An Evaluation of Design Rule Spaces as Risk Containers

Andrew Leigh, Michel Wermelinger, Andrea Zisman
Computing and Communications Department, The Open University, Milton Keynes, United Kingdom
andrew.leigh@open.ac.uk

***Abstract*—It is well understood that software development can be a risky enterprise and industrial projects often overrun budget and schedule. Effective risk management is, therefore, vital for a successful project outcome. Design Rule Spaces (DRSpaces) have been used by other researchers to understand why implemented software is error-prone [1, 2]. This industrial case study evaluates whether such spaces are durable, meaningful, and isolating risk containers. DRSpaces were created from UML class diagrams of architectural design artefacts. In our study, object orientated metrics were calculated from the UML diagrams, and compared to the error-proneness of the DRSpace implementation, to determine whether architectural coupling translated into implementation difficulties. A correlation between architectural coupling and error-proneness of DRSpaces was observed in the case study. Software developers were asked to identify DRSpaces they found difficult to implement, in order to understand which factors, other than architectural coupling, were also important. The qualitative results show agreement between the code areas developers found difficult to implement and the error-prone DRSpaces. However, the results also show that architectural coupling is just one risk factor of many. The case study suggests that architectural DRSpaces can be used to facilitate a targeted risk review prior to implementation and manage risk.**

Keywords- analysis; architecture; risk; software

I. INTRODUCTION

The motivation for our research is to help software projects to be more successful in terms of quality, cost, and schedule. According to Akingbehin [3], errors found during design stage of software development are only three to six times more expensive to correct when compared to errors found during the requirements stage. Moreover, errors found at the testing stages are 15-70 times more expensive to correct. In this work, we focus on using architecture analysis at the design stage to identify technical risks that may otherwise lead to errors later on in the software development life-cycle. The specific problem we consider in this paper is how can architectures be partitioned to aid effective risk assessment at the design stage of software development and manage risks during the subsequent implementation stage.

Several approaches have been proposed to support the application of multi-criteria decision analysis (MCDA) techniques in order to avoid making wrong architectural choices and lowering the risk of implementation. Research of MCDA techniques has been described in recent surveys by Falessi et al [4] and Rekha & Muccini [5]. A software architecture decision can be seen as a choice that trades off various quality attributes important to stakeholders. As stated by Bengtsson [6], "tradeoffs between qualities are inevitable and need to be made explicit during architectural design. This is especially important because architectural designs are generally very hard to change at a later stage".

However, tradeoffs need to be assessed for residual risks that may be carried forth into the software development process. Once risks have been captured, understood, and mitigated it is vital that they are not forgotten and are well managed. Therefore, it is necessary to create risk containers that are:

1. **Durable** – they must persist throughout the software development life-cycle so that attributed risks do not loose attention;
2. **Meaningful** – they must be understandable to both architects and developers so that architects can elicit risks and manage them during the implementation;
3. **Isolating** – they should separate areas of low and high risk within the overall architecture to focus on the areas of greatest risk.

Baldwin and Clark [7] conceived Design Rules (DRs) to represent the interfaces that split an architecture into independent modules. Their concept was used by Xiao et al [1] to propose DRSpaces, which are graphs that represent overlapping collections of programs within an architecture. Each space must have one or more leading classes that relate to DRs. In DRSpaces, the vertices represent related classes and the edges are relationships between those classes such as inheritance, composition, aggregation, realisation, design patterns, method parameters, and variable types. In this paper, we evaluate if architectural DRSpaces are durable, meaningful, and isolating risk containers.

If an architectural DRSpace has high coupling it is more likely to be difficult to implement and maintain when compared to other DRSpaces with less coupling. Irrespective of whether the coupling is to other classes in the same DRSpace or to classes in other DRSpaces, the resultant risk has been isolated within that DRSpace. This is because the source of the difficulty is coupling that stems from fulfilment of the DR. This hypothesis can be verified by determining whether there is a correlation between coupling of a DRSpace and the error-proneness of its implementation.

In the case study used in this paper, we verify the risk isolating characteristic of architectural DRSpaces. We do not verify meaningfulness and durability characteristics explicitly because the software was developed by closely

following the architecture. We therefore assume that the case study DRSpaces are meaningful and durable

The remainder of this paper is structured as follows. In Section 2 we present an account of related work. In Section 3 we explain how data was collected before presenting our analysis in Section 4. Finally, we summarise our conclusions in Section 5.

II. RELATED WORK

Xiao et al [1] recently proposed DRSpaces as a method to gain architectural insight. They used a tool called Titan to parse source code for DRs and attribute files to DRSpaces. In order to compare DRSpaces for error-proneness they used a bug space, which is a collection of programs that contains at least n bugs. Once the programs were attributed to DRSpaces, error-proneness of each DRSpace was determined by expressing the percentage of the bug space occupied by the DRSpace.

Xiao et al [1] observed that DRSpaces were stable over time and that buggy files within them are architecturally connected. They also observed lots of files changed together despite not being structurally related and suggested that these files may have "shared secrets" that cannot be determined from structure. They concluded that all error-prone DRSpaces exhibited multiple structural and evolutionary coupling issues. Their results support our view that if DRSpaces can be constructed from architectural plans they are likely to be risk isolating. Kazman et al [2] used DRSpaces as a basis for estimating a return on investment due to preventive maintenance.

Naedele et al [8] have a similar motivation to ours for their comparison of software development to manufacturing. They identified a number of gaps in software development processes including: "Lack of common conceptual frameworks driving improvement loops from development data". They proposed that DRSpaces could be used to fill this gap owing to their stability. This aspect supports our view that DRSpaces are durable risk containers.

Although many of the related DRSpace approaches [1, 2, 8] have similar motivation to our work, the majority of the approaches are based on code analysis to identify architectural problems. However, code analysis does not support the identification of architectural risks before implementation. The main difference between our work and existing works [1, 2, 8] is that we are investigating DRSpace analysis during the design stage, instead of during the implementation stage. This is because identifying and avoiding potential risks at the design stage will be cheaper.

III. DATA COLLECTION

The case study that we use in this paper is a deployed system from an organization, which wishes to remain anonymous. It includes an Application Programming Interface (API) that enables three clients to integrate with a database within an enterprise system. The API has 474 Java classes containing a total of 87.85 thousand lines of code (KLOC). The data collected covered a period of four years from the start of the project until the factory acceptance test (FAT) build. Three developers, each with more than ten years' experience, were involved in developing the API.

The case study uses UML class diagrams and key design decisions documented in the architectural design. The classes were manually allocated to the DRSpaces. Each key interface (DR) was used as the basis of a DRSpace.

All classes *subordinate* to a DR were added to the respective DRSpace. Subordination was determined by fulfilment of a design pattern, inheritance, realisation, composition or aggregation. For example, DRSpace A_1 represents an interface and all its realisations and DRSpace A_2 represents all classes that comprised a Service Facade. The *implementation* classes unreferenced by the UML diagrams were manually added to DRSpaces by means of package and class naming convention. For example, if a DRSpace contained an inheritance tree, any additional subclasses conceived during implementation would be added to that DRSpace. This was possible due to strict class naming conventions. As per the Xiao et al [1] method it was valid for a class to be a member of more than one DRSpace.

In order to test our hypothesis we used the Coupling Between Objects (CBO) metric proposed by Chidamber and Kemerer [9] to measure coupling between API classes. In addition, we also used Number of Children (NOC) and Depth in Inheritance Tree (DIT) to measure inheritance relationships between API classes and Weighted Methods per Class (WMC) to measure how many methods each API class has. The relationships and operations on the UML class diagrams and the DRSpace class allocations were recorded in files. The metrics were then automatically calculated for each DRSpace from the file contents, according to the following rules:

- *Coupling Between Objects (CBO):* the sum over all classes in the DRSpace of relationships of types like aggregation, composition, and dependencies (if the class is the relationship parent), and generalisation (if the class is the child).
- *Number of Children (NOC):* the sum over all classes in the DRSpace of all subclasses for each class.
- *Depth in Inheritance Tree (DIT):* the sum over all classes in the DRSpace of each class position in the inheritance tree.
- *Weighted Methods per Class (WMC):* the sum over all classes in the DRSpace of the explicit and inherited methods for each class.

A large DRSpace, with many classes, is more likely to have high values for the metrics. In the next section we report both the absolute and the normalized by size values of the metrics.

We automatically extracted data for each Java class from the Subversion code repository to determine error-proneness. The captured data are: class name, KLOC, and number of associated closed bug numbers. The bug space threshold was defined by the 75th percentile of bugs per KLOC, which was determined to be 55. Thus, the bug space contained all files with an error rate of 55 bugs per KLOC or more. Each DRSpace occupied a percentage of the bug space according to how many of bug space's files were attributed to the DRSpace.

Without being given prior knowledge of the DRSpaces, developers were asked to nominate and justify why they found particular groups of related classes difficult to implement and maintain. The nominated classes were translated to DRSpaces by identifying which spaces contained those classes. Two groups of the nominated classes fitted neatly into single DRSpaces. This suggests that DRSpaces are meaningful to developers as well as architects. The third group of classes could not be attributed to any of the DRSpaces because there was insufficient information in the design document to form the DRSpace that would have contained those classes.

There were three reasons why developers were asked to independently nominate areas of code. Firstly, to avoid bias because an author is the project architect. Secondly, because discussing the DRSpaces found with architects (or developers) might influence the interviewee. Thirdly, because the developer's perception of the difficulties is more relevant than the architect's perception due to the fact that the developers are the ones who implement and maintain the code. Table I shows the DRSpaces which relate to the groups of classes nominated by the developers.

TABLE I. DRSPACES NOMINATED BY DEVELOPERS AS DIFFICULT TO IMPLEMENT AND MAINTAIN

DRSpace	Justification
A_2	Currently any specific job type data has to be parsed from the message body.
A_4	There seem to be too many subclasses to maintain and they are very complicated.
None	Complex synchronization code in Package X is duplicated amongst concrete subclasses, it should be consolidated in the abstract class.

IV. ANALYSIS

The results of the DRSpace analysis are shown in Table II. The column entitled Bug Space % All Bugs indicates that the top five DRSpaces account for 66% of the bug space files, and the top ten DRSpaces account for 90% of the bug space files when considering all bugs. This is in agreement with Xiao et al [1] that "a few error-prone DRSpaces can capture a large portion of the project's error-prone files". The results show we were able to attribute 81% of all API classes to DRSpaces formed from the UML class diagrams and naming conventions. Table II also lists the values of Coupling Between Objects (CBO), Depth in Inheritance Tree (DIT), Number of Children (NOC) and Weighted Methods per Class (WMC) for each DRSpace.

Our hypothesis predicts agreement between rankings of DRSpaces by the proportion of the bug spaced occupied and the CBO metric derived from the architectural design. In order to test this hypothesis, Spearman's [10] rank correlation coefficient which is a statistical measurement of the association between two variables was used (see Table III). The method used to calculate metrics relied upon the UML class diagrams to determine the architected coupling. The ρ value suggests a strong positive correlation between CBO and error-proneness is present in the data when all bugs are considered. According to the Spearman's rank critical values, the α level indicates significance at the 0.001 level for absolute and normalised metrics and the null hypothesis can be rejected.

TABLE II. DRSPACE ERROR-PRONENESS AND OO METRICS

								Normalised By Size %			
DRSpace	Size %	Bug Space % All Bugs	Bug Space % Recent Bugs (Last 6 Months)	CBO	DIT	NOC	WMC	CBO	DIT	NOC	WMC
A_1	14.49	20.00	7.75	65	26	22	140	9.42	3.77	3.19	20.29
A_2	20.34	13.18	12.40	158	15	11	208	32.13	3.05	2.24	42.30
A_3	9.06	12.27	2.33	36	26	11	83	3.26	2.36	1.00	7.52
A_4	10.56	10.91	9.30	77	12	13	185	8.13	1.27	1.37	19.54
A_5	2.92	10.00	2.33	10	10	10	0	0.29	0.29	0.29	0.00
A_6	3.60	8.18	0.00	34	6	6	19	1.22	0.22	0.22	0.68
A_7	5.62	4.55	4.65	28	4	0	10	1.57	0.22	0.00	0.56
A_8	3.67	2.27	4.26	59	13	29	61	2.17	0.48	1.06	2.24
A_9	1.57	1.82	0.00	5	0	0	4	0.08	0.00	0.00	0.06
A_10	3.15	1.82	1.16	25	15	6	133	0.79	0.47	0.19	4.18
A_11	1.99	1.36	4.65	10	8	3	12	0.20	0.16	0.06	0.24
A_12	2.58	0.91	0.00	0	0	0	3	0.00	0.00	0.00	0.08
A_13	1.35	0.00	2.33	2	0	0	0	0.03	0.00	0.00	0.00

Comparison of Table I and Table II suggests disagreement between the most error-prone DRSpaces and those nominated when all bugs are considered. This is because the two nominations that could be resolved to DRSpaces are ranked 2nd (A_2) and 4th (A_4), respectively. The precision and recall of the developer nominations is 0.5, if only the two most error-prone DRSpaces are considered. However, A_2 and A_4 would be the top two if ranked by CBO.

The data collected included all fixed bugs including early developmental bugs. If the data is filtered to include only recent bugs, those created and fixed within the last six months, a different error-prone ranking emerges and the two most error-prone DRSpaces are in complete agreement (precision and recall of 1) with the groups of classes that could be resolved to DRSpaces. This seems to indicate that developers failed to remember older difficulties.

The correlation between the absolute values of CBO and error-proneness is weakened when only recent bugs are considered. However, it remains almost as strong when normalised by DRSpace size. In either case, it is more significant than the 0.05 level. The metric results suggest all metrics are useful predictors of error-proneness. This is because whether considering absolute or size normalized, all or recent bugs, all results show strong correlations that are significant to the 0.05 level or greater.

A developer nominated DRSpace A_2 because they found it difficult to parse specific data from a generic log format. This difficulty is not related to coupling and affects only a handful of the operations provided by the DRSpace. Therefore, based on coupling alone, the real difficulty highlighted by the developer could not have been identified as a risk before implementation. However, because this DRSpace represents the Service Facade pattern, and only the

service classes access the log, the risk is isolated within the DRSpace due to log coupling.

DRSpace A_4 was nominated because there were too many subclasses that were complicated to maintain. In Table II DRSpace A_4 is ranked either 2nd or 3rd for CBO, NOC and WMC for all bugs when using absolute or size normalised values. This result is consistent with the developer's justification and supportive of the correlation between architectural coupling and error-proneness. Once again, due to the fact that all subclasses are within the DRSpace the risk is isolated.

TABLE III. SPEARMAN'S RANK FOR OO METRICS AND ERROR-PRONENESS

		Absolute Values				Normalised By Size %			
Bug Sample	**Variable**	**CBO**	**DIT**	**NOC**	**WMC**	**CBO**	**DIT**	**NOC**	**WMC**
All Bugs	ρ	0.841	0.747	0.732	0.673	0.918	0.879	0.860	0.784
	α level	0.001	0.005	0.005	0.010	0.001	0.001	0.001	0.005
Recent Bugs Last Six Months	ρ	0.713	0.510	0.549	0.593	0.868	0.846	0.728	0.743
	α level	0.005	0.050	0.050	0.025	0.001	0.001	0.005	0.005

In the case study, only the root class of the DR was specified in the architectural design document for the group of classes nominated as Package X. Thus it was not possible to form a DRSpace and calculate metrics using our method. The narrative of the design document specified the creator pattern should be used to load and cache objects and that a lazy load policy should be employed for the cache.

The developer's justification for the nomination is that complex synchronisation code is duplicated into subclasses. Synchronisation code is needed to ensure that a second thread waits until the object is loaded upon first access. Therefore, the architectural styles of caching and lazy loading combined with NOC appear to be the reason for Package X being error-prone when considering recent bugs. These "shared constraints" are an example of the "shared secrets" described by Xiao et al [1].

Although Package X did not relate to a DRSpace that could have been formed at the design stage using our method, the abstract creator and its implementation subclasses accounted for 44% of the Bug Space when only recent bugs are considered. This highlights the importance of a comprehensive architectural description for an effective risk analysis. If the DR was documented in the UML class diagrams, and if our method was applied before implementation, some types of mitigations such as re-design, pair programming, and increasing the budget/schedule could have been considered. This is because this would have caused relatively high scores for CBO and NOC.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we present three key observations about DRSpaces formed from UML designs. Firstly, a significant correlation between DRSpace Coupling Between Objects (CBO) computed from the design and error-proneness. Secondly, groups of classes nominated by developers as difficult to implement/maintain fitted neatly inside the DRSpaces (when they could be matched to design DRSpaces). Thirdly, the nominated DRSpaces are the most error-prone when recent bugs are considered. These observations support our novel use of DRSpaces as a basis for structured risk analysis based on UML class diagrams before implementation. Whilst the effect of coupling is well understood, our contribution is to demonstrate that DRSpaces are isolating risk containers because coupling stemming from the DR is contained within the DRSpace, and greater coupling has a higher risk of resulting in implementation errors.

Analysis of the developer nominations suggests that focusing on coupling alone is too crude and that even if DRs are used to partition the architecture for the purpose of risk analysis, other factors must be considered to determine the implementation risks. In their work on technical debt, Kazman et al [2] noted that "one of our lessons learned is that we can influence projects to improve their record-keeping practices". This study shows the importance of comprehensively documenting the architectural designs in both formal (UML) and narrative terms to aid an effective risk assessment before the implementation stage. Future work will further evaluate the method by considering why the developers did not nominate DRSpace A_1 and DRSpace A_3, which had relatively high metric scores and error-proneness. In addition, we would like to use other case studies to verify DRSpaces are meaningful and durable risk containers.

REFERENCES

[1] L. Xiao, L, et al., *"Design rule spaces: A new form of architecture insight."* in *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014. pp. 967-977.

[2] R. Kazman, R, et al., *"A Case Study in Locating the Architectural Roots of Technical Debt,"* in *Proceedings of the 37th International Conference on Software Engineering,* IEEE, 2015. pp. 179-188.

[3] K. Akingbehin, *"A quantitative supplement to the definition of software quality,"* in *Third ACIS International Conference on Software Engineering Research, Management and Applications*, IEEE, 2005. pp. 348-352.

[4] D. Falessi, et al., *"Decision-making techniques for software architecture design." ACM Computing Surveys*, vol. 43, no 4, pp. 1-28, Winter 2011.

[5] S. Rekha and H. Muccini, *"Suitability of software architecture decision making methods for group decisions"*, in *8th European Conference on Software Architecture*, Springer, 2014. pp. 17-32.

[6] P. Bengtsson, et al., *"Architecture-level modifiability analysis (ALMA)."* in *The Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129-147, January 2004.

[7] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*, MIT press, 2000.

[8] M. Naedele, et al., *"Manufacturing execution systems: A vision for managing software development."* in *Journal of Systems and Software*, vol. 101, pp. 59-68, March 2015.

[9] S. R. Chidamber and C. F. Kemerer, *"A metrics suite for object oriented design."* in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, June 1994.

[10] C. Spearman, *"The proof and measurement of association between two things."* in *The American Journal of Psychology*, vol. 15, no. 1, pp. 72-101, 1904.