Software Architecture Risk Containers

Andrew Leigh, Michel Wermelinger, Andrea Zisman

School of Computing & Communications
The Open University, Milton Keynes, United Kingdom

Abstract. Our motivation is to determine whether risks such as implementation error-proneness can be isolated into three types of containers at design time. This paper identifies several container candidates in other research that fit the risk container concept. Two industrial case studies were used to determine which of three container types tested is most effective at isolating and predicting at design time the risk of implementation error-proneness. We found that Design Rule Containers were more effective than Use Case and Resource Containers.

1 Introduction

According to Bass et al. (2012) 161 historical projects were analysed by Boehm and Turner who found that the bigger the project is, the more architecture risk assessment is needed to avoid rework. No results for the comparative performance of architecture evaluation methods for isolating risks were found in existing work. Not knowing the risk scope limits the ability to estimate the risk impact and cost of mitigations. Our proposition is to investigate whether it is more effective to base risk assessment around ***risk containers*** that isolate related risk-inducing elements.

In this paper, we test three types of risk containers for their ability to ***isolate the risk of implementation error-proneness at the design stage***, namely *Design Rule*, *Use Case* and *Resource Containers*. If container level design metrics that indicate a design might be complex to implement (e.g. coupling metrics), can be used to rank containers, then containers can be used to predict the areas of greatest risk. If the degree of element sharing between containers is low, they are said to be element isolating. Furthermore, if containers are risk predicting and element isolating, they must also be risk isolating because the elements in the container are inducing the risk, and they are not shared with other containers. Risk isolating containers would enable practitioners to identify the risk areas and understand their scope in terms of the affected elements. In this paper we address the following research question:

Can the risk of implementation error-proneness be isolated within risk containers based on the design time architectural description?

The remainder of this paper is structured as follows. Section 2 lists the existing work that most closely fits the risk container concept. Section 3 presents the method

used to test three types of containers using two industrial case studies. Section 4 presents analysis of the results. Finally, conclusions are drawn in Section 5.

2 Background

We next present existing work that most closely fits the proposed concept of architecture risk containers. This section is organised by the container types we synthesised from the commonalities we found between architecture evaluation techniques.

Attack Graph Containers are tuples containing nodes that an attacker can interact with to exploit a vulnerability in a goal component (Said et al. 2011). UML models are used to estimate component failure probability to assess scenario security risks. Probabilities assigned to graph elements are used to calculate the probability of failure for the goal component. Since the tuple isolates the elements associated with the risk, attack graphs fit the risk container concept.

Design Rule Containers (DRSpaces) proposed by Xiao et al. (2014), are graphs based on the key interfaces (design rules) that split an architecture into independent modules. The vertices are related classes and the edges are the relationships between those related classes. Xiao et al. concluded that if a leading file of a DRSpace is error-prone, a large proportion of the other DRSpace files are likely to be error-prone, and that most error-prone files will be found in just a few DRSpaces. Xiao et al used a clustering algorithm called Design Rule Hierarchy (DRH) proposed by Wong et al. (2009) to extract DRSpaces from source code. Wong et al. were motivated to develop the DRH algorithm to separate modules of related elements in UML designs to maximise developer parallelism. Leigh et al. (2016) manually populated DRSpaces from UML class diagrams taken from an industrial case study. The term 'Design Rule Containers' is used to standardise terminology in this paper.

Component Containers contain the classes a component is composed of. Stevanetic and Zdun (2016) calculated design metrics from UML to indicate the understandability of components. Their results show that if the internal relationships of a component are difficult to comprehend it might be difficult to maintain and therefore the classes it is composed of isolate the risk. Abdelmoez et al. (2006) estimated requirement maturity and traced it to components to determine component change probability and identify maintainability risks. Goseva-Popstojanova at al. (2003) and Yacoub and Ammar (2002) calculated complexity metrics from designs to assess reliability risks of components. These contributions suggest maintainability and reliability risks could be isolated within Component Containers.

Resource Containers contain the elements dependent on a resource such as a component, service or data store. Stevanetic and Zdun (2016) also showed that if the component functionality is difficult to comprehend, developers might misunderstand how to use it, leading to more errors in dependent code. A Resource Container could be used to isolate elements dependent upon the resource component to isolate the risk.

Scenario Containers contain the elements that support a specific scenario. Williams and Smith (1998) and Cortellessa et al. (2005) used resource estimates (e.g. CPU) to determine whether a scenario is likely to exceed a non-functional require-

ment. Their methods are limited by their dependency on the accuracy of design time resource estimates and assumptions about the target platform. Their results suggest Scenario Containers could isolate performance related risks at design time.

Use Case Containers contain the elements that support a specific use case. Mustafiz et al. (2008) assign a success probability to each use case step. Use cases are then analysed to compare the achievable reliability with the required reliability. The research by Mustafiz et al. and Goseva-Popstojanova et al. suggests reliability risks can be isolated to the set of operations or classes that fulfil the use case.

Despite the methods found being suggestive of risk containers, little evidence about their risk isolation properties is provided. No results regarding the comparative performance of risk containers types for isolating different risks were found. These limitations mean practitioners have no advice for selecting which containers to use for specific risks. For example, the work of Abdelmoez et al. (2006), Xiao et al. (2014), Leigh et al. (2016), and Stevanovic and Zdun (2016) identifies Design Rule and Component Containers as container candidates for maintainability risks. Whilst we know something about specific cases where each have been effective, their relative performance for isolating risks remains unknown.

Sections 3 and 4 present our most recent work to understand how effectively different design time risk containers isolate the risk of implementation error-proneness.

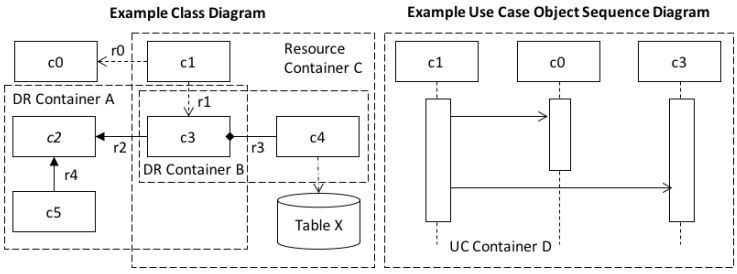
3 Method

This section describes the method used to test how well three risk container types isolate at ***design time*** the ***risk of implementation error-proneness***. These three risk container types have been chosen due to the different ways they split the architecture. Design Rule (DR) Containers group elements subordinate to modularising design rules, Use Case (UC) Containers group elements supporting use cases, and the Resource Containers group elements that depend upon a database table (as opposed to resources such as CPU).

3.1 Risk Container Creation

DR Containers were constructed using the method described in Leigh et al. (2016). Each design rule class was used as a container basis before expansion with subordinate classes by consulting design relationships. For example, in Figure 1 element c2 is the basis of DR Container A because it is an abstract class. Elements c3 and c5 are then added to that container because they are sub-classes. One UC Container was created per use case and included each class referenced by the use case. Note how Figure 1 shows that UC Container D contains all the elements on the use case sequence diagram. Resource Containers were populated with all elements dependent upon a specific database table by seeding with the table encapsulation element and recursively adding elements where the encapsulation element is the ***child*** of a relationship. This can be seen in Figure 1 where element c4 is the basis for container D and c1 and c3 are added because they are recursively dependent upon c4. Additional implementation classes were added to the initially populated containers using strict

name conventions. This was necessary because each design element was realised by an interface and an implementation. Therefore, when an interface was allocated to a container its one to one implementation class was also added. Unlike Xiao et al. (2014) who used automation, our containers were populated by manual analysis.



Example Container Population

Risk Container	Risk Container Type	Based On Feature	Contents
A	DR Container	Abstract Class c2	3 = {c2, c3, c5}
B	DR Container	Composition c3	2 = {c3, c4}
C	Resource Container	Table X	3 = {c1, c3, c4}
D	UC Container	Example Use Case	3 = {c0, c1, c3}

Example Coupling Calculation

DRSpace	Raw Coupling	Divided Coupling
A	3 = 1 (r2) + 1 (r3) + 1 (r4)	2 = 0.5 (r2) + 0.5 (r3) + 1 (r4)
B	2 = 1 (r2) + 1 (r3)	1 = 0.5 (r2) + 0.5 (r3)

Figure 1. Example Container Population and Coupling Calculation

Control containers were created for each container type by randomly allocating the same elements in the test containers to the same number of control containers. The average number of elements per control container and the average number of containers per element was approximately the same as the test containers. The control containers were used to determine whether basing containers on related elements is a better indicator of error-proneness than random population. The control containers are based on the mean values from ten sets of random assignments per container type.

Coupling is a significant contributory factor in the complexity of software (Bass et al. 2012). Thus, it is expected that containers having elements with greater coupling are more likely to be error-prone during implementation. Such a correlation would imply that error-proneness has been isolated to a degree. That is because the error-proneness is associated with coupling stemming from the architectural feature (e.g. Design Rule, Use Case or Resource) on which the risk container is based. If the architectural feature were to be removed, the risk associated with it would be eliminated (and potentially replaced with risk attached to substitute features). Thus, the presence of a correlation between feature based containers and implementation error-proneness implies isolation to the scope of the features on which the containers have been based.

Tightly coupled elements are more likely to change together when software is developed and maintained due to the ripple effect. As stated by Bass et al. (2012), 'reducing the strength of coupling between two modules A and B will decrease the expected cost of any medication that affects A' (p. 122). Therefore, a more precise answer to our research question can be obtained by determining to what degree containers share elements and how much of the coupling is between elements in the same container. That is because if containers are risk isolating the architectural elements

should exist in few containers and elements should have less coupling to other elements outside of their own container.

As per Figure 1, design coupling is calculated for all elements in the container for relationships like aggregation, composition and dependencies if the element is the relationship parent, and generalisations if the element is the child. This metric indicates how tightly coupled container members are to other elements in the architecture.

Error-proneness is defined as the files having more confirmed bugs per thousand lines of released code (KLOC) than a threshold. We used again the 75th percentile of bugs per KLOC as the threshold (Leigh et al. 2016). Bug identifiers were extracted from the Subversion commit messages for each implementation file.

Using our method, we can compare the container types tested by how well they implicitly isolate error-proneness based on correlation between design time container level coupling and implementation error-proneness, and how well they explicitly isolate individual architectural elements and internal coupling. Thus, a strong and significant correlation between coupling and error-proneness, in combination with high isolation metrics, would answer the research question affirmatively.

3.2 Metric Calculation

The following risk container type metrics were used to test their relative capability for predicting the risk of implementation error-proneness and being element isolating:

- *Number of Containers (N):* number of containers the design has been split into.
- *Percent Container Coverage (PCC):* percentage of all implementation elements that were allocated to risk containers. This metric indicates how much of the implementation was represented in the design.
- *Spearman's rank correlation ρ and confidence level α between design coupling and implementation error-proneness:* Spearman's rank correlation coefficient ρ and confidence level α is computed to indicate the association between container level coupling and percentage of error-prone files. The correlation indicates how well the container type predicts and implicitly isolates error-proneness.
- *Mean Containers Per Class (CPC-M):* mean number of containers each element has been allocated to. This metric indicates the average amount of element sharing between containers and represents the degree of element isolation.
- *Upper Quartile Containers Per Class (CPC-UQ):* 75th percentile of containers each element has been allocated to. This metric is used to confirm the degree of element isolation within containers.
- *Mean Percent Internal Coupling (IC-M):* mean percentage of container level coupling that is between two elements inside the same container. This metric indicates the degree of coupling isolated within containers.
- *Upper Quartile Internal Coupling (IC-UQ):* 75th percentile percentage of container level coupling that is between two elements inside the same container. This metric is used to confirm the degree of coupling isolated within containers.
- *Percent Single Neat Container Change Sets (NCC):* percentage of Subversion change sets that fit neatly inside a single risk container. This metric indicates how well containers isolate source code edits made to change the software and fix bugs.

4 Analysis

Two cases studies from the same software company have been used to evaluate our method. The company prefers to remain anonymous, but their name is registered with the Open University in an intellectual property agreement.

4.1 Case Study 1 – API

The first case study is a bespoke Application Programming Interface (API) that enables clients to integrate with a database in an enterprise solution. The architectural description of the API is a UML model and the implementation contains 87.85 KLOC of object-oriented Java code. Table 1 shows the API results.

	API Case Study								
API Case Study	**Coverage**		**Risk Predicting**		**Element Isolating**				
Container Type	**N**	**PCC**	**ρ**	**α**	**CPC-M**	**CPC-UQ**	**IC-M**	**IC-UQ**	**NCC**
Control DR Containers	13	80.90	-0.05	>0.100	1.08	1.00	9.03	12.48	26.40
Control UC Containers	36	12.13	0.38	0.025	5.29	6.60	12.66	17.24	0.49
Control Resource Containers	23	32.81	0.04	>0.100	6.37	9.00	24.09	28.53	0.00
DR Containers	13	**80.89**	**0.85**	**0.001**	**1.08**	**1.00**	**35.33**	**55.56**	**41.33**
UC Containers	36	12.13	0.71	0.001	4.74	5.00	18.52	23.08	2.22
Resource Containers	23	30.81	0.63	0.001	7.77	8.00	**37.37**	**41.54**	0.00

Table 1. Case Study 1 Results

DR Containers have the strongest (ρ) and most significant (α) correlation. The random assignment of elements to containers resulted in a negative correlation for the control DR Containers. DR Containers had the lowest mean containers per class (CPC-M). On average, each element is allocated to just over one DR Container. This contrasts with UC Containers and Resource Containers where each element is allocated on average to approximately 5 and 8 containers respectively. The 75th percentile (CPC-UQ) confirms that elements are distributed across fewer DR Containers than UC and Resource Containers. The mean percentage of coupling where both related elements are inside the same container (IC-M) is greatest for Resource Containers and DR Containers. However, the 75th percentile (IC-UQ) is greater for DR Containers which suggests they typically have less external coupling than Resource Containers.

The percentage of Subversion change sets fitting neatly inside a single container (NCC) is greatest for DR Containers. This result is expected because DR Containers have the highest IC-M/IC-UQ and lowest CPC-M/CPC-UQ, which suggests that DR Containers better isolate the related source code files developers must edit when making changes or fixing bugs in the software. All test containers have stronger correlation and are more risk isolating than their corresponding control containers.

In Leigh et al. (2016) we asked developers to nominate areas of the API that were difficult to implement and maintain. We observed that 2 of 3 nominations fitted neatly inside a DR Container. It is worth noting that none of the nominations could be matched to the API UC Containers. Some elements belonging to the nominated areas could be matched to Resource Containers but a Resource Container that fitted the

whole nomination could not be found. This further strengthens the evidence for DR Containers being the most isolating container type in the API case study.

4.2 Case Study 2 – Server

The second case study is concerned with the Server side modules of a COTS data management application. The Server architecture is documented in MS Word documents and the implementation contains 333.55 KLOC of procedural Oracle PL/SQL code. Table 2 shows the Server results.

	Server Case Study								
Server Case Study	**Coverage**		**Risk Predicting**		**Element Isolating**				
Container Type	**N**	**PCC**	**ρ**	**α**	**CPC-M**	**CPC-UQ**	**IC-M**	**IC-UQ**	**NCC**
Control DR Containers	9	12.50	-0.07	>0.100	1.14	1.00	7.45	6.25	23.48
Control UC Containers	68	5.60	0.03	>0.100	6.16	9.70	0.04	0.00	3.26
Control Resource Containers	16	9.48	0.02	>0.100	2.64	5.20	9.19	10.99	0.00
DR Containers	9	**12.50**	**0.92**	**0.001**	**1.14**	**1.00**	**48.61**	**100.00**	**23.19**
UC Containers	68	5.60	0.32	0.005	5.69	8.00	0.00	0.00	3.26
Resource Containers	16	9.48	0.31	0.250	2.77	6.00	15.14	6.00	0.00

Table 2. Case Study 2 Results

DR Containers again had the strongest (ρ) and most significant (α) correlation. The strong correlation observed for UC Containers in the API case study was not reproduced. DR Containers again have the lowest mean CPC-M and CPC-UQ indicating elements are shared between fewer DR Containers than UC and Resource Containers. DR Containers also have the highest IC-M and IC-UQ which again suggests DR Containers have less coupling to external elements. The change set isolating results for the API were not reproduced in the Server because NCC is approximately the same for the test containers as their corresponding controls.

In both case studies DR Containers cover more of the design (PCC) than UC and Resource Containers. This means more of the risk was isolated into DR Containers. The much lower PCC values calculated for the server were due to more time having passed since the design was produced. This meant more implementation elements were present that were not documented in the design.

5 Conclusion

This paper presents the results of testing three types of risk containers to determine their relative efficacy at isolating the risk of implementation error-proneness at design time. The three types tested were Xiao et al.'s (2014) DRSpaces, adapted to split the architectural design by modularising design rules, and two novel containers that group elements supporting use cases, and elements dependent upon databases.

Results from two industrial projects suggest DR Containers are the most effective at isolating the risk of implementation error-proneness at design time. This is due to them having the strongest correlation between container level design coupling and implementation error-proneness and least amount of element sharing and external coupling in both case studies. The results strengthen our previous evidence (Leigh et

al., 2016) that DR Containers can be used for design time assessment of software architectures for the risk of implementation error-proneness, based on UML class diagrams or module dependency graphs.

Whilst DR Containers are the most effective of the three container types tested, even more effective risk containers may remain to be found. Further investigation is also needed to understand why the high number of change sets fitting neatly inside a single DR Container in the API was not observed in the Server case study. Furthermore, work is required to determine whether container based risk assessment is generalizable for other risks, and if so, whether the same containers or others work best, and at which levels of architecture abstraction different container types are effective. More work is also required to determine how meaningful different container types are to software practitioners and how durable they are throughout the software development life-cycle. These questions represent opportunities for future work.

6 References

1. Abdelmoez, W.M., Goseva-Popstojanova, K. and Ammar, H.H. (2006). *Methodology for maintainability-based risk assessment*. In RAMS'06. Annual Reliability and Maintainability Symposium, IEEE, pp. 337-342.
2. Bass, L., Clement, P. and Kazman, R. (2012). *Software Architecture in Practice*, 3rd Ed., Addison Wesley, Reading, USA, pp. 121-124 and p. 280.
3. Cortellessa, V., Goseva-Popstojanova, K., Appukkutty, K., Guedem, A.R., Hassan, A., Elnaggar, R., Abdelmoez, W. and Ammar, H.H. (2005). Model-based performance risk analysis. *IEEE Transactions on Software Engineering*, 31(1), pp. 3-20.
4. Goseva-Popstojanova, K., Hassan, A., Guedem, A., Abdelmoez, W., Nassar, D.E.M., Ammar, H. and Mili, A. (2003). Architectural-level risk analysis using UML. *IEEE transactions on software engineering*, 29(10), pp. 946-960.
5. Leigh, A., Wermelinger, M. and Zisman, A. (2016). *An Evaluation of Design Rule Spaces as Risk Containers*. In Proc. of the 13th Working Int. Conf. on Software Architecture (WICSA), IEEE, pp. 295-298.
6. Mustafiz, S., Sun, X., Kienzle, J. and Vangheluwe, H. (2008). Model-driven assessment of system dependability. *Journal of Software and Systems Modelling*, 7(4), pp. 487-502.
7. Said, F.H., Ammar, H.H., Valenti, M.C., Ross, A. and Lai, H.J. (2011). *Security-based Risk Assessment for Software Architecture*. West Virginia University Libraries, pp 1-126.
8. Stevanetic, S. and Zdun, U. (2016*). Exploring the Understandability of Components in Architectural Component Models using Component Level Metrics and Participants' Experience*. In 19th Int. ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), IEEE, pp. 1-6.
9. Williams, L.G. and Smith, C.U. (1998). *Performance evaluation of software architectures*. In Proc. of the 1st Int. workshop on Software and performance, ACM, pp. 164-177.
10. Wong, S., Cai, Y., Valetto, G., Simeonov, G. and Sethi, K. (2009). *Design rule hierarchies and parallelism in software development tasks*. In Proc. of the 24th Int. Conf. on Automated Software Engineering (ASE), ACM, pp. 197–208.
11. Xiao, L., Cai, Y. and Kazman, R. (2014). *Design rule spaces: A new form of architecture insight*. In Proc. of the 36th Int. Conf. on Software Engineering, ACM, pp. 967-977.
12. Yacoub, S.M. and Ammar, H.H. (2002). A methodology for architecture-level reliability risk analysis. *IEEE Transactions on Software engineering*, 28(6), pp. 529-547.