLocator → setAttribute$TextFigure → locate$OffsetLocator → setAttribute$TextFigure → set$FigureAttributes → setAttribute$TextFigure → locate$OffsetLocator
UC_{82}	write$TextFigure → writeString$StorableOutput → write$TextFigure → write$FigureAttributes → writeString$StorableOutput →write$FigureAttributes → writeInt$StorableOutput → write$FigureAttributes → writeStorable$StorableOutput → write$Storable →writeStorable$StorableOutput → write$FigureAttributes → write$TextFigure → writeInt$StorableOutput → write$TextFigure → writeBoolean$StorableOutput → write$TextFigure→ writeStorable$StorableOutput → write$OffsetLocator → writeInt$StorableOutput → write$OffsetLocator → writeInt$StorableOutput
UC_{89}	connectedTextLocator$AbstractFigure → center$RelativeLocator
UC_{90}	invokeStep$ChangeConnectionHandle → center$Geom
UC_{94}	read$CompositeFigure→ readInt$StorableInput → read$CompositeFigure → readStorable$StorableInput → read$Storable
UC_{97}	mouseDrag$ConnectionTool → center$Geom
UC_{99}	execute$CutCommand → getClipboard$Clipboard → execute$CutCommand → setContents$Clipboard
UC_{101}	copySelection$FigureTransferCommand → getClipboard$Clipboard → copySelection$FigureTransferCommand → setContents$Clipboard
UC_{104}	execute$PasteCommand→ getClipboard$Clipboard → execute$PasteCommand → getContents$Clipboard → →
UC_{111}	constrainPoint$StandardDrawingView → range$Geom → constrainPoint$StandardDrawingView → range$Geom
UC_{112}	mouseDragged$StandardDrawingView → range$Geom → mouseDragged$StandardDrawingView → range$Geom

UC_{113}	mousePressed$StandardDrawingView → range$Geom→ mousePressed$StandardDrawingView → range$Geom
UC_{114}	mouseReleased$StandardDrawingView → range$Geom → mouseReleased$StandardDrawingView → range$Geom
UC_{126}	mouseExited$PaletteButton → paletteUserOver$PaletteListener
UC_{128}	mouseReleased$PaletteButton→ paletteUserSelected$PaletteListener

Table A.4: List of the new UCs added to version 5.3 (i.e. N values)

UC#	UC Path
UC_1	chop$ChopDiamondConnector → pointToAngle$Geom → chop$ChopDiamondConnector angleToPoint$Geom
UC_6	chop$DiamondFigure → chop$PolygonFigure→ length2$Geom → → → →
UC_{12}	endDraggingFrame$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{13}	endResizingFrame$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{14}	add$MDIDesktopPane → resizeDesktop$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{15}	cascadeFrames$MDIDesktopPane → setNormalSize$MDIDesktopManager → setAllSize$MDIDesktopPane → setNormalSize$MDIDesktopManager → cascadeFrames$MDIDesktopPane → resizeDesktop$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{16}	checkDesktopSize$MDIDesktopPane → resizeDesktop$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{17}	remove$MDIDesktopPane → resizeDesktop$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{18}	tileFrames$MDIDesktopPane → setNormalSize$MDIDesktopManager → setAllSize$MDIDesktopPane → setNormalSize$MDIDesktopManager → tileFrames$MDIDesktopPane → resizeDesktop$MDIDesktopManager → setAllSize$MDIDesktopPane
UC_{23}	activate$PolygonTool → fireToolActivatedEvent$EventDispatcher
UC_{27}	mouseDrag$PolygonTool → addPoint$PolygonFigure
UC_{28}	mouseMove$PolygonTool →pointCount$PolygonFigure → mouseMove$PolygonTool → setPointAt$PolygonFigure
UC_{35}	read$AbstractLineDecoration → readString$StorableInput → read$AbstractLineDecoration → readColor$FigureAttributes →readInt$StorableInput
UC_{36}	write$AbstractLineDecoration→ writeColor$FigureAttributes → writeString$StorableOutput → writeColor$FigureAttributes → writeInt$StorableOutput → writeColor$FigureAttributes →write$AbstractLineDecoration → writeString$StorableOutput
UC_{39}	draw$AttributeFigure → hasDefined$FigureAttributes → draw$AttributeFigure → get$FigureAttributes→ draw$AttributeFigure→ set$FigureAttributes → draw$AttributeFigure → isTransparent$ColorMap
UC_{40}	getAttribute$AttributeFigure → hasDefined$FigureAttributes → getAttribute$AttributeFigure→ get$FigureAttributes → getAttribute$AttributeFigure → set$FigureAttributes
UC_{41}	getFillColor$AttributeFigure → hasDefined$FigureAttributes → getFillColor$AttributeFigure → get$FigureAttributes → getFillColor$AttributeFigure → set$FigureAttributes
UC_{42}	initializeAttributes$AttributeFigure→ set$FigureAttributes
UC_{43}	read$AttributeFigure→ readString$StorableInput → read$AttributeFigure → read$FigureAttributes → readString$StorableInput → read$FigureAttributes → readInt$StorableInput → read$FigureAttributes → readStorable$StorableInput → read$Storable
UC_{44}	setAttribute$AttributeFigure → set$FigureAttributes
UC_{45}	write$AttributeFigure → writeString$StorableOutput → write$AttributeFigure → write$FigureAttributes → writeString$StorableOutput→ write$FigureAttributes → writeInt$StorableOutput → write$FigureAttributes →writeStorable$StorableOutput → write$Storable

UC_{46}	writeObject$AttributeFigure→ hasDefined$FigureAttributes → writeObject$AttributeFigure → get$FigureAttributes → writeObject$AttributeFigure → set$FigureAttributes
UC_{53}	draw$ImageFigure→ instance$Iconkit → draw$ImageFigure → getImage$Iconkit
UC_{54}	handles$ImageFigure → addHandles$BoxHandleKit
UC_{75}	activate$ScribbleTool → fireToolActivatedEvent$EventDispatcher
UC_{93}	findPoint$AbstractConnector → center$Geom
UC_{102}	orphan$CompositeFigure→ add$QuadTree →getAbsoluteBoundingRectangle2D$QuadTree
UC_{103}	removeAll$CompositeFigure→ remove$QuadTree
UC_{107}	activate$CreationTool → fireToolUsableEvent$EventDispatcher
UC_{110}	insertFigures$NullDrawingView→ pointToAngle$Geom → insertFigures$NullDrawingView angleToPoint$Geom
UC_{111}	selectionElements$NullDrawingView → fireToolActivatedEvent$EventDispatcher
UC_{118}	mouseMove$SelectionTool → writeDouble$StorableOutput
UC_{120}	getData$StandardFigureSelection→ nextElement$ReverseVectorEnumerator
UC_{131}	addCheckItem$CommandMenu → name$Command
UC_{133}	figureChanged$GraphLayout → execute$Command
UC_{135}	execute$RedoCommand → name$Command → execute$RedoCommand → addCommandListener$Command
UC_{136}	isExecutableWithView$RedoCommand → isExecutable$Command
UC_{138}	assertCompatibleVersion$StandardVersionControlStrategy → paletteUserOver$PaletteListener
UC_{139}	handleIncompatibleVersions$StandardVersionControlStrategy → isRedoable$UndoManager → isRedoable$Undoable → isRedoable$UndoManager → handleIncompatibleVersions$StandardVersionControlStrategy → popRedo$UndoManager → handleIncompatibleVersions$StandardVersionControlStrategy → redo$Undoable → handleIncompatibleVersions$StandardVersionControlStrategy → isUndoable$Undoable → handleIncompatibleVersions$StandardVersionControlStrategy → pushUndo$UndoManager → isUndoable$Undoable → pushUndo$UndoManager → handleIncompatibleVersions$StandardVersionControlStrategy → getDrawingView$Undoable
UC_{140}	isCompatibleVersion$StandardVersionControlStrategy → getRedoSize$UndoManager
UC_{143}	duplicateAffectedFigures$UndoableAdapter → read$Storable
UC_{144}	release$UndoableAdapter → write$Storable
UC_{145}	execute$UndoableCommand → getFileFilter$StorageFormat
UC_{146}	view$UndoableCommand → getFileFilter$StorageFormat
UC_{146}	invokeEnd$UndoableHandle → getFileFilter$StorageFormat
UC_{147}	invokeEnd$UndoableHandle → duplicateFigures$StandardFigureSelection → writeInt$StorableOutput → duplicateFigures$StandardFigureSelection→ writeStorable$StorableOutput → write$Storable→ writeStorable$StorableOutput → duplicateFigures$StandardFigureSelection → close$StorableOutput→ duplicateFigures$StandardFigureSelection → readInt$StorableInput → duplicateFigures$StandardFigureSelection → readStorable$StorableInput → read$Storable
UC_{148}	deactivate$UndoableTool → getEmptyEnumeration$FigureEnumerator
UC_{149}	execute$UndoCommand → getDrawingEditor$Command → execute$UndoCommand → execute$Command → execute$UndoCommand → getUndoActivity$Command→ execute$UndoCommand → isUndoable$Undoable
UC_{150}	isExecutableWithView$UndoCommand → getDrawingEditor$Command
UC_{151}	getAffectedFigures$UndoRedoActivity→ isUndoable$Undoable
UC_{152}	getAffectedFiguresCount$UndoRedoActivity → isUndoable$UndoManager → isUndoable$Undoable→ isUndoable$UndoManager → getAffectedFiguresCount$UndoRedoActivity → popUndo$UndoManager → getAffectedFiguresCount$UndoRedoActivity → undo$Undoable → getAffectedFiguresCount$UndoRedoActivity →isRedoable$Undoable → getAffectedFiguresCount$UndoRedoActivity → pushRedo$UndoManager → isRedoable$Undoable→ pushRedo$UndoManager → getAffectedFiguresCount$UndoRedoActivity → getDrawingView$Undoable
UC_{153}	getDrawingView$UndoRedoActivity → getUndoSize$UndoManager
UC_{154}	isRedoable$UndoRedoActivity→ isRedoable$Undoable

UC_{155}	isUndoable$UndoRedoActivity → getAffectedFigures$Undoable
UC_{156}	redo$UndoRedoActivity → getAffectedFiguresCount$Undoable
UC_{157}	release$UndoRedoActivity→ getDrawingView$Undoable
UC_{158}	setAffectedFigures$UndoRedoActivity → isUndoable$Undoable
UC_{159}	setRedoable$UndoRedoActivity → isRedoable$Undoable
UC_{160}	setUndoable$UndoRedoActivity → isRedoable$Undoable → setUndoable$UndoRedoActivity → undo$Undoable
UC_{161}	undo$UndoRedoActivity → release$Undoable
UC_{162}	undo$UndoRedoActivity → setAffectedFigures$Undoable
UC_{163}	undo$UndoRedoActivity → setUndoable$Undoable
UC_{164}	undo$UndoRedoActivity → setRedoable$Undoable
UC_{165}	undo$UndoRedoActivity → isUndoable$Undoable→ undo$UndoRedoActivity → redo$Undoable