The Journal of Systems & Software 176 (2021) 110947

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Evaluating the effectiveness of risk containers to isolate change propagation ☆

Andrew Leigh, Michel Wermelinger *, Andrea Zisman

The Open University, Computing and Communications Department, Milton Keynes, MK7 6AA, United Kingdom

ARTICLE INFO

Article history:
Available online 10 March 2021

Keywords:
Software architecture
Change propagation
Maintainability
Risk
Technical debt

ABSTRACT

Previous studies indicate that error-proneness risks can be isolated into risk containers created from architectural designs, to help detect and mitigate such risks early on. Like error-proneness, change propagation may lead to higher implementation and maintenance costs. We used automated tools to analyse four software development projects using three risk container types, each type based on a different architectural perspective. A strong and significant correlation between design change propagation and implementation co-change was observed for all three container types. We found that Design Rule Containers (DRCs), based on class diagrams, are the most effective for isolating change propagation because they have the least amount of container overlap, highest levels of internal coupling, highest co-change probability between classes that share containers, and the most change sets isolated in containers. Developers from two projects were able to justify why design dependencies had resulted in the top five DRCs being predicted to isolate the most change propagation. This and the previous error-proneness research suggests DRCs are an effective technique to detect and contain code maintainability risks at the design stage. These results provide some evidence that class diagrams are more useful than use case sequence diagrams for analysing maintainability risks in designs.

1. Introduction

Unmanaged risks have been identified as a common factor in software projects that failed in terms of budget and schedule (Charette, 2005). Our aim is to help practitioners identify and mitigate maintainability risks early in the life-cycle, to increase the chances of project success. This is important because "maintainability and development costs are in exponential relationship with each other" (Bakota et al., 2012, p. 1).

Our approach digests an architecture into smaller chunks of elements, termed risk containers, which can be ranked by relative risk indicators and easily reasoned about. Ranking enables allocation of resources to areas of greatest risk, while a container's scope shows the potential risk impact. Bouwers et al. (2010) explain that smaller architectural chunks can be easily understood. Their view is supported by our experimental results (Leigh et al., 2019): participants find it easier to locate cyclic dependencies in smaller risk containers than in the overall architecture diagram.

The technical insight gleaned from the containers and their risk indicators support impact assessment and better inform risk management (e.g. re-design, implementation mitigations, or risk acceptance). Managers could track risk containers in project risk registers to better manage quality, schedule and cost compromises with technical teams, and avoid unnecessary technical debt whilst driving project progress forward.

Our previous work (Leigh et al., 2017) compared Design Rule Containers to Use Case Containers and Resource Containers. Design Rule Containers are based on Design Rule Spaces (Xiao et al., 2014) populated from UML class diagrams (Fig. 1). Resource Containers are seeded with resource (database, service, file) encapsulation classes with dependent classes recursively added (Fig. 2). Use Case Containers have classes in the sequence diagram of a use case (Fig. 3).

Risk containers created from designs enable projects using upfront design to isolate error-proneness risks before they become a problem. This is advantageous because errors found during design are less expensive to correct than errors found during testing (Akingbehin, 2005). Forming containers from design diagrams allows programming language independence.

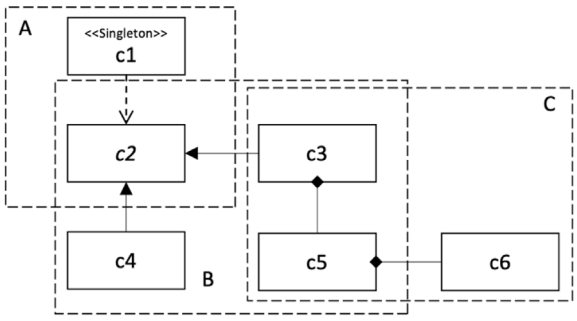
We selected these container types because they represent different and commonly used approaches to decomposing software systems from different architectural perspectives. They allow us to evaluate container types, perspectives, and diagrams that are helpful for risk analysis.

Our previous work (Leigh et al., 2016, 2017) was limited to one maintainability risk, namely error-proneness, which gave

☆ Editor: Doo-Hwan Bae.

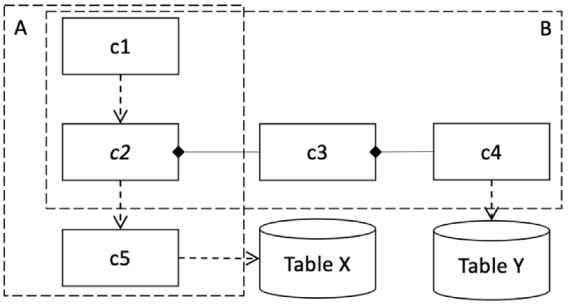
* Corresponding author.
E-mail addresses: andrew.leigh@open.ac.uk (A. Leigh), michel.wermelinger@open.ac.uk (M. Wermelinger), andrea.zisman@open.ac.uk (A. Zisman).

https://doi.org/10.1016/j.jss.2021.110947
0164-1212/



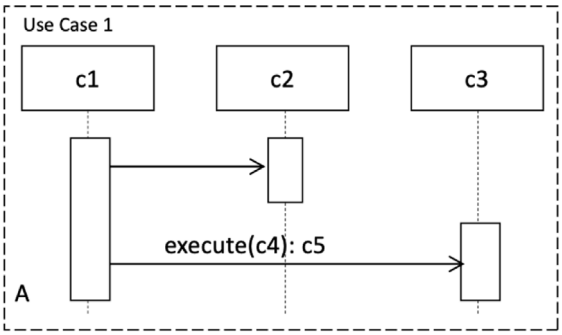
DR Container	Design Rule Type	Roots	Contents
A	Pattern	c1	c1, c2
B	Abstraction	c2, c3, c4	c2, c3, c4, c5
C	Composition	c3, c5, c6	c3, c5, c6

Fig. 1. Design rule container example.



Resource Container	Resource	Root	Contents
A	Table X	c5	c1, c2, c5
B	Table Y	c4	c1, c2, c3, c4

Fig. 2. Resource container example.



UC Container	Use Case	Roots	Contents
A	Use Case 1	c1, c2, c3, c4, c5	c1, c2, c3, c4, c5

Fig. 3. Use case container example.

no insights about the generalizability of risk container types. This paper addresses that limitation with the following research question:

Can implementation change propagation risks be isolated within risk containers created from a design time architectural description?

Curtis et al. (2012) define technical debt as "the cost of fixing quality problems in production code that the organization knows must be eliminated to control costs or avoid operational problems". Areas of excessive change propagation could be classified as technical debt. Nugroho et al. (2011) explain that technical debt accrues compound interest. It is advantageous to address such technical debt as early as possible, which supports the need for approaches based on upfront designs.

Similar to error-proneness, change propagation is relevant to all software development projects. Highly coupled classes are more likely to change together when software is developed and maintained due to the ripple effect (Lindvall et al., 2003). 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 modification that affects A". Hassan and Holt (2004) add that "the dangers associated with not fully propagating changes have been noted and elaborated by many researchers" and cite Brooks (1995) as cautioning about the risk associated with developers losing grasp of a system as it evolves.

Wolfe and Horowitz (2017) explain that attention towards a search field of objects is modulated by five factors because limits on visual processing make it impossible to recognize everything at once. This implies that the size and complexity of the search scene will influence how quickly an individual can locate objects of interest contained within it. Therefore, if the time to complete a task is limited, as it is usually the case in software development, objects of interest are more likely to be missed in larger and more complex search fields. Thus, greater complexity in design structures will increase the chances of software developers misunderstanding and not fully appreciating all of the discrete program modifications needed to correctly propagate an intended change. We presented some evidence to support this view in Leigh et al. (2019): when asked to identify the impact of design changes, participants made no errors if changes spanned multiple simpler container diagrams, but some participants assessing the same change on a single more complex diagram made errors. There have been further participants since that publication, but it remains the case that just one single error was observed in cases where eight participants were asked to impact changes spanning multiple container diagrams. This compares to one error being observed when just four participants had to impact the same change on a single diagram. The lower error frequency observed for the container participants supports the views or Wolfe and Horowitz.

The risk associated with developer oversights, misunderstanding and unfamiliarity leading to changes not being fully propagated can result in two potential impacts: (1) required modifications could be missed during change cost estimation resulting in the effort to implement change requests being underestimated; and (2) hard to find bugs might be inadvertently introduced that increase maintenance costs when they have to be fixed. Thus, excessive change propagation is a risk because it has the potential to unexpectedly inflate the cost of maintaining highly coupled parts of a software system. Isolation of change propagation risks in containers would support better understanding and estimation of the scope of change and will allow the change to be implemented with less risk helping to reduce unexpected impacts on the budget and schedule.

In this paper we make several contributions: (i) an evaluation of each risk container type's ability to predict and isolate implementation change propagation; (ii) identification of the most change propagation isolating container type; (iii) and formally defined container level change propagation metrics. We extend our previous work with: (iv) analysis of additional projects; (v) more detailed descriptions of the container population algorithms; and (vi) a formal definition of our class isolation metrics.

The rest of this paper is organized as follows. In Section 2 we present an account of related works including risk containers, change propagation, and isolating change propagation. In Section 3 we describe the method used in our work. In Section 4 we report our experimental results. Finally, in Section 5 we conclude the paper and discuss future work opportunities.

2. Related work

In this section we review work on risk containers and change propagation within software architectures.

2.1. Risk container

The notion of risk container was first proposed in (Leigh et al., 2016), when we evaluated Design Rule Spaces (DRSpaces) for isolating implementation error-proneness. DRSpaces (Xiao et al., 2014) are graphs based on modularizing design rules (key interfaces) that split an architecture into independent modules. A module refers to parts of the system that are decoupled from each other by design rules. For example, a module might be an abstraction and its implementations (inheritance hierarchy) because those classes modularize chunks of related functionality for use by the rest of the system. Related functionality is modularized into the inheritance hierarchy because all members of the inheritance hierarchy provide common methods (i.e., methods with the same interface) to the rest of the system, whether they have different implementations per hierarchy member, or whether members provide the implementation they inherit from a super class. Therefore, from the perspective of classes outside of the inheritance hierarchy, the inheritance hierarchy constitutes a module of related functionality (common methods).

Xiao et al. extracted DRSpaces from source code using (Wong et al.'s 2009) Design Rule Hierarchy (DRH) algorithm, whereas Design Rule Containers (Leigh et al., 2016, 2017) were populated from Unified Modelling Language (UML) designs. These different approaches reflect different motivations. Xiao et al. provided architectural insights into error and change-proneness of source code (Xiao et al., 2014), and we determined whether implementation error-proneness risks can be identified in upfront designs (Leigh et al., 2016, 2017).

When we compared container types for their ability to predict and isolate error-proneness we found that Design Rule Containers share fewer classes and isolate more structural coupling than Resource and Use Case Containers. This observation, allied to a strong and significant correlation between container structural design coupling and container implementation error-proneness, suggests that Design Rule Containers are the most effective of the three types tested for isolating error-proneness (Leigh et al., 2017).

According to Lehnert's (2011) taxonomy for software architecture change impact analysis techniques, Design Rule Containers can be categorized as a global analysis of architecture model scope with an input (artefact) granularity of classes, and an output (results) granularity of sub-systems (modules). Use Case and Resource Containers do not fit neatly into Lehnert's output classification scheme because the available input/output entity granularities are system, sub-system, component, interface, class, and method.

2.2. Change propagation

Change propagation is an important factor in maintainability. Martin (1997) suggested the Stable Dependencies Principle (SDP): if A depends on B, A should be less stable (less resistant to change) than B in order to ease any necessary propagation of changes from B to A. Wermelinger et al. (2011) assessed eight years of Eclipse's architectural evolution, observing that the SDP and other dependency-based principles were part of Eclipse's success in sustaining continuous evolution.

Bengtsson et al. (2004) based their Architecture Level Modifiability Analysis (ALMA) method on assessing architectures using a set of predefined change scenarios. The architecture that could be adapted to support the most change scenarios with minimal impact is deemed to be the most modifiable. ALMA relies on subjective interviews of the designers to gather the likely change scenarios and is limited because it can only deal with business change requirements that have already been identified.

The approach in Clarkson et al. (2004) is not specific to software design. It requires every pathway between all components to be determined using a component dependency graph. The pathways between two components are used to calculate the combined risk of change propagation. The initial probabilities for change propagating between two components are derived from observed data. This method conflicts with our goal of design-time risk assessment because data on observed co-change is not available prior to implementation. The use of metrics allows an architecture to be assessed for arbitrary changes as opposed to ALMA's limited set of predicted change scenarios.

Abdelmoez et al.'s (2005) method does not require data on observed co-changes. Instead of calculating the change propagation probability from component dependencies, their method is based on the usage of public attributes and function parameters of class A by class B. This requires a higher fidelity diagram such as an object sequence diagram. In this case, change propagation is assessed to a higher fidelity than the cruder overall component level dependencies used in Clarkson et al. (2004). Shaik et al. (2006) reported that (Abdelmoez et al.'s 2005) change propagation metric of is "helpful and effective in assessing the design quality of software architectures".

Hassan and Holt (2004) tested four change propagation predicting heuristics on five systems. The four heuristics were based on: (i) the entities changed by the same developer, assuming that specific developers maintain specific areas of coupled code; (ii) co-change; (iii) call, use and dependency relations; and (iv) code structures defined in the same file. Average precision of the predicted versus the actual changes was always low. Recall was greater than 0.74 for all heuristics apart from (iii). Hassan and Holt concluded their "results cast doubt on the effectiveness of code structures such as call graphs as good indicators of change propagation" (p. 9) as used by Abdelmoez et al.

Rostami et al. (2015) attempt to address these limitations by enhancing the architecture with additional context information and providing tooling to recommend the software units impacted by a change scenario. The context information can be drawn from various sources including source code files, technologies, build configuration, test cases, allocation context, deployment and personnel. The authors found that for most change scenarios tested, participants who used their tooling were able to identify change impacts more precisely than a control group and expert group who did not use their tool. Whilst the benefits of allying automation with a diverse range of contextual information are clear, many of the context sources are not available at the design stage.

Mo et al. (2016) proposed the Decoupling Level (DL) metric to assess how decoupled architectural modules are from each other. Their work on DL extends the work with the DRH algorithm already discussed (Wong et al., 2009). The overall DL for an architecture is the sum of the DLs for each layer in the DRH. As specified, DL does not support our goal because its formula is based on source code files rather than design diagrams. DL also does not support our goal of ranking parts of an architecture w.r.t. change propagation risk. Although the DRH-level DL values could be used to rank layers, they could not be used to rank containers.

2.3. Isolating change propagation

Wong et al.'s (2009) motivation for the DRH algorithm was to separate modules of related classes in UML designs to maximize

developer parallelism. For their proposition, changes should not propagate beyond the module boundaries. The authors observed lower levels of communication between developers working on different modules in the same layer than those working across layers of the DRH. Kazman et al. (2015) provided further evidence by using DRSpaces to identify that change-prone files are highly architecturally connected, i.e. classes with a high degree of coupling are more likely to change together. We found that Design Rule Containers have a higher degree of internal coupling and isolate more change sets than the other types of containers tested (Leigh et al., 2017).

Whilst all this evidence is supportive, it does not directly test whether change propagation risks can be isolated into containers, which is the purpose of the work presented in this paper. If container-level change propagation metrics are effective, it would enable designs to be improved to increase developer parallelism and reduce the cost of implementation changes. In addition, it would provide evidence that the risk container technique (Leigh et al., 2016, 2017) is applicable to change propagation risks, as well as error-proneness risks. Moreover, it will identify the most effective type of risk container for change propagation.

ID	Class Name	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	APIList		G	G	G	G	G	G	G	G	G	G	G	G											
2	AssociationList	G																D							
3	ValidAssociationList	G													D										
4	SystemList	G															D								
5	MetadataCommonList	G																		D					
6	LocationList	G																	D						
7	ValidStyleList	G																			D				
8	PropertyList	G																							D
9	ValidLocationList	G																				D			
10	ExternalValueList	G																							
11	ValidComponentList	G																						D	
12	ValidPropertyList	G																					D		
13	DescriptionList	G														D									
14	ValidAssociation			D														A				A	A	A	
15	Description													D											
16	System				D												CA	A	C		A			A	
17	Association		D												A		A	A	A						A
18	Location					D											C	A				A			
19	NamedValue				D																G				
20	ValidStyle						D										A			G				C	
21	ValidLocation							D							A				A						
22	ValidProperty											D			A										A
23	ValidSystem										D				A		A				C			C	
24	Property								D									A					A		

Fig. 4. Design Rule Container population A = Aggregation, C = Composition, D = Dependency, G = Generalization..

3. Method

We describe our method to determine whether change propagation can be isolated into risk containers created from designs. Due to the ripple effect (Lindvall et al., 2003; Bass et al., 2012), we hypothesize that greater structural coupling between classes results in greater predicted change propagation probability between those classes (e.g. if the interface of an abstract method changes, all subclasses change). This should result in greater observed co-change within containers. We consider direct structural coupling as opposed to indirect coupling via data.

We used the same overall test construct as for error-proneness in Leigh et al. (2017). Firstly, we consider the degree of overlap (class sharing) between containers. If it is low, the containers are considered to be class isolating. Secondly, we determine whether container-level change propagation metrics predict implementation co-change. Containers of a given type (design rule, resource, use case) are considered risk predicting if container-level change propagation metrics calculated from the design correlate to implementation co-change. If the containers are risk predicting and class isolating, they must also be risk isolating because the classes in the container are inducing the risk, and they are not shared with other containers. Thirdly, we tested whether classes sharing containers are more likely to change together than classes that do not share containers, and which type of container isolated the most change sets, in order to corroborate whether containers isolate change propagation. Finally, we discussed the results with developers in order to establish whether the change propagation observed could be explained in terms of coupling problems, and determine whether the developers recognized the ranked containers as entities about which they could reason.

3.1. Software used in the study

To perform a fair evaluation between the three container types, we considered projects with: (i) a structural graph indicating the design dependencies between classes (e.g. a class diagram) with at least five design rules; (ii) at least five clearly defined resource encapsulation classes; (iii) a mapping between at least five use cases and the classes designed to fulfil them (e.g. use case sequence diagrams); (iv) a change history of at least 100 revisions; and (v) correspondence between the classes and relationships documented in the design and those found in the implementation. We hypothesize that the change propagation probability calculated from a design should correlate to change propagation during its implementation (co-change). In order to test this hypothesis, we used Spearman (1904) rank correlation coefficient, which is a statistical measurement of the association between two variables. The value of five was selected for criteria (i-iii) because five is the minimum sample size needed to achieve a significance level of 0.05 using a two-tailed test based on the critical values of the Spearman's rank correlation coefficient (Zar, 1984, Table B.19).

Further to the base criteria (i–v), it was also important to consider: (vi) industry projects to support the goal of helping practitioners; (vii) projects from multiple sources to ensure that results are not skewed by specific team practices; (viii) at least one project where we were able to interview developers to check they agreed that the change propagation predicted was grounded in coupling problems; and some generalizability criteria (ix) projects of different size; (x) projects including different software functionality; and (xi) projects implemented in different programming languages.

In our study we used four projects, as follows. We used two industrial projects from a software development company. The first project contains UML class and sequence diagrams of an Application Programming Interface (API) implemented in Java to enable clients to integrate with a database in an enterprise solution. This project was used in our previous error-proneness research (Leigh et al., 2016, 2017), in which developers were asked to identify which Design Rule Containers were most likely to contain error-prone classes. They identified container A_4 as being difficult to implement and maintain. Fig. 4 represents A_4 as a Design Structure Matrix with four relationship types: Aggregation, Composition, Dependency, and Generalization.

The developer explained "there seem to be too many subclasses to maintain and they are very complicated". The former part of the explanation alludes to the problem being rooted in change propagation, i.e. the subclasses are changed together during maintenance. We would expect our method to predict a relatively high risk of change propagation isolated in container A_4 when compared to other API Design Rule Containers.

The second project (Server) represents the server-side modules of a data management application. Its design was described as a series of PL/SQL package dependency specifications (dependency graph), and use case descriptions identifying the packages supporting each use case. We assumed PL/SQL packages are analogous to Java classes because both represent nodes when Java

Table 1
Software projects studied.

Name	Source	Design		Implementation				
		UML	Classes	Language	Classes	KLOC	Revisions	Coverage %
API	Industry	Yes	188	Java	445	87.85	2179	42
Server	Industry	No	31	PL/SQL	232	258.05	2202	13
Ps2tsa	Academia	Yes	36	Java	43	3.78	1040	84
OBTW	Academia	Yes	34	Java	66	12.50	105	52

Table 2
Design container coverage of implementation after N revisions.

Name	Number of design containers			Implementation coverage %		
	DRCs	RCs	UCC	DRCs	RCs	UCCs
API	15	23	35	59.78	45.62	16.18
Server	9	14	68	12.50	9.91	6.47
Ps2tsa	11	7	10	79.07	34.88	32.56
OBTW	9	8	12	46.97	43.94	25.76

and PL/SQL code is transformed into a dependency graph for input into our method, i.e. both represent low level software units (graph nodes) that other graph nodes (other Java classes or PL/SQL packages) depend upon (indicated by graph edges). This enabled us to extract the package relationships needed to construct the containers for a system without a UML design. We also used the Server project in our work with error-proneness risks (Leigh et al., 2017).

To obtain non-industry projects we searched the Lindholmen dataset (Hebig et al., 2016) of open source projects using UML. We selected the OpenBundestagsWahl (OBTW) (Löwer et al., 2014) project on GitHub because its design includes classes that encapsulate data retrieved from comma-separated data file resources, enabling all three container types to be populated. The project analyses German election results.

The fourth software project is ps2tsa, a companion application for the Planetside 2™ computer game (http://www.planetside2.com, accessed 9th December 2018). It presents a map of the ensuing game play to help players make better tactical decisions. Ps2tsa was selected because it meets criteria (i–v) and we could speak with the student to confirm which classes encapsulated resources.

Table 1 summarizes the key characteristics of each project. Coverage indicates what percentage of the Java classes or PL/SQL packages could be allocated to design time risk containers, i.e. how much of the potential implementation risk could have been analysed during design using each container type. Having correspondence between the design and implementation (criteria v) is essential to the integrity of the research. Correlation between change propagation predicted from the design and implementation co-change would not be expected without correspondence. The number of design classes shown in Table 1 indicates the subset of implementation classes that were traced back to the upfront design by name. It is to be expected that the academic projects should have the highest coverage because they had undergone very little maintenance once the students had completed them. Inspection of the industry designs revealed that the designs had not been kept up to date during the maintenance phase. Despite this limitation, calculating Coverage (in Table 1) as the percentage of implementation classes found in the design, demonstrates that the research has the potential to investigate the change propagation risk for 13% (31/232 packages) of the Server project after seven years of maintenance and 2202 revisions, and for 42% (188/445 classes) of the API project after three years of maintenance and 2179 revisions.

Coverage of the implementation by the risk containers was highest for Design Rule Containers in all projects, as shown in Table 2. Risk container coverage is important because it indicates how much of the implementation can be assessed for change propagation risk using our method. Only 42% of the API implementation classes were documented in the design. However, many classes deliberately have a 1:1 interface to hide implementation details from users of the API. In these cases, the interface and its sole implementation are to all intents and purposes a single class and we assigned both the interface and its implementation to the appropriate risk container, even though only the interface is referenced in the design diagram. This is why API coverage is higher than 42% in Table 2 for Design Rule and Resource Containers.

3.2. Container definition

For all projects we created three sets of containers: Design Rule Containers (DRCs), Use Case Containers (UCCs) and Resource Containers (RCs). All three types of containers are initially populated with roots: DRC roots are the classes that represent the design rule; UCC roots are all classes referred to by a use case diagram; and the RC root is the resource encapsulating class. For example, in Fig. 1 the root of container A (a pattern design rule) is class c1, the roots of container B (an abstraction design rule) are classes c2, c3, and c4, and the roots of container C (composition design rule) are classes c3, c5, and c6. In Fig. 2 the root of container A is class c5 and the root of container B is class c4. In Fig. 3 classes c1, c2, c3, c4, and c5 are the roots of container A.

Once populated with their roots, DRCs and RCs are then expanded according to algorithms shown in Figs. 5 and 6, using the relationships between classes found in the design. Expansion ensures DRCs represent the whole ‘module’ and RCs represent all items dependent upon the resource. UCC containers are not expanded beyond their roots because it is assumed that the use case sequence diagram includes all classes needed to fulfil the use case. When working with our industry partner we confirmed with the developers, because we did not have the source code, that the classes we had identified as roots of Resource Containers were in fact resource encapsulating to help ensure a fair comparison with the other container types whose roots are more explicit in the design.

DRCs were constructed using the method described in (Leigh et al., 2016, 2017). As explained by Xiao et al. (2014), design rule classes are the key interfaces that separate an architecture into independent modules. They stem from Baldwin and Clark’s design rules (2000). Our method supports three types of design rules: abstractions, compositions and patterns. Fig. 5 presents the population rules for DRCs (and our interpretation of DRSpaces) as an object-oriented pseudo code algorithm. Once DRCs have

```
class Relationship {
    source : Class -- Generalisation sub-class or Aggregation/Composition/Generalisation source
    target : Class -- Generalisation super-class or Aggregation/Composition/Generalisation target
    type : String  -- A=Aggregation, C=Composition, D=Dependency, G=Generalisation
    getSource() { return self.source }
    getTarget() { return self.target }
    getType() { return self.type }
}

class DesignGraph {
    edges : Set
    addEdge(dependency : Relationship) { self.edges.add(dependency) }
    getEdges(): Set { return self.edges }
    getDirectDependencies(sourceClass : Class, type : String): Set {
        Set directDependencies = Set.create()
        forAll(dependency : self.getEdges()) {
            if (dependency.getSource() == sourceClass and dependency.getType() == type) {
                directDependencies.add(dependency);
            }
        }
        return directDependencies
    }
}

class Container {
    name : String
    members : Set
    setName(name : String) { self.name = name }
    add(member : Class) { self.members.add(member) }
    addAll(members : Set) { self.members.addAll(members) }
    getMembers(): Set { return self.members }
}

class ContainerAlgorithms {

    -- Populates a Design Rule Container for a given set of root classes.
    -- name is the name of the Design Rule Container to be created.
    -- roots for an abstraction design rule is the members of an inheritance hierarchy.
    -- roots for a composition design rule is all of the classes in composite structure.
    -- roots for a pattern design rule is all of classes that implement a design pattern.
    -- graph represents the dependencies between classes.
    -- returns a fully populated Design Rule Container
    populateDRC(name : String, roots : Set, graph : DesignGraph) : Container {
        -- Create the container and add the roots
        Container drc = Container.create()
        drc.setName(name)
        drc.addAll(roots)

        -- Expand with classes that the root(s) depend upon directly
        forAll(root in drc.getMembers()) {
            drc.addAll(graph.getDirectDependencies(root, "A"));
            drc.addAll(graph.getDirectDependencies(root, "C"));
            drc.addAll(graph.getDirectDependencies(root, "D"));
        }

        return drc
    }
```

Fig. 5. Design rule container population algorithm.

```
        }

        return drc
    }

    -- Populates a Resource Container for a given root class.
    -- name is the name of the Resource Container to be created.
    -- root is the resource encapsulating class.
    -- graph represents the dependencies between classes.
    -- returns a fully populated Resource Container.
    populateRC(name : String, root : Class, graph : DesignGraph) : Container {
        -- Create the container and add the root
        Container rc = Container.create()
        rc.setName(name)
        rc.add(root)

        -- Recursively expand with classes that depend upon the root
        expandRC(rc, graph)

        return rc
    }

    expandRC(rc : Container, graph : DesignGraph) {
        int size = rc.getMembers().size()
        forAll(member : rc.getMembers()) {
            forAll(dependency : graph.getEdges()) {
                if (member == dependency.getTarget()) rc.add(dependency.getSource())
            }
        }
        if (rc.getMembers().size() > size) expandRC(rc, graph); -- recursive
    }

}
```

Fig. 6. Resource container population algorithm.

been initially populated with their roots based on patterns, abstractions or compositions (line 47 of the Fig. 5 algorithm), they are expanded with direct aggregations and dependencies (lines 51–53).

For an abstraction DRC (first rule of the algorithm), N is the Depth of the Inheritance Tree (Chidamber and Kemerer, 1994). In Fig. 1, classes c2, c3, and c4 are roots of container B and are added by line 47 of the algorithm because they represent an inheritance tree. Once the roots (c2, c3, and c4) have been added the container is then expanded with classes that the root members depend upon. Therefore, class c5 is also added by line 52 because it is a composition dependency of c3; c6 is not added because it is not coupled to the abstract class or its sub-classes.

The second rule of the algorithm leads to the creation of Container C in Fig. 1 because classes c3, c5, and c6 represent a composite structure. For a composition DRC, all members of a composite structure are added as the container roots, e.g. classes c3, c5, and c6 are added as the roots to container C by line 47 of the algorithm. Lines 51–53 add no further classes because the composition classes do not depend on any other classes.

In Fig. 1, the designer chose to base class c1 on the singleton design pattern and container A is an example of a pattern design rule (third rule of the algorithm). Therefore, the sole root class of container A is class c1 and is added by line 47 of the algorithm. If, for example, a model-view-controller pattern had been used instead, the model, view and controller classes would be the container's root classes and be added by line 47. The only other class added to A is c2 because it is the only direct dependency (or aggregation) to which the pattern classes are structurally coupled. Class c2 is added by line 53.

Fig. 6 expresses the Resource Container population rules as an object-oriented pseudo code algorithm. The basis for each container is a class that encapsulates an external resource such as a database table (e.g. a Data Access Object). In Fig. 2, class c5 is the root of container A because it encapsulates resource X and is therefore added to the container by line 68. The container is expanded by recursively adding, up to N levels, those classes (c1 and c2) that depend on the encapsulation class (c5) by line 71 of the algorithm. Thus, N will be the maximum number of recursive dependencies from the resource encapsulating class.

One Use Case Container was created for each use case that has a design artefact, such as an object sequence diagram, indicating which classes fulfil the use case. The sequence diagram in Fig. 3 shows that classes c1, c2, c3, c4 and c5 are all used to fulfil Use Case 1. All classes referenced by the use case (c1, c2, c3, c4 and c5) are considered roots and Use Case Containers are not expanded beyond those roots. As a consequence, Use Case Container A contains all the classes referred to by the use case sequence diagram.

Table 2 indicates how many containers of each type were created per software project using the population algorithms described in this section. They are the test containers. In order to have a baseline to compare the performance of test containers, we created control containers for each container type by randomly allocating the classes of the test containers to the same number of control containers. We considered using packages as the baseline but not all of the project designs were organized in packages. The mean number of classes per control container and the mean number of containers to which each class belongs was approximately the same as the test containers. The control containers are based on the mean values from ten sets of random assignments per container type. The control containers were used to determine whether using containers based on related classes is a better indicator of change propagation than random population. If correspondence is not present between the implementation classes and the design, the test and control containers should exhibit similar performance for isolating change propagation.

3.3. Automated container analysis tools

We developed a suite of automated container analysis tools to populate and analyse risk containers from the design, to avoid human error threatening our evaluation, and to demonstrate that practitioners could easily apply our approach. The activity diagram shown in Fig. 7 illustrates how the tools analyse designs.

The automated process begins with extracting the underlying graph structure between the classes from the design. The 'Extract design graph' activity records each aggregation, composition, generalization, and dependency relationship found in the design. In parallel, the log command is used to determine all of the Git/Subversion log changes for the classes found in the design and save them into a change sets file. Next the process populates the risk containers by applying the algorithms described in Section 3.2 to the extracted design graph saved in the relationships file. The roots of abstraction and composition Design Rule Containers are automatically identified from generalizations and compositions in the relationships file. The roots of pattern Design Rules and Resource Containers are manually identified and input into the automatic analysis tools as a parameter. The roots of Use Case Containers are automatically identified as all of the classes referenced by a use case in the design. Once the container roots have been determined, the relationship data is used to automatically expand all Design Rule and Resource Containers with additional classes based on the algorithms shown in Fig. 5 (lines 51–53) and Fig. 6 (line 71). The containers created are stored in a file for each project. Once the container, change and relationship files have been populated, the tools automatically calculate the metrics (Table 3) that help determine whether the different types of containers can be used to predict and isolate change propagation using upfront design diagrams. The metrics calculated are output in an HTML report.

The industry designs were printed documents and the UML diagrams for OBTW and ps2tsa were binary image files. These caused the relationships to be manually extracted from each project. Use Case Containers were also manually populated because the designs were not machine-readable. The root classes of pattern Design Rule (e.g. class c1 in Fig. 1) and Resource Containers (e.g. classes c4 and c5 in Fig. 2) had to be manually identified because the designs did not indicate which classes represented the application of design patterns and which classes encapsulated resources. However, once the relationships had been manually extracted, the roots of abstraction and composition Design Rule Containers were automatically identified, and container expansion was performed automatically for all container types. Automatic identification of abstraction and composition Design Rules and automatic container expansion was achieved for all four projects.

The relationship expansion approach demonstrates the feasibility for fully automating the method should a machine-readable UML model be available from which the relationships could be automatically extracted. Pattern based Design Rule Containers and Resource Containers would require designers to declare patterns and resource encapsulating classes in the design (e.g., using UML stereotypes). Furthermore, the process is agnostic w.r.t. the source of design relationships used to build the input dependency graph, e.g. whether it is machine readable UML, source code (like PL/SQL packages), or UML recovered from source code. This means that although our research interest focuses on design-time analysis, risk container analysis could be applied to any stage of the software development life-cycle whether an upfront design is available or not.

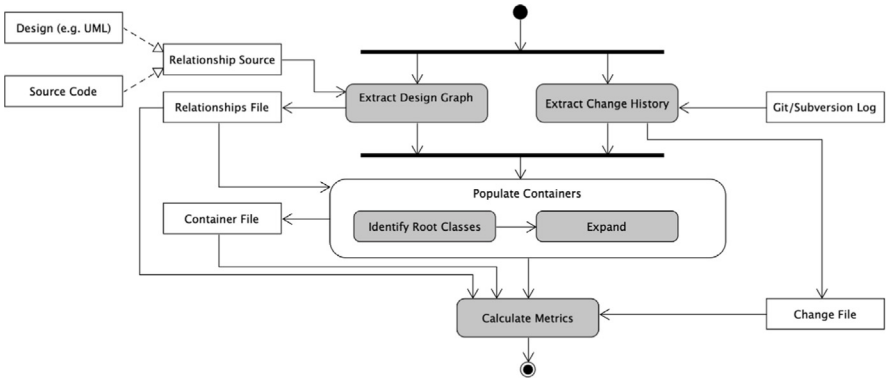


Fig. 7. Automated container analysis tools.

3.4. Class isolation metrics (container overlap)

We start by analysing the degree of overlap between the different containers of each given type, determined by the Containers Per Class and Internal Coupling metrics.

Containers Per Class is the mean number of containers to which each class is allocated. Its value for Fig. 1 is 1.5, because classes c1, c4 and c6 are in one container, and classes c2, c3 and c5 are in two containers. We also calculate the min, max, lower and upper quartile to check the distribution and ensure that the mean values are not misleading.

Internal Coupling is the mean, over all containers, of the percentage of dependencies between two classes inside the same container. A container's internal coupling is the ratio between the dependencies that are within the container and the total number of dependencies in which container's classes are involved. A dependency only counts if the dependent class is within the container. In Fig. 1, class c2 does not depend on classes c3 or c4 (it is the other way around), so the internal coupling of container A is 100%: the one internal dependency is also the total dependencies of c1 and c2. In container B three of the four dependencies are internal, and in container C two of the three dependencies are internal. So, the internal coupling for the 'system' in Fig. 1 is 81%, the mean of 100%, 75% and 67%. The min, max, lower and upper quartile are also calculated to confirm the mean.

Container types that tend to have fewer containers per class and higher internal coupling are considered to be more class isolating (less overlapping). We next consider whether the different container types are change propagation (risk) predicting, by comparing container level design metrics to container level implementation metrics.

3.5. Change propagation correlation metrics

We applied the following Clarkson et al. (2004) formula to calculate a change propagation probability between every pair of classes a and b in a system. The risk of change propagating to b, starting at a, is

$$R(b,a) = 1 - \prod (1 - \rho(b,u))$$

with ρ(b,u) the risk of change propagating to b from the penultimate subsystem u in the chain from a to b. The product is taken over all possible dependency paths from a to b. Since our approach relies on design metrics, as opposed to implementation metrics, we have to provide a substitute for ρ. We use 0.5 as a hypothetical co-change probability between different classes because the non-zero implementation co-change, over all pairs of classes in the four software projects, has a mean of 0.46. It is more appropriate than a constant of 1 because the observed data suggests that coupled classes are not guaranteed to change together. It is also more appropriate than using the actual co-change value of 0.46, which cannot be determined from the design.

Consider classes c3 and c6 in Fig. 1. There is one dependency path: a change to c6 could require a change to c5, which in turn could require a change to c3. This means the Clarkson change propagation probability would be calculated as follows:

$$1 - (1 - \rho(c3, c5) \times \rho(c5, c6)) = 1 - (1 - 0.5 \times 0.5) = 0.25$$

If a direct relationship between classes c3 and c6 were added, the change propagation probability would be:

$$1 - ((1 - \rho(c3, c5) \times \rho(c5, c6)) \times (1 - \rho(c3, c6)))$$
$$= 1 - ((1 - 0.25) \times (1 - 0.5)) = 0.875$$

The Change Propagation of the Design (CPD) for a container is the sum of the change propagation probability over all pairs of distinct classes within the container (see Table 3). The CPD calculations for the classes in Fig. 1 are shown in Table 4, as well as fictitious data for observed co-change probability. Note that ρ(a,a)=1 and that the Clarkson probability is calculated in both directions for each class pair due to relationship navigability.

The CPD value for container B is 1.5: the sum of the change propagation probabilities between the classes are as follows: class c2 and the other class members of B (c3, c4, c5) is $0+0+0=0$; class c3 and the other class members of B is 1; class c4 and the other class members of B is 0.5; and class c5 and the other class members of B is 0, for a total CPD value of $0+1+0.5+0=1.5$.

The Change Propagation of the Implementation (CPI) of a container is the sum of the actual co-change probabilities between each pair of distinct classes in the container. A log of all change sets was extracted from the project Git/Subversion repositories and parsed to identify the number of times each class changed with each other class, to compute the observed co-change probability. For example, assuming the fictitious observed probabilities of Table 4, the CPI for the same container B is 2.88.

Once the design and implementation change propagation values are calculated for each container, we use Spearman's rank correlation coefficient to test our hypothesis: change propagation probability calculated from the design should correlate to observed co-change in the implementation.

3.6. Containers in common co-change metrics

If containers isolate the risk of change propagation, we expect that a class should be more likely to be changed with a class with which it shares a container, than with one with which it does not. To test this hypothesis, we calculate Containers in Common

Table 3
Summary of metrics and functions used.

Function name	Description
M	The set of all the member classes allocated to a specific risk container.
C	The set of all of the classes in a particular software system.
R	The set of all risk containers of a given type for a particular software system.
S	A set containing all the Subversion/Git change sets.
$dep\,(c)$	Number of dependencies of class c.
$depi\,(c)$	Number of dependencies of class c where ***the dependency is also a container member.***
$clk\,(c1, c2)$	The Clarkson change propagation probability between class c1 and class c2.
$cic\,(c1)$	Set of other classes that class c1 shares containers with (containers in common).
$ncic\,(c1)$	Set of other classes that class c1 does not share containers with (no containers in common).
$oth\,(r1, c1)$	The other classes that class c1 shares container r1 with (i.e. the other container members).
$chg\,(c1)$	Number of change sets involving class c1.
$cochg\,(c1, c2)$	Number of change sets involving both class c1 and class c2.

Purpose	Metric name	Formula	Description
Class Isolation Metrics (Container Overlap)	Containers per Class (CPC)	$\frac{1}{\lvert C\rvert}\sum_{c\in C}\sum_{r\in R}\begin{Bmatrix}1 & if\ c\in r\\ 0 & if\ c\notin r\end{Bmatrix}$	Mean number of containers each class has been allocated to. This metric indicates the average amount of class sharing between containers.
	Internal Coupling (IC)	$\frac{1}{\lvert R\rvert}\sum_{M\in R}\frac{\sum_{c\in M}(depi\,(c))}{\sum_{c\in M}(dep\,(c))}$	Mean container level coupling between two classes inside the same container. This metric indicates the degree of coupling isolated within containers.
Change Propagation Correlation Metrics	Change Propagation Design (CPD)	$\sum_{c1,c2\in M s.t. c1\neq c2} clk(c1, c2)$	Sum of the Clarkson et al. change propagation probability between each pair of classes in a container.
	Change Propagation Implementation (CPI)	$\sum_{c1,c2\in M s.t. c1\neq c2}\frac{cochg\,(c1, c2)}{chg(c1)}$	Sum of co-change probability between each pair of classes in a container.
Containers in Common Co-change Metrics	Containers in Common Co-change Probability (CCCP)	$\frac{\sum_{c1\in C}\sum_{c2\in cic(c1)}\frac{cochg(c1,c2)}{chg(c1)}}{\sum_{c1\in C}\lvert cic(c1)\rvert}$	Mean co-change probability between classes that share one or more containers.
	No Containers in Common Co-change Probability (NCCCP)	$\frac{\sum_{c1\in C}\sum_{c2\in ncic(c1)}\frac{cochg(c1,c2)}{chg(c1)}}{\sum_{c1\in C}\lvert ncic(c1)\rvert}$	Mean co-change probability between classes that do not share any containers
Change Set Isolation Metrics	Isolated Change Sets (ICS)	$\frac{\lvert\{s\in S\lvert\exists r\in Rs\subseteq r\}\rvert}{\lvert S\rvert}$	Percentage of Subversion or Git change sets that are subsets of a risk container.

Table 4
Co-change probability for Fig. 1.

Change Propagation Design (CPD)							Change Propagation Implementation (CPI)						
Class	c1	c2	c3	c4	c5	c6	Class	c1	c2	c3	c4	c5	C6
c1	1.00	0.50	0.00	0.00	0.00	0.00	c1	1.00	0.56	0.11	0.11	0.11	0.11
c2	0.00	1.00	0.00	0.00	0.00	0.00	c2	0.29	1.00	0.24	0.24	0.18	0.06
c3	0.00	0.50	1.00	0.00	0.50	0.25	c3	0.09	0.36	1.00	0.27	0.18	0.09
c4	0.00	0.50	0.00	1.00	0.00	0.03	c4	0.10	0.40	0.30	1.00	0.10	0.10
c5	0.00	0.00	0.00	0.00	1.00	0.50	c5	0.11	0.33	0.22	0.11	1.00	0.22
c6	0.00	0.00	0.00	0.03	0.03	1.00	c6	0.17	0.17	0.17	0.17	0.33	1.00

Co-change Probability (CCCP) and No Containers in Common Co-change Probability (NCCCP) using the formulae shown in Table 3. We expect CCCP to be greater than NCCCP if a container type isolates the risk of change propagation.

To demonstrate how CCCP is calculated, consider the example *implementation* co-change data shown in Table 4. Firstly, we would calculate the sum of co-change probabilities for each class and the other classes with which it shares one or more containers. Using class c2 shown in Fig. 1 as an example, this would result in a value of 0.95 because class c2 shares containers with classes c1, c3, c4 and c5 and the respective co-change probabilities are: 0.29+0.24+0.24+0.18 = 0.95. This initial calculation is summed for all classes in the software studied which using the data in Table 4 would give an overall total of 4.59 because c1 = 0.56, c2 = 0.95, c3 = 0.90, c4 = 0.80, c5 = 0.88 and c6 = 0.50. The mean probability is the sum of co-change probabilities over the number of probabilities summed: 4.59/18 = 0.255.

The number of probabilities is 1+4+4+3+4+2 = 18 because c1 shares a container with one other class, c2 shares a container with four other classes, c3 shares a container with four other classes, c4 shares a container with three other classes, c5 shares a container with four other classes, and c6 shares a container with two other classes.

NCCCP is calculated in the same way as CCCP but only probabilities between classes that do not share any containers are included. The NCCCP for the example data in Table 4 and the containers shown in Fig. 1 is 0.117. Mean and upper quartile

values over each class are reported. The upper quartile is used to confirm the mean.

Our final hypothesis is that the number of change sets that are isolated in containers ought to be greater for container types that are more isolating of implementation change propagation. To test this hypothesis, we calculate Isolated Change Sets, the percentage of Git/Subversion change sets that are a subset of a risk container. Container types with greater CCCP than NCCCP ought to also have greater Isolated Change Sets than other container types where the difference between CCCP and NCCCP is less.

4. Results

In this section we describe the results of determining whether any of the container types isolate the risk of change propagation.

The most change propagation isolating container types should exhibit: (i) lower containers per class; (ii) higher percentage of internal coupling; (iii) a strong and significant correlation between internal change propagation probability calculated from the design and internal implementation co-change; (iv) greater mean co-change between classes that share containers than between classes that do not share containers; and (v) higher percentage of change sets isolated into containers.

We would not expect the control containers to exhibit these qualities because they are populated by random assignment and are not based on the coupling that is hypothesized to cause change propagation.

4.1. Class isolation results (container overlap)

Fig. 8 shows that in all but one project, the CPC metric demonstrates that Design Rule Containers (DRCs) have the least amount of class sharing. In the ps2tsa project DRCs do not have the lowest CPC (1.65), but it is only marginally greater than the RC value (1.47). Similar levels of class sharing were observed in the respective control containers. This is to be expected because control containers were populated by allocating the same classes to the same number of containers as those in the test containers.

Fig. 9 shows that levels of internal coupling (IC) are greatest for DRCs all four projects. DRCs have the second highest mean IC for OBTW, where RCs have marginally higher mean. However, the median and overall distribution is higher for OBTW DRCs than RCs. In all projects the IC values for the test containers are greater than those for the corresponding control containers. This is also to be expected because the control containers were populated by random assignment of classes, while the test containers were based on coupling. The difference between test and control containers is greatest for DRCs in all projects.

In summary, in three projects DRCs have the least amount of class sharing and come marginally in second place in the fourth project. DRCs have the greatest levels of internal coupling in all four projects. Overall, DRCs can be considered the most class isolating container type for the projects used in the experiments.

4.2. Change propagation correlation results

Table 5 lists Spearman's rank correlations between container level change propagation probability predicted from the design (CPD) and calculated from the implementation (CPI). All projects show a strong and significant correlation (α<=0.010) for all test container types.

All controls showed weaker correlations. These results suggest that CPD is an effective design time indicator of implementation change propagation (co-change) internal to the container for all container types. The UCC correlation results (for all projects) may contradict the suggestion of Hassan and Holt (2004): "results cast doubt on the effectiveness of code structures such as call graphs as good indicators of change propagation" (p. 9). That is because UCCs are populated with all of the classes found on the object sequence diagram, which represents a call graph for a given use case.

Table 5
CPD and CPI correlation.

Project	Containers	DRC		UCC		RC	
		ρ	α	P	α	P	α
API	Test	0.95	0.001	0.85	0.001	0.97	0.001
	Control	0.48	0.100	0.68	0.001	0.51	0.020
Server	Test	0.98	0.001	0.92	0.001	0.91	0.001
	Control	0.12	>0.500	0.69	0.001	0.50	0.100
Ps2tsa	Test	0.98	0.001	1.00	0.001	0.94	0.010
	Control	0.21	>0.500	0.39	0.500	0.53	0.500
OBTW	Test	0.95	0.001	0.92	0.001	1.00	0.001
	Control	0.53	0.200	0.68	0.020	0.49	0.500

Table 6
Containers in common co-change.

Project	Container type	CCCP		NCCCP		Z Test Significance
		Mean	Q75	Mean	Q75	
API	DRC	0.44	0.67	0.31	0.50	0.001
	UCC	0.37	0.50	0.33	0.50	0.010
	RC	0.34	0.50	0.32	0.50	0.001
Server	DRC	0.52	0.67	0.39	0.67	0.001
	UCC	0.32	0.44	0.42	0.67	0.100
	RC	0.39	0.50	0.42	0.67	>0.100
Ps2tsa	DRC	0.57	0.92	0.25	0.40	0.001
	UCC	0.51	0.77	0.29	0.50	0.001
	RC	0.53	0.71	0.30	0.50	0.001
OBTW	DRC	0.33	0.43	0.30	0.43	0.050
	UCC	0.38	0.57	0.30	0.43	0.001
	RC	0.31	0.43	0.32	0.44	>0.100

We computed the predicted change probability (using Clarkson's formula) and the observed co-change (using the repository's log) for each class. The correlation between both is $\rho = 0.39$ for API, $\rho = 0.42$ for Server, $\rho = 0.41$ for ps2tsa, and $\rho = -0.12$ for OBTW. We observe that the correlations are always lower than between CPD and CPI, which are the corresponding design and implementation change propagation metrics at container level. This suggests that risk containers are isolating areas of co-change due to design coupling.

4.3. Containers in common co-change results

Table 6 shows the values observed for implementation co-change probability between classes that share containers (CCCP) and classes that do not share containers (NCCCP). A z test was conducted to determine the significance of the difference between CCCP and NCCCP.

In all projects used in our evaluation, co-change is more likely between classes that share a Design Rule Container, than between classes that do not (α between 0.05 and 0.001). The same is only true for Use Case Containers in the API, ps2tsa and OBTW projects, and for Resource Containers in the API and ps2tsa projects. The probability of a class changing with a class with which it does not share a container is about the same for all container types in all projects.

In the control containers, the CCCP and NCCCP are approximately the same for all container types for all projects. The values observed are similar to those observed for classes that do not share containers in the test containers (NCCCP). This consistency between the controls and test containers is to be expected, due to the absence of architectural connections to carry the ripple effect (Lindvall et al., 2003; Bass et al., 2012).

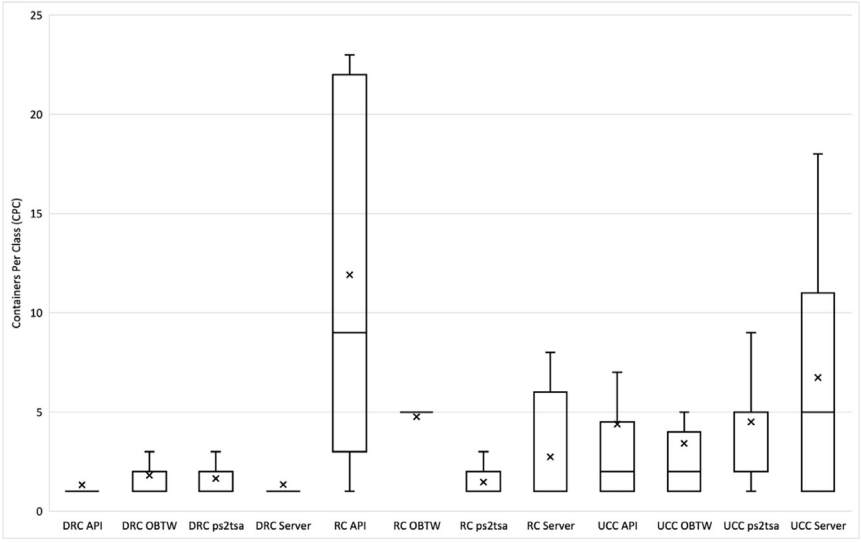


Fig. 8. Containers per class (x = mean marker).

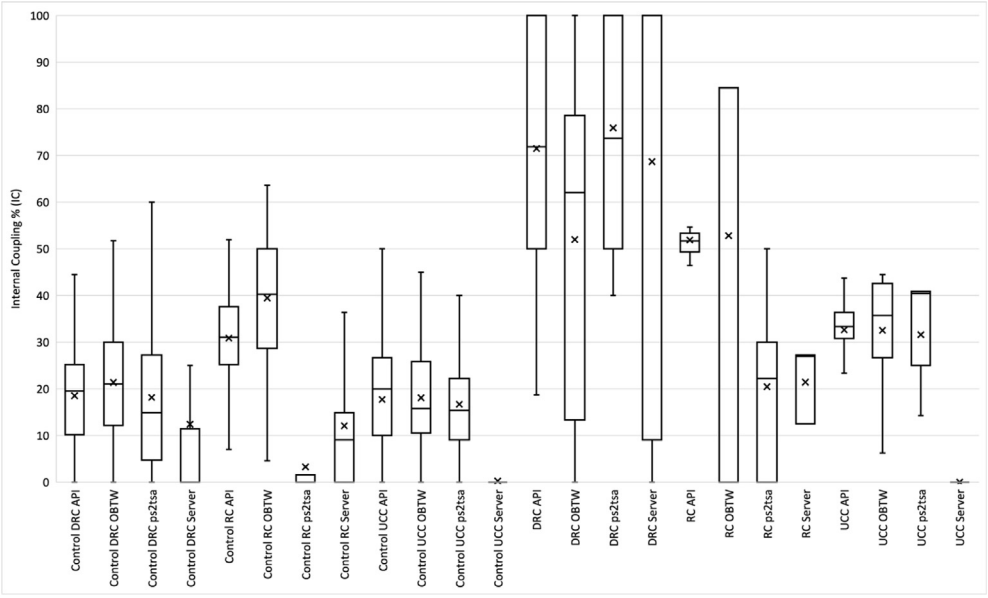


Fig. 9. Internal coupling percentage (x = mean marker).

The CCCP and NCCCP results provide further evidence that DRCs populated from design class diagrams isolate change propagation during the implementation.

4.4. Change sets

Fig. 10 shows the number of isolated change sets for the test Design Rule Containers are greater than those for the other container types in the OBTW project, but approximately equal highest in the other three projects. In all projects the change sets isolated into test Design Rule Containers are greater than their corresponding controls, albeit in the case of the Server project the difference is very small (API = − 30%, Server = − 1%, ps2tsa = − 68% and OBTW = − 15%).

The low isolated change sets values for Use Case Containers, allied to the observation that in one project NCCCP was greater than CCCP for Use Case Containers, casts doubt over whether they are really isolating change propagation. A z-test confirmed that the Design Rule and Resource Container ICS values for the API and OBTW projects are significantly greater than for Use Case Containers ($\alpha = 0.001$).

4.5. Developer feedback

Having identified Design Rule Containers as the most promising container type, we interviewed the API developer. Firstly, to determine if the developer recognized and could reason about the containers, i.e. whether they were meaningful entities to them. Secondly, to corroborate that the containers predicted to contain the most change propagation did contain coupling problems that would account for the high level of change propagation. The interview used a structured questionnaire. Prior to questioning the developer, Fig. 1 was explained to them.

The developer was allowed to refer to the un-ranked container summary shown in Table 7, as well as a full list of classes for each container.

Initially the developer was asked to name the Design Rule Containers most likely to contain classes that will have to be frequently changed together to add functionality or fix bugs (picking

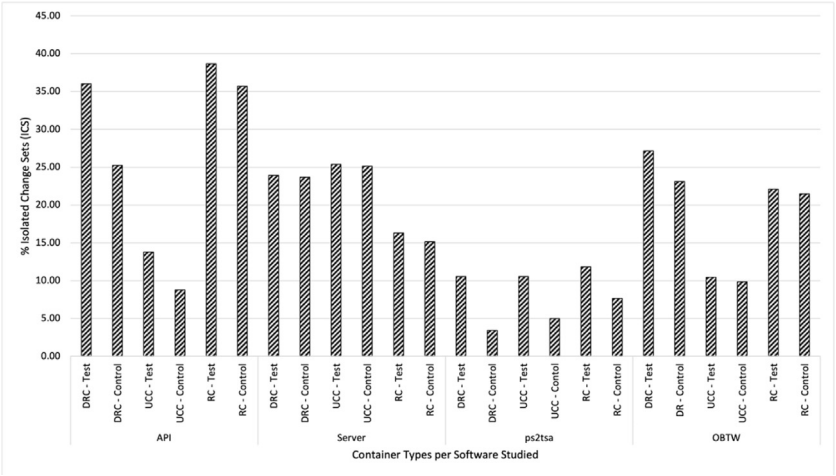


Fig. 10. Number of change sets.

Table 7
Unranked container list (1st row only).

Container	Basis	Root class(es)	Other members
A_1	Abstraction	NamedValue	All sub-classes

Table 8
Riskiest design rule containers.

DRC	Metric rank	Developer rank
A_4	1	1
A_2	2	3
A_10	3	5
A_1	4	6
A_8	5	4

their top five). Only once they had nominated their top five, were the top five containers ranked by the CPD metric revealed to the developer. The developer was then asked to explain why they thought the classes ranked in the top five by the CPD metric would have to be frequently changed together. Table 8 shows that of the top five containers predicted to isolate the most change propagation by the CPD metric, the developer agreed on four of them (albeit with a different ranking).

Both the developer and CPD ranked A_4, which is illustrated in Fig. 4, as the highest container. The developer had implicated A_4 as having excessive change propagation by stating: "there seem to be too many subclasses to maintain and they are very complicated". They further explained that "complex concrete sub-classes emerged from the diverse use cases the lists had to support. This caused conflicts between abstract class code and concrete sub-class code". This response indicates the co-change in A_4 stems from the generalization relationships and, hence, is rooted in coupling between the container's member classes.

Their explanation for container A_2 to be highly ranked was: "lots of common code was duplicated in the different service classes. This was a deliberate development strategy to allow different developers to work independently". The developer went on to explain that the duplicated code fulfilled its responsibility using other utility classes. This meant that all service classes were coupled to the utility classes, thus the container exhibits a high level of internal coupling. With the benefit of hindsight, the developer believes the A_2 could be simplified.

Containers A_1 and A_10 have classes that represent table structures in the relational database the API can be used to access. Despite containers A_1 and A_10 each having classes related to a different set of tables, the developer provided the same justification for both containers. They explained that: "when I had to change the base class, I often also had to change the sub-classes. This was because the class structures directly reflect the underlying table structures". Furthermore, they described three recurring change scenarios where a ripple effect between the container contents occurred: (i) "when a new common column was added to all tables, the abstract class and all sub-classes had to be changed"; (ii) "when new tables were introduced, or relationships refactored, all referencing sub-classes had to be changed"; (iii) "if there was a bug in low level table access routines it often propagated across the sub-classes by copy/paste. Hence, when found, all sub-classes had to be changed". Design Rule Containers were more effective at isolating the risk associated with these change scenarios than Resource Containers, despite Resource Containers being based on tables. That is because these scenarios impacted many Resource Containers, as each one is based on a different table, whereas the risk was isolated into a single Design Rule Container.

The developer explained that in A_8 "lots of common code was duplicated in sub-classes. To add new abstract functionality, all the concrete classes had to be changed".

Mo et al. (2015) identified several common causes of error-proneness in the projects they analysed with DRSpaces, namely: unstable interfaces, implicit cross module dependencies, unhealthy inheritance, and cyclic dependencies. The API developer's responses suggest that Design Rule Containers A_4, A_10, A_1 and A_8 contain examples of unhealthy inheritance and other coupling relationships driving their relatively high levels of change propagation. The Change Propagation Design (CPD) metric captures such relationships and hence the agreement with the developer's assessment.

The exercise was repeated with the student who developed the ps2tsa project. Their top five Design Rule Containers were ranked in positions 2, 3, 4, 7 and 10 out of 11 by the CPD metric. Thus, agreement between the ps2tsa developer's top five and CPD was less than that of the API developer (60% as opposed to 80%). When the risk containers ranked in the top five places by the CPD metric were revealed, the ps2tsa developer was able to justify why the predicted co-change is based on design dependencies. When asked to justify why they nominated a container ranked in 10th place, the developer explained they had nominated it because that container is based on a composition design rule. They had suspected composition dependencies may have been stronger than abstraction and pattern dependencies, a suspicion that is not supported by the co-change data in this case.

The developer's ability to recognize Design Rule Container contents and reason about the causes of coupling problems associated with them, suggests they are based on modularizing design rules as per the Design Rule Hierarchy algorithm, i.e. the developer was able to comprehend them as decoupled 'technical' modules.

5. Validity and discussion

We focused on Messick (1987) consequential threats because the manual aspects of our method could be subject to misinterpretation, and generalizability threats because they undermine applicability to other software projects.

5.1. Consequential

Design Rule Containers aim to extend the work of Xiao et al. (2014) and Wong et al. (2009) and we could have misinterpreted the DRH algorithm or the DRSpaces based upon it. The consistency of our change propagation results with Wong et al.'s underlying hypothesis suggests that we have interpreted Design Rule Hierarchy correctly.

Manual identification of pattern design rules is prone to ad-hoc human error. The consequences of missing a pattern is that the design will not be as fully decomposed into risk containers as it could be. A similar threat exists for Resource Containers through failure to identify an external resource encapsulation class. Whilst this threatens the optimum use of the technique, change propagation would still be contained within a larger container, in the case where a container could be further decomposed.

Manual extraction of design relationships is another threat that potentially affects all three container types tested. This threat was mitigated by manually transcribing the relationships documented in the designs on three separate occasions and comparing the relationship files produced. Any discrepancies between the three versions were investigated and transcription errors were corrected prior to analysis. This threat is further mitigated by the consistency of the results between the different software projects and the ability of API and ps2tsa developers to recognize the Design Rule Containers produced as independent 'modules'.

As Use Case Containers were manually populated, classes could have been omitted or added to particular container by mistake. We argue that owing to their simplicity, i.e. they are populated with all classes referred to by an object sequence diagram, the risk level is minimal and acceptable. Design Rule and Resource Containers were expanded automatically using the relationships file and are, therefore, not subject to this threat.

We have comprehensively unit tested the programs used to analyse the projects, with the Java JUnit testing framework. Whilst some bugs will undoubtedly still exist, we have taken reasonable precautions to find and fix bugs prior to analysing the results.

5.2. Generalizability

In order to apply Clarkson et al.'s (2004) method of calculating change propagation probability, we used a constant value of 0.5 in place of observed data. Whilst the correlation results in Table 5 suggest this was a reasonable value to use, we do not know whether it could be further optimized.

We assumed that a PL/SQL package in the Server project is analogous to a Java class in the other projects. This is a reasonable assumption to make because PL/SQL packages and Java classes are both nodes in the underlying structural graph of the software. The consistency of the results over all four projects further supports that view.

Not all software development projects will have the upfront designs our method requires. The emergence of agile methods may magnify this threat. However, Nord et al. (2014) explain that architecture is needed in large scale agile projects to avoid excessive redesign later. According to Bass et al. (2012), Boehm and Turner analysed 161 projects and found that the bigger the project, the more architecture risk assessment is needed to avoid rework. Our method is therefore more likely to be applicable to larger scale software development where the architecture risk assessment has produced upfront designs. Nord et al.'s "zipper model" that enables architecture and feature stories to proceed in parallel may provide a means to introduce upfront risk analysis using risk containers into agile projects without impeding agility.

In terms of the UML class diagrams needed to construct Design Rule Containers, Petre (2013) found that seven of the eleven UML users did use upfront class diagrams. This suggests that Design Rule Containers are likely to be feasible for projects using UML. For projects not using UML, our analysis of the Server project shows that any dependency graph can be used. However, the threats of UML and design availability in agile projects are mitigated by having an analytical process that is agnostic of how the class dependency graph is obtained, as explained in Section 3.3.

According to Zhang and Tan's study (2007), large Java systems have 314 to 12299 classes. Only the API project is in that range, but still well below the median (877). To counter this threat, we developed a Java parser to extract "design-like" relationships from source code that include: (i) dependencies on extended or implemented types; (ii) dependencies on non-static member variable types; (iii) public method parameter and return types upon which classes depend. These extraction rules provide a similar level of detail to the projects' design diagrams.

The parser extracted 2429 "design-like" relationships from ArgoUML which is a large Java project according to Zhang and Tan. We then automatically populated 73 abstraction Design Rule Containers based on the extracted relationships. The metrics for ArgoUML are: CPC = 1.49, IC = 74.85, ICS = 29.73, CCCP = 0.2 is greater than NCCCP = 0.07, and Spearman's rank correlation between CPD and CPI is strong ($\rho = 0.93$) and significant ($\alpha = 0.001$). These results are consistent with those for the other projects.

Whilst this is not a completely comparable test, because pattern and composition Design Rule Containers were not created, it provides early confidence that the method is fully automatable and scalable. It also suggests the approach has potential to work for any source of relationships (design or source code). Furthermore, automation based on any source of relationships would enable the approach to accommodate design drifting and decay (as observed in the API and Server projects) by applying risk container analysis iteratively throughout all stages of the software development life-cycle. It also suggests the process could be applied to projects where an upfront design is not available, as is often the case in agile projects.

6. Conclusion

Our aim is to help practitioners identify and mitigate maintainability risks early in the software development life-cycle. This paper provides several new contributions: (i) an evaluation of the ability of design time risk containers for predicting and isolating implementation change propagation; (ii) identification of the most change propagation isolating container type; and (iii) formally defined container change propagation metrics. This paper also extends our previous work by providing: (iv) analysis of additional projects; (v) a more detailed description of the

container population algorithms; and (vi) a formal definition of the class isolation metrics.

The three container types tested were: (1) Design Rule Containers (DRCs) that group classes by modularizing design rules; (2) Use Case Containers that group classes supporting use cases; and (3) Resource Containers that group classes dependent upon external resources such as database tables, files and services. Results from two industrial projects (API and Server) and two student dissertation projects (ps2tsa and OBTW) suggest that DRCs are more effective than the other container types because DRCs have: (i) the least amount of class sharing (container overlap), (ii) the highest levels of internal coupling, (iii) the highest probability of classes changing with other classes with which they share DRCs; (iv) the most change sets isolated in containers, and (v) a strong and significant correlation between design change propagation probability and implementation co-change. Crucially, the correlations between design change propagation and implementation co-change were much weaker for control containers based on random assignment of classes to containers than for the three test container types (Design Rule, Use Case and Resource). This suggests the observations in support of our conclusion (i-v) are due to design dependencies. In one project, the qualitative analysis of its lead developer agreed with the metrics in determining four of the five DRCs that isolate the most change propagation. In a second project the developer agreed with three out of five DRCs predicted to isolate the most change propagation. In both cases the developers were able to identify and justify why design dependencies had resulted in the top five DRCs being predicted to isolate the most change propagation.

These results have three potential applications to practitioners. Firstly, developers assigned to work on different DRCs are less likely to make changes to shared classes than they would be if work were allocated based on the other container types. This supports Wong et al.'s (2009) original goal for the Design Rule Hierarchy algorithm upon which DRSpaces (Xiao et al., 2014) and DRCs (Leigh et al., 2016, 2017) are based. Secondly, the ability to rank containers according to relative levels of predicted internal change propagation, would allow practitioners to prioritize containers for redesign to reduce the cost of making changes during implementation. Thirdly, practitioners can factor container size into redesign estimates to help judge whether the likely risk impact warrants such mitigation, as well as increasing the likelihood of the redesign being implemented within budget.

When considered in combination with our previous error-proneness research (Leigh et al., 2016, 2017), which showed container-level coupling to be a significant indicator of implementation error-proneness for DRCs in the API and Server projects, the new change propagation results presented in this paper suggest DRCs are the most useful container type for these maintainability risks overall. That suggests class diagrams are more helpful than use case sequence diagrams when assessing upfront designs for potential implementation maintainability risks.

Opportunities for future work include analysing if container-based risk assessment is generalizable for other risk categories such as security risks. We also plan to investigate if container-based risk assessment can be effective in agile software projects, using minimal design, to balance the benefits of discipline and agility.

CRediT authorship contribution statement

Andrew Leigh: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Visualization, Project administration. **Michel Wermelinger:** Conceptualization, Methodology, Validation, Formal analysis, Writing - review & editing, Supervision. **Andrea Zisman:** Conceptualization, Methodology, Validation, Formal analysis, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abdelmoez, W., Shereshevsky, M., Gunnalan, R., Ammar, H.H., Bo, Y., Bogazzi, S., Korkmaz, M., Mili, A., 2005. Quantifying software architectures: an analysis of change propagation probabilities. In: Proc. of 3rd Int'L Conf. on Computer Systems and Applications. IEEE, pp. 124–132. http://dx.doi.org/10.1109/AICCSA.2005.1387113.

Akingbehin, K., 2005. A quantitative supplement to the definition of software quality. In: Third ACIS Int'L Conf. on Software Eng. Research. IEEE, pp. 348–352. http://dx.doi.org/10.1109/SERA.2005.15.

Bakota, T., Hegedus, P., Ladanyi, G., Kortvelyesi, P., Ferenc, R., Gyimothy, T., 2012. A cost model based on software maintainability. In: Proc. 28th IEEE Int'l Conf. on Software Maintenance, pp. 316–325. doi:10.1109/ICSM.2012.6405288.

Baldwin, C.Y., Clark, K.B., 2000. Design Rules, 1: The Power of Modularity. MIT Press.

Bass, L., Kazman, R., Clements, P., 2012. Software Architecture in Practice, third ed. Addison Wesley, p. 122, 2013.

Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H., 2004. Architecture-level modifiability analysis. J. Syst. Softw. 69, http://dx.doi.org/10.1016/S0164-1212(03)00080-3.

Bouwers, E., Visser, J., Lilienthal, C., van Deursen, A., 2010. A cognitive model for software architecture complexity. In: Proc. 18th Int'L Conf. on Program Comprehension. IEEE, pp. 152–155. http://dx.doi.org/10.1109/ICPC.2010.28.

Brooks, J.P., 1995. The Mythical Man Month: Essays on Software Engineering. Addison Wesley Professional.

Charette, R., 2005. Why software fails [software failure]. IEEE Spectr. 42 (9), 42–49. http://dx.doi.org/10.1109/MSPEC.2005.1502528.

Chidamber, S.R., Kemerer, C.F., 1994. A Metrics Suite for Object Oriented Design. In: IEEE transactions on software engineering, 20, (6), Institute of Electrical and Electronics Engineers (IEEE), New York, pp. 476–493. http://dx.doi.org/10.1109/32.295895.

Clarkson, P.J., Simons, C., Eckert, C., 2004. Predicting change propagation in complex design. J. Mech. Des. 126 (5), 788–798. http://dx.doi.org/10.1115/1.1765117.

Curtis, B., Sappidi, J., Szynkarski, A., 2012. Estimating the size, cost, and types of technical debt. In: Proc. 3rd Int'L Workshop Managing Technical Debt. IEEE Press, pp. 49–53. http://dx.doi.org/10.1109/MTD.2012.6226000.

Hassan, A.E., Holt, R.C., 2004. Predicting change propagation in software systems. In: Proc. Int'L Conf. on Software Maintenance. IEEE, pp. 284–293. http://dx.doi.org/10.1109/ICSM.2004.1357812.

Hebig, R., Ho-Quang, T., Robles, G., Fernandez, M.A., Chaudron, M.R.V., 2016. The quest for open source projects that use UML: Mining GitHub. In: Proc. 19th Int'L Conf. on Model Driven Eng. Languages and Systems. ACM, pp. 173–183. http://dx.doi.org/10.1145/2976767.2976778.

Kazman, R., Yuanfang, C., Ran, M., Qiong, F., Xiao, L., Haziyev, S., Fedak, V., Shapochka, A., 2015. A case study in locating the architectural roots of technical debt. In: Proc. Int'L Conf. on Software Eng.. IEEE, pp. 179–188. http://dx.doi.org/10.1109/ICSE.2015.146.

Lehnert, S., 2011. A taxonomy for software change impact analysis. In: Proc. 12th Int'L Workshop on Principles of Software Evolution and 7th Annual ERCIM Workshop on Software Evolution. ACM, pp. 41–50. http://dx.doi.org/10.1145/2024445.2024454.

Leigh, A., Wermelinger, M., Zisman, A., 2016. An evaluation of design rule spaces as risk containers. In: Proc. 13th Working Int'L Conf. on Software Architecture. IEEE, pp. 295–298. http://dx.doi.org/10.1109/WICSA.2016.34.

Leigh, A., Wermelinger, M., Zisman, A., 2017. Software Architecture Risk Containers In: Proc. 11th European Conf. on Software Architecture, pp. 171-179, 2017.

Leigh, A., Wermelinger, M., Zisman, A., 2019. Risk containers - a help or hindrance to practitioners?. In: Proc. of Int'L Conf. on Software Architecture Workshops (ICSAW). IEEE, pp. 230–233. http://dx.doi.org/10.1109/ICSA-C.2019.00047.

Lindvall, M., Tvedt, R.T., Costa, P., 2003. An empirically-based process for software architecture evaluation. Empirical Software Engineering. Boston: Kluwer Academic Publishers 8 (1), 83–108. http://dx.doi.org/10.1023/A:1021772917036.

Löwer, et al., 2014. OpenBundestagswahl, GitHub. [online]. Available: https://github.com/smolvo/OpenBundestagswahl (Accessed: 9th Dec-2018).

Martin, R.C., 1997. Large-scale stability. C++ Report 9 (2), 54–60.

Messick, S., 1987. Validity. In: ETS Research Report Series, (2), http://dx.doi.org/10.1002/j.2330-8516.1987.tb00244.x.

Mo, R., Cai, Y., Kazman, R., Xiao, L., 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: Proc. 12th Working IEEE/IFIP Conf. on Software Architecture. IEEE, pp. 51–60. http://dx.doi.org/10.1109/WICSA.2015.12.

Mo, R., Cai, Y., Kazman, R., Xiao, L., Qiong, F., 2016. Decoupling level: a new metric for architectural maintenance complexity. In: Proc. 38th Int'L Conf. on Software Eng.. ACM, http://dx.doi.org/10.1145/2884781.2884825.

Nord, R.L., Ozkaya, I., Kruchten, P., 2014. Agile in distress: Architecture to the rescue. In: Proc. 15th Int'l Conf. on Agile Software Development, pp. 43-57.

Nugroho, A., Visser, J., Kuipers, T., 2011. An empirical model of technical debt and interest. In: Proc. 2nd Workshop on Managing Technical Debt. ACM, pp. 1–8. http://dx.doi.org/10.1145/1985362.1985364.

Petre, M., 2013. UML in practice. In: Proc. Int'l Conf. on Software Eng., pp. 722-731, doi:10.1109/ICSE.2013.6606618.

Planetside 2, 2018. Daybreak game company LLC. [online]. Available: https://www.planetside2.com (Accessed: 9th Dec-2018).

Rostami, K., Stammel, J., Heinrich, R., Reussner, R., 2015. Architecture-based assessment and planning of change requests. In: Proc. 11th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures, pp. 21-30, doi:10.1145/2737182.2737198.

Shaik, I., Abdelmoez, W., Gunnalan, R., Mili, A., Fuhrman, C., Shereshevsky, M., Zeid, A., Ammar, H.H., 2006. Using change propagation probabilities to assess quality attributes of software architectures. In: Proc. Int'L Conf. on Computer Systems and Applications. IEEE, pp. 704–711. http://dx.doi.org/10.1109/AICCSA.2006.205167.

Spearman, C., 1904. The proof and measurement of association between two things. Am. J. Psychol. 15 (1), 72–101.

Wermelinger, M., Yu, Y., Lozano, A., Capiluppi, A., 2011. Assessing architectural evolution: a case study. Empir. Softw. Eng. 16 (5), 623–666. http://dx.doi.org/10.1007/s10664-011-9164-x.

Wolfe, J.M., Horowitz, T.S., 2017. Five factors that guide attention in visual search. Nat. Hum. Behav. 1 (3), 1–8.

Wong, S., Cai, Y., Valetto, G., Simeonov, G., Sethi, K., 2009. Design rule hierarchies and parallelism in software development tasks. In: Proc. 24th Int'L Conf. on Automated Software Eng. IEEE, pp. 197–208. http://dx.doi.org/10.1109/ASE.2009.53.

Xiao, L., Cai, Y., Kazman, R., 2014. Design rule spaces: a new form of architecture insight. In: Proc. Int'L Conf. on Software Eng. ACM, pp. 967–977. http://dx.doi.org/10.1145/2568225.2568241.

Zar, J.H., 1984. Biostatistical Analysis, second ed. Prentice-Hall, 1984, Table B.19.

Zhang, H., Tan, H.B.K., 2007. An empirical study of class sizes for large java systems. In: Proc. 14th Asia-Pacific Software Engineering Conference. IEEE, pp. 230–237. http://dx.doi.org/10.1109/ASPEC.2007.64.

Andrew Leigh received the M.Sc. degree in software development from The Open University and is working on his Ph.D. thesis. His research interest is architecture analysis to predict and contain maintainability risks.

Michel Wermelinger received a Ph.D. in computer science from the New University of Lisbon. He is a senior lecturer in computing at the Open University. His research interests are software architecture and software maintenance.

Andrea Zisman is a professor in Computing at The Open University. She has been research-active in the areas of software and service engineering, where she has published extensively.