Isolating and Predicting Risks in Architectural Design

Thesis

How to cite:

For guidance on citations see FAQs.

© 2021 Andrew Philip Leigh

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Version of Record

Link(s) to article on publisher’s website:
http://dx.doi.org/doi:10.21954/ou.ro.000142c8

oro.open.ac.uk
Isolating and Predicting Risks in Architectural Design

PhD Thesis

Author: Andrew Philip Leigh
Supervisors: Dr. Michel Wermelinger, Prof. Andrea Zisman

June 2021
(Revised February 2022)
Abstract

Failure to manage risks often causes budget and schedule problems in software projects. Software architectures are difficult to change at later stages and determine software quality attributes such as maintainability. Error-proneness and change propagation risks occur in software that is difficult to maintain. If risks can be isolated into architecture subsets called risk containers, practitioners could prioritise mitigations towards the riskiest containers to increase the chances of project success. Some existing software architecture analysis techniques attribute maintainability risks to architecture subsets, but previous research fails to consider which kinds of architecture subsets are risk isolating (risk inducing architecture elements are not shared with other containers). Furthermore, the architecture description standard does not include guidance for describing risks. This thesis addresses those knowledge gaps. Three types of risk container are extracted from the architectural designs of four software projects and metrics are used to determine which container type is the most risk isolating. Design Rule Containers are the most effective container type because they have very strong and significant correlations between design metrics and implementation change propagation, moderate or stronger correlations between design metrics and implementation error-proneness, but crucially they overlap less than Resource and Use Case Containers which means they are more risk isolating. An experiment demonstrates that participants were able to locate more error-inducing architecture flaws using smaller container diagrams, than when using a larger overall diagram. An architecture risk model based on the international standards for architecture description and risk management is synthesised from the output architecture analysis techniques, including risk containers, so that the isolated risks can be communicated. Surveyed practitioners consider the model to be more applicable to waterfall software development than agile and prefer the model to textual risk descriptions due to the fidelity, rigour, and traceability supported by the model.
Acknowledgements

Undertaking a PhD whilst in full time employment is a significant challenge. A challenge that one cannot meet without the help of others. I would like to express my gratitude towards the following individuals who have supported my academic adventure.

Dr. Michel Wermelinger and Prof. Andrea Zisman for their expert coaching and mentoring, and for showing me that an engineering mindset and a research mindset are not the same.

Chris Stanley, Martyn Cole, Phil Gray, and Ashley Lane for helping to provide the resources needed to complete this research project.

Andy Kinsey for championing my application as well as providing the confidence and encouragement I needed to begin.

Martyn Abbott, Matthew Beaumont, Michael Robson, and Tony Powers for their feedback and help trialling experiments.

Simon Godden for teaching me how you put code together matters, many years ago.

Karl Laycock for providing inspiration when my mathematical ability was lacking.

My wife Susan Leigh for patiently allowing me to pursue my passion for software engineering, and for providing the moral support I needed when it all became a bit too much.

Permission to reproduce extracts from British Standards is granted by BSI Standards Limited (BSI). No other use of this material is permitted.
# Table of Contents

Abstract .................................................................................................................. 2  
Acknowledgements ................................................................................................. 3  
Table of Contents .................................................................................................... 4  
List of Figures .......................................................................................................... 9  
List of Tables ........................................................................................................... 11  
Chapter 1 - Introduction ......................................................................................... 13  
  1.1 Motivation ..................................................................................................... 13  
  1.2 Research Issues ............................................................................................ 17  
  1.3 Research Glossary ......................................................................................... 17  
  1.4 Research Problem .......................................................................................... 20  
  1.5 Overall Proposal ............................................................................................ 21  
  1.6 Research Scope .............................................................................................. 23  
  1.7 Research Questions ....................................................................................... 25  
  1.8 Hypotheses .................................................................................................... 25  
  1.9 Method .......................................................................................................... 27  
  1.10 Contributions ............................................................................................... 27  
  1.11 Published Work ............................................................................................ 28  
  1.12 Dissertation Structure .................................................................................. 29  
Chapter 2 - Literature Review .............................................................................. 31  
  2.1 Identification of Maintainability Risks in Architectures ................................. 31  
    2.1.1 Search Criteria .......................................................................................... 31  
    2.1.2 Search Method .......................................................................................... 32  
    2.1.3 Generic Risk Identification ...................................................................... 35  
    2.1.4 Maintainability Risk Identification ............................................................ 39  
    2.1.5 Error-proneness Risk Identification ........................................................... 44  
    2.1.6 Change Propagation Risk Identification .................................................... 46  
    2.1.7 Section Summary ...................................................................................... 49  
  2.2 Software Metrics ............................................................................................ 50  
    2.2.1 Error-proneness Indicators ...................................................................... 50  
    2.2.2 Change propagation Indicators ................................................................. 55  
  2.3 Modelling Risks ............................................................................................. 57  
  2.4 The Research Gap .......................................................................................... 60  
  2.5 Chapter Summary ........................................................................................... 64
Appendix E - Risk Model Questionnaire.................................................................224
Appendix F - Risk Model Participant Information Sheet .....................................232
Appendix G - Risk Model Survey Consent Form..................................................234
Appendix H - Risk Model Survey Permission Letter ..........................................235
Appendix I - Risk Model Survey Participant Comment Coding............................236
List of Figures

Figure 1-1 Container Based Software Architecture Risk Assessment Process ......................... 22
Figure 2-1 Security Concepts Model (adapted from Fabian et al., 2010, Fig. 2) .................... 59
Figure 2-2 Risk Model (abridged from Mayer et al., 2019, Fig. 6) .................................... 60
Figure 4-1 Design Rule Container Population Algorithm .................................................. 82
Figure 4-2 Design Rule Container Example ....................................................................... 83
Figure 4-3 Resource Container Population Algorithm ....................................................... 85
Figure 4-4 Resource Container Example ............................................................................ 85
Figure 4-5 Use Case Container Example ......................................................................... 86
Figure 4-6 Automated Container Analysis Tools Activity Diagram ..................................... 87
Figure 4-7 Design Rule Container Example ..................................................................... 93
Figure 4-8 Containers per Class (x=mean marker) ............................................................. 94
Figure 4-9 Internal Coupling Percentage (x=mean marker) ............................................ 95
Figure 5-1 Design Rule Container Example ..................................................................... 101
Figure 5-2 Design Structure Matrix of an Abstract Class and Sub-classes ......................... 103
Figure 5-3 Directed Graphs for Design Rule Container Examples ....................................... 104
Figure 5-4 Transforming Containers into Strongly Connected Graphs ......................... 105
Figure 5-5 Distribution of Bug Space Percentage (x=mean marker) ............................ 110
Figure 5-6 Distribution of Container Coupling Between Objects (x=mean marker) ........ 111
Figure 5-7 Distribution of Container Fan In (x=mean marker) .......................................... 115
Figure 5-8 Distribution of Container Cluster Factor (x=mean marker) ............................ 118
Figure 5-9 Distribution of Container McCabe Cyclomatic Complexity (x=mean marker) ... 120
Figure 6-1 Design Rule Container Example ..................................................................... 127
Figure 6-2 Percentage of Isolated Change Sets ................................................................. 135
Figure 6-3 Percentage of Neatly Isolated Change Sets ...................................................... 136
Figure 7-1 Usefulness Experiment Instructions ................................................................. 147
Figure 7-2 Participant Experience (x=mean marker) ......................................................... 150
Figure 7-3 Time Taken per Question (x=mean marker) .................................................. 151
Figure 7-4 Percentage of Cyclic Dependencies Correctly Identified (x=mean marker) ..... 151
Figure 7-5 % of Change Impacts Correctly Identified (x=mean marker) ....................... 153
Figure 8-1 Method used to synthesise an Architecture Risk Model .................................. 158
Figure 8-2 Grouping Risks and Relating them to Concerns ............................................. 161
Figure 8-3 Relating Risks to ‘risky’ AD Elements ............................................................. 162
Figure 8-4 Specifying Risk Indicators ............................................................................. 163
Figure 8-5 Support for Risk Containers ............................................................................. 163
Figure 8-6 Relating Impacts and Mitigations to Risks ......................................................... 165
Figure 8-7 An Architecture Risk Model ................................................................................ 166
Figure 8-8 Support for Iterative Risk Management .............................................................. 168
Figure 8-9 Participant Experience Levels (x=mean marker; Questions 1-10) ....................... 175
Figure 8-10 Applicability to Agile, Scaled Agile, and Waterfall (Questions 11-17) ............. 176
Figure 8-11 Practitioner Approach Preference (Question 19) .............................................. 177
Figure 8-12 Model Completeness (Questions 20 and 21) .................................................... 178
Figure 8-13 Revised Architecture Risk Model ...................................................................... 184
Figure B-9-1 Usefulness Experiment Control Class Diagram .............................................. 216
Figure B-9-2 Usefulness Experiment Example Design Rule Container Class Diagram ........ 217
Figure B-9-3 Usefulness Experiment Example Resource Container Class Diagram .......... 217
Figure B-9-4 Usefulness Experiment Example Use Case Container Sequence Diagram ...... 218
List of Tables

Table 1-1 Research Contributions........................................................................................................ 28
Table 2-1 Results of Literature Search.................................................................................................. 33
Table 2-2 Comparison of Proposed Research to Nearest Matches ................................................... 62
Table 3-1 Matrix of research approaches and methods ........................................................................ 67
Table 3-2 Spearman’s Rank Classifications .......................................................................................... 69
Table 3-3 Software Projects Studied.................................................................................................... 75
Table 3-4 The Element Data File Format ............................................................................................... 76
Table 3-5 The Relationship Data File Format ......................................................................................... 77
Table 3-6 The Co-Change Data File Format .......................................................................................... 79
Table 3-7 Container Data File Format ................................................................................................ 79
Table 4-1 Method Automation .............................................................................................................. 89
Table 4-2 Number and Coverage of Containers ................................................................................... 91
Table 5-1 Container level Cyclomatic Complexity Calculations for Figure 4-2................................. 105
Table 5-2 Number and Coverage of Containers ................................................................................ 109
Table 5-3 Percentage of Containers with Excess Bug Space Occupancy ........................................ 110
Table 5-4 CBO and Error-proneness Correlation ................................................................................. 112
Table 5-5 CBOC and TBFC Baseline Metrics ....................................................................................... 113
Table 5-6 CDD and Error-proneness Correlation ................................................................................. 114
Table 5-7 Fl and Error-proneness Correlation ...................................................................................... 115
Table 5-8 FIC and TBFC Baseline Metrics ........................................................................................... 116
Table 5-9 FIDD and Error-proneness Correlation ............................................................................... 117
Table 5-10 LCF and Error-proneness Correlation ................................................................................. 119
Table 5-11 MCC and Error-proneness Correlation ............................................................................. 120
Table 6-1 Co-change Probability for Figure 6-1 .................................................................................. 128
Table 6-2 CPD and CPI Correlation ................................................................................................... 132
Table 6-3 Class level CPDC and CPIC Baseline Metrics ................................................................. 133
Table 6-4 Containers in Common Co-Change ....................................................................................... 134
Table 6-5 Example of un-ranked Container Summary used in Developer Interviews ................. 138
Table 6-6 API Design Rule Containers ranked by Risk ............................................................... 138
Table 7-1 Themes Identified from Qualitative Responses ................................................................. 153
Table 8-1 ISO 42010 and 31000 Concept Definition ......................................................................... 160
Table 8-2 Definition of new Items proposed for an Architecture Risk Model ................................. 166
Table 8-3 Comparison between the proposed and existing risk models ..................................... 170
Table 8-4 Results of Thematic Analysis (Questions 11-22) .................................................. 179
Chapter 1 - Introduction

This chapter introduces the work described in this dissertation by presenting the research motivation, research issues, research glossary, the overall proposition, research scope, research questions, hypotheses, and the method used to evaluate whether risks can be isolated within software architectures. The chapter finishes by explaining the research contributions, relationship to previously published work, and dissertation structure.

1.1 Motivation

Many software projects fail when measured in terms of budget and schedule, and neglecting to manage risks is a common cause of failure (Charette, 2005). Software architectural design influences risk because design decisions constrain the achievement of software quality attributes such as efficiency, functionality, maintainability, portability, reliability, and usability (Bengtsson et al., 2004; Bass, Clements, and Kazman, 2012). According to Alberts (2006) these are considered design threats. In addition, the strength of the relationship between software architecture and project outcome led Poort and van Vliet (2012) to suggest architecting, should be considered an exercise in risk and cost management. Moreover, the cost to repair problems escalates as the software development life-cycle proceeds (Akingbehin, 2005, Figure 1): errors found at the design stage are only 3-6 times more expensive to correct that at the requirements stage, whereas those found at the test stages are 15-70 times more expensive to correct, and those found during the operation stage are 40-1000 times more expensive to correct.

Failure to address the above problems leads to technical debt, a metaphor first coined by Cunningham (1992). Technical debt is the deferred cost (i.e., the debt) of not fixing quality problems before code becomes operational (Curtis, Sappidi, and Szynkarski, 2012). If quality problems remain untreated, technical debt accrues compound interest (Nugroho, Visser, and Kuipers, 2011). For example, if code is difficult to maintain due to architectural coupling problems (Lindvall, Tvedt, and Costa, 2003), code changes will be more expensive (the interest) than they would be if the coupling problems had already been resolved. Interest compounds because each change to the module incurs interest until the problems are resolved. Therefore, for projects to be more successful in terms of budget and schedule, maintainability risks need to be identified early owing to the compounding effect of: (i) architectural design enabling or inhibiting maintainability (Bass, Clements, and Kazman, 2012, p. 40); (ii) costs of repairing problems escalating beyond the design stage (Akingbehin, 2005); and (iii) technical debt resulting in compound interest (Nugroho, Visser, and Kuipers, 2011).
Maintainability problems stem from the limitations of human cognition. 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 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 architectural design complexity will increase the chances of software developers misunderstanding and not fully appreciating all the discrete program modifications needed to correctly propagate an intended change, whether it is to fix bugs or adapt software. This view is supported by software understandability research which has found that more complex software architectures take longer to comprehend, which in turn makes them more difficult to maintain, and increases the risk of mistakes leading to errors (Genero et al., 2003; Stevanetic and Zdun, 2016). It is this limit of human visual processing can result in maintainability risks such as error-proneness and change propagation.

Many different approaches have been proposed for identifying and managing maintainability risks. A survey of by Malhotra and Chug (2016) revealed 96 primary studies of different techniques for making software maintainability predictions. Malhotra and Chug classified the techniques as statistical algorithms, machine-learning algorithms, nature-inspired techniques, expert judgement, and hybrid techniques. Many of these techniques are based on established software design quality metrics (McCabe, 1976; Chidamber and Kemerer, 1994) that measure the complexity, coupling, and cohesion properties of architecture. Greater complexity, tighter coupling, and lack of cohesion are associated with the symptoms of maintainability problems such as error-proneness (Munson and Khoshgoftaar 1992; Coleman et al., 1994; Schroeder 1999; Basili, Briand, and Melo, 1996; Briand et al., 1998, 2000; Yu, Systa, and Muller, 2002; Subramanyam and Krishnan, 2003; and Gyimothy, Ferenc, and Siket, 2005). Other researchers have sought to understand how the symptoms of maintainability problems can be traced back to architectural problems by clustering code into modules, ranking them by relative error and change-proneness, and looking for causal architecture smells (Xiao, Cai, and Kazman, 2014; Mo et al., 2015).

Some researchers have investigated whether problems can be avoided even earlier by providing techniques to support improved architecture decision making (Falessi et al., 2011). Furthermore, others have proposed reusable design patterns to improve the maintainability characteristics of software in particular circumstances (Gamma et al., 1994). Finally, with the emergence of agile methods (Martin, 2002), continuous refactoring based on the output of
static analysis tools has emerged to avoid debt levels spiralling out of control (Vassallo, Palomba, and Gall, 2018).

Despite these different approaches for reducing maintainability risks (decision techniques, design patterns, architecture analysis techniques, and continuous refactoring), there is still an opportunity for improvement. Decision techniques do not explicitly identify the maintainability properties of architectures, which limits measurable improvement. Whilst design patterns are in widespread use, there is little empirical evidence about whether they work in practice (Me, Calero, and Lago, 2016), and pattern misuse can introduce problems. Many architecture analysis techniques require structured upfront designs such as Unified Modelling Language (UML) designs (Object Management Group, 2017). Structured upfront designs may not be available in projects using agile, which favours working software over comprehensive documentation. In a survey of 50 software developers only 15 used UML, 11 of those used UML selectively, and none used UML wholeheartedly (Petre, 2013). However, despite structured designs being lacking and the trend towards agile, Nord, Ozkaya, and Kruchten (2014) explain that architecture is needed in large-scale agile projects to avoid excessive redesign later. After all, relying on continuous refactoring alone only defers the problem and means a proportion of the project budget will be spent on rework. Furthermore, refactoring without an architecture description runs the risk of introducing problems because developers may change code without understanding all the consequences. Nugroho and Chaudron (2014) found that producing UML class diagrams and sequence diagrams reduces defect density in the code and the time required to fix defects. They concluded that the production of UML class diagrams and sequence diagrams may help improve software quality and increase software maintenance productivity. This situation suggests a need for upfront design approaches that are lightweight enough such that they won’t negate the benefits of agility.

The source code based approach proposed by Xiao, Cai, and Kazman (2014) is interesting with respect to investigating new techniques that do not require extensive upfront design. Firstly, because insights are gained based on structural relationships between classes. Petre (2013) found that class diagrams (which convey structural designs) are the UML diagram type that are most likely used by practitioners (marginally more likely to be used than sequence and activity diagrams, but much more likely to be used than state machine and use case diagrams). Secondly, because Xiao, Cai, and Kazman’s technique allows clusters of known scope to be ranked for relative levels of error-proneness risk. Ranking clusters enables architects and developers to visualise the most impactful maintainability problems by comparing risk levels associated with each cluster. This raises the question:
“could the approach proposed by Xiao, Cai, and Kazman be applied to upfront class diagrams rather than source code to create risk views earlier in the life-cycle?”

Applying the approach of Xiao, Cai, and Kazman to upfront class diagrams may enable risks to be isolated into architecture subsets at the design stage in order to produce risk views. Such risk views would allow architects to easily visualise the riskier parts of the architecture and mitigate those risks before proceeding to the implementation stage. A risk container is the term used in this dissertation for architecture subsets into which risks can be isolated. Xiao, Cai, and Kazman demonstrated error-proneness risks can be isolated into DRSpaces. That means DRSpaces are considered a type of risk container. Preceding ‘container’ with ‘risk’ in the term ‘risk container’ distinguishes the proposition from other types of containers used in software engineering such as Docker deployment containers (Docker, 2021) and Linux virtualisation containers (Red Hat, 2021). Thus, the originality of the research is in basing risk assessment around containers that entrap and isolate the risk-causing architectural elements. Simply applying Xiao, Cai, and Kazman’s method to class diagrams is not enough because that assumes class diagrams are the most helpful type of UML diagram for predicting maintainability problems, and that clustering into modules is the most effective way to identify areas of greater maintainability risk. The purpose of the research presented in this dissertation is to address the following question:

**How effective are different types of risk containers at isolating different risks?**

To evaluate the proposition of container-based risk assessment, different types of architectural design containers are trialled to determine how well each type of container isolates different types of implementation risks, and how meaningful they are to the development team. Effective and meaningful containers can be recorded in the project risk register and used to manage the isolated risks throughout the software development life-cycle.

The results of this research demonstrate that: (i) risks can be isolated into containers; (ii) containers can be ranked using risk indicating design metrics; (iii) Design Rule Containers are the most effective risk container type tested; (iv) class diagrams are more helpful than use case diagrams for predicting maintainability risks; and (v) presenting designs as a set of risk container diagrams helps practitioners to mitigate the isolated risks. These properties allow risk assessors to concentrate in the areas of greatest risk and understand the scope of each ranked risk instance in order to manage maintainability risks earlier in the software development life-cycle.
1.2 Research Issues

Conducting the research presented several issues that had to be considered and mitigated in the proposed approach:

RI1: The general scope of architecture risk assessment is very broad and too big for a PhD. The scope needed to be constrained by limiting the types of risks researched.

RI2: Risk isolation is an open concept, and it is not feasible to test all approaches for isolating risks within the scope of a PhD.

RI3: It is difficult to get access to data about industrial software developments owing to the poor availability and commercial sensitivity of such data.

RI4: It is difficult to get access to industry stakeholders for research participation because they are busy.

1.3 Research Glossary

This section defines the key technical terms used throughout this dissertation in alphabetical order.

Architecture Element – ISO 42010 (2011) defines an architecture element as any construct in an architectural design. Scenarios, Use Cases, Components, and Classes are all examples of architecture elements. This dissertation uses the terms architecture element and element interchangeably.

Change propagation – is when a change to element A necessitates a change to element B due to the ripple effect characterised by Lindvall, Tvedt, and Costa (2003). As stated by Bass, Clements, and Kazman (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. 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. Change propagation is relevant to all software development projects because software often changes throughout its lifetime to meet new requirements and fix bugs.

**Error-proneness** – this risk is defined as the additional time and cost of fixing bugs due to architecture elements that are more likely to contain bugs and defects found during testing or operation than other elements. This definition is based on that used by Harrison (1988) who investigated the use of complexity metrics to prioritise testing effort. The terms error(-prone), fault(-prone), and defect(-prone) are interchangeably used as per precedent found in the literature review (Basili, Briand, and Melo, 1996; Briand et al., 1998 and 2000; Yu, Systa, and Muller, 2002; Subramanyam and Krishnan, 2003; Gyimothy, Ferenc, and Siket, 2005; and Xiao, Cai, and Kazman, 2014). Coupling is a significant contributory factor in the complexity of software (Bass, Clements, and Kazman, 2012). Thus, it is expected that more complex parts of an architecture containing elements with greater coupling are more likely to be error-prone during implementation. That is because developers are more likely to make mistakes and introduce bugs in complex areas of the architecture which represent more complex search fields in Wolfe and Horowitz’s (2017) terms. Fixing these bugs introduced by coding mistakes has the potential to impose unexpected cost and time penalties upon projects which, if not managed, will result in projects failing to meet budgets and schedules (Charette, 2004). All software development projects have the potential to be adversely impacted by unmanaged error-proneness risks because all projects have the potential for software developers to make mistakes when working with complex and highly coupled code.

**Maintainability Risk** – is defined as the probability of maintainability problems resulting from a design and the severity of those maintainability problems. This definition extends the definition of risk below and is adapted from that used by Goseva-Popstojanova et al. (2003), which is the probability of malfunctioning and the severity of malfunctioning. Malfunctioning is replaced with maintainability problems to indicate that this dissertation focuses on maintainability risks. Design is included in the definition to indicate that this dissertation focuses on maintainability risks resulting from design threats (Alberts, 2006). Error-proneness and excessive change propagation are examples of maintainability risks.

**Meaningful** - To satisfy their purpose, risk containers must act as a useful device for mitigating risks and it must be possible to create them from design artefacts used in practice.
Even if it were proven beyond all doubt that a type of risk container can isolate instances of a risk, the containers are only meaningful if practitioners can use them to understand what is inducing the risk. In summary, risk containers are meaningful to practitioners if they can:

M1: be used to isolate real project risks;
M2: be formed from design artefacts used in practice;
M3: help partitioners locate risk inducing architectural flaws more easily.

Risk – ISO 31000 (2018) defines risk as the effect of uncertainty on objectives. This dissertation defines a risk as the probability and severity of an undesired event or trait that could impact objectives. For example, the probability of software being difficult to maintain, resulting in excessive maintenance costs due to error-proneness and change propagation, which threatens schedule and budgetary objectives.

Risk Container – a largely self-contained subset of elements in an architectural design, which can be ranked by relative risk indicators and easily reasoned about such that they are risk isolating. Whilst the goal is to have completely self-contained risk containers that do not share architecture elements with other containers, it is expected owing to coupling that containers will overlap to some degree.

Risk Container Type – classifies risk containers extracted from an architectural design using a specific method or algorithm.

Risk Indicator – Evidence of a risk being present that can be determined from a software design. A risk indicator could be a design metric such as Chidamber and Kemerer’s (1994) Coupling Between Objects for error-proneness, or Change Propagation Probability when concerned about an architecture being expensive to extend (Abdelmoez, Goseva-Popstojanova, and Ammar, 2006). However, there is no reason why the risk indicator couldn’t be a qualitative review performed by software engineers.

Risk Isolating - is defined as architecture elements that cause a risk being isolated within the bounds of a risk container. Thus, for a risk container type to be considered risk isolating, it must be possible for risks to be isolated within containers of that type. It must be possible to predict which containers isolate risks to use risk containers to analyse upfront designs. Ideally the risk inducing architecture elements should not be shared with other containers for the risk to have
been truly isolated within the bounds of a container. Thus, the most risk isolating containers will contain the risk inducing elements and have the least amount of overlap with other containers.

**Risk Model** – an architecture model that describes how architecture elements can induce and are impacted by risks. This definition is based on the ISO 42010 (2011) definition of an Architecture Model.

**Risk Predicting** – a container type is considered risk predicting if container metrics calculated from the design are predictive of implementation problems such as error-prone-ness and excessive change propagation.

**Risk View** – an architecture view that shows where the risks are within an architecture model. This definition is based on the ISO 42010 (2011) definition of an Architecture View.

**1.4 Research Problem**

The research problem investigated by this dissertation is to understand whether the architecture elements that may lead to error-prone-ness and change propagation risks can be isolated into risk containers. Isolation of error-prone-ness and change propagation risks into risk containers would enable development teams to focus mitigations on the highest risk areas. For example, developers could recommend redesigning a complex area. Alternatively, if redesign is not possible, developers could recommend focusing pair programming, peer review, and test effort on the containers representing areas that are riskier. Prioritising testing based on ranked risk containers is in keeping with the proposition of Harrison (1988) to use software metrics for allocating test resources.

To test whether risk container types are risk isolating the question is decomposed into two parts:

1. How element isolating (self-contained) is the container type, i.e. to what degree do containers overlap and share elements, and how much of the coupling is between elements in the same container? A completely element isolating container is one where the elements of which it is composed are not shared with other containers, and those elements are not coupled to elements in other containers.
2. Are container level design metrics predictive of the risk isolated, i.e. do container metrics calculated from designs correlate to container risk indicators calculated from the implementation?
If a container type is both element isolating and risk predicting it is considered risk isolating. 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. If the 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 container types that satisfy the definition of being meaningful to practitioners would support the motivation because they isolate risks real project risks (M1), can be formed from design artefacts used in practice (M2), and help practitioners locate risk inducing architectural flaws more easily (M3). Thus, containers that are risk isolating and meaningful to practitioners support the proposition that they are useful for risk management because the technical insight gleaned from the containers and their risk indicators support impact assessment and better inform risk management (e.g., redesign, 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 to avoid unnecessary technical debt, whilst driving project progress forward.

1.5 Overall Proposal

Figure 1-1 presents a UML activity diagram illustrating a design for a container-based software architecture risk assessment process. It is supposed to be used once some degree of architectural description has been drafted.

It is envisaged that different types of risk container may be more suited to isolating specific risks than others. For example, whilst DRSpaces have been shown to isolate error-proneness risks (Xiao, Cai, Kazman, 2014), risk containers based on attack graphs may be more suited to isolating security risks (Said et al., 2011). Thus, the process needs to allow different types of containers to be used for different types of risk. Therefore, it was decided to take the same approach as Stoermer, Bachmann, and Verhoef (2003), whose abstract review process is not prescriptive of how each of the activities are fulfilled. Basing container-based software architecture risk assessment on an abstract review process enables different container types to be used for different types of risk, whilst still following the same overall process.

In order to produce risk views that highlight the riskier parts, it was decided to base the process on ranking risks like the approach by Xu, Huang, and Wei (2010). However, instead of
ranking whole architectures, the process ranks subsets of the architecture (i.e., risk containers) for the risks of concern. The fundamental difference between the proposed process and those discussed in the literature review is that it attempts to isolate risks into containers so that they may be managed.

Once an architecture has been developed (Figure 1-1, activity 1), the first step is to work with stakeholders to identify their risks of concern (Figure 1-1, activity 2). This is similar to eliciting non-functional requirements and quality attributes, but takes a pessimistic perspective. For example, instead of requiring the architecture to be easy to develop and maintain, it would be specified that the risk of error-proneness must be minimised. Xu, Huang, and Wei (2010) suggested that risk and quality attribute-based approaches could be complementary, i.e. avoiding risks and achieving desired quality attributes are both important to the goal of achieving successful software development projects. Risk container types and indicators are selected once the risks of concern have been identified by the stakeholders (Figure 1-1, activity 3). A risk indicator could be a design metric such as Chidamber and Kemerer’s (1994) Coupling Between Objects for error proneness, or Change Propagation Probability when concerned about an architecture being expensive to extend (Abdelmoez, Goseva-Popstojanova, and Ammar,
However, there is no reason why the risk indicator couldn’t be a qualitative review performed by software engineers.

Once the container types have been selected, the architectural description is split up using rules specified for that type of container (Figure 1-1, activity 4). Once risk indicators have been chosen, and the architectural description has been split into risk containers, the risk indicators will be resolved to determine which containers are predicted to isolate the greatest risk (Figure 1-1, activity 5). In the case of metrics, container level metrics shall be calculated. In the case of a qualitative review, it could be as simple as the software engineers scoring the containers for their perceived risk. Once the risk indicators have been calculated, the containers shall be ranked from high to low risk (Figure 1-1, activity 6). Then a decision shall be considered as to whether any of the risk containers should be re-factored before proceeding to implementation (Figure 1-1, activity 7). The calculation of the risk indicators and any subsequent refactoring iterates until the riskiness of all containers is accepted by the stakeholders.

Finally, if any containers having risk indicators that are still cause for concern remain, they are added to the project risk register so they can be managed during implementation (Figure 1-1, activity 8). For example, by assigning more experienced staff or using pair programming to implement those risky areas of the architecture. I propose that remaining risks are stored in an architecture risk model that is part of the architecture description, to enable the results of the risk container analysis to be referenceable from the architecture. Doing so will enable architects to identify elements that give rise to the risk and the elements that are potentially impacted by the risk in the future. The architecture risk model and its views represent an implementation of a risk register for architects to use. A proposed specification for an architecture model is evaluated as part of the research scope.

1.6 Research Scope

The process diagram and description presented above (section 1.5) aid comprehension of how the techniques tested in this dissertation may fit into a process. The research in this dissertation concentrates in the area of Figure 1-1 labelled Scope of Research. The work concentrates on those activities in which architectural design can be split into containers and container level risk indicators can be derived. Therefore, the research concentrates on activities 3, 4, 5, and 6 as well as the risk model, because they are central to the proposition of container-based risk assessment. The activities called Identify Risks of Concern and Add to Risk Register are out of scope because they are agnostic regarding isolating risks in the architectural description.
Refactor Worst is out of scope because the proposed research is about isolating risks as opposed to the specific mitigations needed to reduce the impact of predicted risks.

The work in this dissertation aimed to produce the following deliverables for software architecture practitioners to use:

1. Risk container formulation rules for the container types tested.
2. Container level design metric formulae for the risks tested.
3. A proposed architecture risk model.

The work was also bounded by the number of risks tested. The risks selected for testing are error-proneness and change propagation as defined in section 1.3. Bengtsson et al. (2004) were strongly motivated by their statement that “several studies have shown that 50-70% of the total lifecycle cost for a software system is spent on evolving the system” (p. 129). A more recent study found that almost a quarter of all developers’ working time is wasted due to technical debt relating to maintainability and evolvability (Besker, Martini, and Bosch, 2019). Therefore, these two maintainability risks were selected because they are relevant to all software development projects, can lead to cost and schedule overruns if they turn into implementation issues, and can be mitigated by architectural refactoring and allocation of experienced resources if identified early. This is consistent with the research motivation of helping software projects be more successful in terms of budget and schedule. The overall research approach was to study software development projects and compare the risk indicators calculated from the design to real project outcomes in terms of project metrics, risk registers, and developer feedback.

The project was also bounded by the number of container types tested to: (i) **Design Rule Containers** which contain the classes that are based on a modularising design rule; (ii) **Use Case Containers** which contain classes in the sequence diagram of a use case; (iii) and **Resource Containers** which contain classes that depend on external resources such as database tables, services, and files. These container types were selected because they represent different and commonly used approaches to decomposing software systems from different architectural perspectives. They allow evaluation of different container types, perspectives, and diagrams to determine which ones help analysis of the selected maintainability risks (error-proneness and change propagation) and were first introduced in Leigh et al. (2016 and 2017).
1.7 Research Questions

The project investigated the following research questions as the basis of trialling containers for the individual process activities:

RQ1: How effective are Design Rule, Resource, and Use Case Containers at predicting and isolating the risks of implementation error-proneness and implementation change propagation?

RQ2: Does splitting an architectural design into smaller container diagrams help practitioners find error-inducing flaws and identify change impacts?

RQ3: What information is necessary in architecture risk models to describe outputs of architecture analysis techniques?

1.8 Hypotheses

The following hypotheses were tested to answer the proposed research questions.

RQ1: How effective are Design Rule, Resource, and Use Case Containers at predicting and isolating the risks of implementation error-proneness and implementation change propagation?

H1: If error-proneness and change propagation risks are isolated into Design Rule, Resource, and Use Case Containers, the risk inducing architectural features, e.g. components or classes, should exist in relatively few risk containers.

H2: If error-proneness and change propagation risks are isolated into Design Rule, Resource, and Use Case Containers, there should be a strong correlation between container level risk indicators and container level project outcomes.

H3: If error-proneness and change propagation risks are isolated into Design Rule, Resource, and Use Case Containers, the set of files that must be changed to fix errors or implement changes should fit neatly into containers.

H4: If error-proneness and change propagation risks are isolated into containers, there should be a correspondence between developer perception of containers that were error-prone or associated with excessive change propagation, and container level risk indicators.
Unless the causal agent of a risk is removed from a container, the risk should be present beyond design and throughout implementation and testing of the software. Therefore, the container’s relative riskiness should remain consistent. If the risk has been isolated by the rules used to split the architecture into containers, then a correlation between the containers with the highest risk indicators and project outcome should be observable and thus they are also predicting. The most effective container type for a specific risk shall be the one that is on balance most risk isolating and predicting. That is because if a container were risk predicting but not risk isolating, it is more difficult to manage because it does not represent a subset of the architecture that relates to a risk instance. Section 1.3 defines risk isolating as how well the container isolates the collection of architectural elements that contribute to a risk instance. The most isolating container would be the one that contains all the contributing elements but is least ‘contaminated’ with elements that are unrelated to the risk instance. Therefore, testing hypotheses H1-H4 supports answering RQ1 and evaluates meaningfulness criteria M1 and M2 to determine which container types are the most isolating for each risk and can be formed from design artefacts used in practice.

RQ2: Does splitting an architectural design into smaller container diagrams help practitioners find error-inducing flaws and identify change impacts?

H5: Smaller Design Rule, Resource, and Use Case Containers diagrams should help practitioners find error-inducing design flaws and identify change impacts more easily than larger architecture diagrams.

The purpose of RQ2 and H5 is to evaluate whether meaningfulness criterion M3 is satisfied by risk containers, i.e. do they help practitioners locate risk inducing architectural design flaws more easily?

RQ3: What information is necessary in architecture risk models to describe outputs of architecture analysis techniques?

H6: Practitioners should view an architecture risk model as being beneficial to risk management.

H7: Practitioners should not think that concepts are missing from the architecture risk model or that risk model concepts are redundant.
The purpose of RQ3, H6, and H7 is to evaluate whether modelling the results of risk container analysis and presenting risks views might also help practitioners to better manage the risks.

1.9 Method

The overall methodology used to develop the work includes several activities ranging from literature review to analysis of existing industrial software development projects, experiments, evaluation, and practitioner surveys. More specifically, the literature review was used to identify existing architecture analysis techniques that can be used to predict maintainability risks, design metrics that are predictive of error-proneness and change propagation, and risk models. Industrial projects were selected and analysed to test hypotheses H1-H4 and answer RQ1 in order to determine which design container types are the most error-proneness and change propagation predicting and isolating. This was done using the standard design, error-proneness, and change propagation metrics identified in the literature review. An online experiment provides evidence whether presenting a design as a set of containers helps developers to identify error inducing flaws and the impact (propagation) of changes. The results of the online experiment are used to assess whether smaller container diagrams help developers find error-inducing flaws and identify change impacts (RQ2; H5). A combination of literature review and a practitioner survey provides evidence whether a proposed architecture risk model is applicable to risk management (RQ3; H6-H7).

The research is reviewed for construct validity threats using Messick’s (1987) classification of consequential, content, substantive, structural, external, and generalizability threats throughout Chapters 4-8.

1.10 Contributions

The research presented in this dissertation makes the research contributions listed in Table 1-1 in relation to the goal of understanding whether risk containers isolate risks, are meaningful to practitioners, and whether storing isolated risks in an architecture risk model would be beneficial to risk management. The contributions are annotated in Table 1-1 with the research questions they address and the sections in which they appear.
Table 1-1 Research Contributions

<table>
<thead>
<tr>
<th>Contribution Identifier</th>
<th>Description</th>
<th>Research Question</th>
<th>Appears In</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Detailed descriptions of the container population algorithms.</td>
<td>RQ1</td>
<td>Chapter 4, Section 4.1</td>
</tr>
<tr>
<td>C2</td>
<td>Evidence whether containers can be derived from designs used in practice.</td>
<td>RQ1</td>
<td>Chapter 4, Sections 4.3-4.4</td>
</tr>
<tr>
<td>C3</td>
<td>A formal definition of the container element isolation metrics used.</td>
<td>RQ1</td>
<td>Chapter 4, Section 4.5</td>
</tr>
<tr>
<td>C4</td>
<td>Identification of the container type that is most element isolating (least overlapping).</td>
<td>RQ1</td>
<td>Chapter 4, Section 4.5</td>
</tr>
<tr>
<td>C5</td>
<td>Formally defined metrics for container level error-proneness.</td>
<td>RQ1</td>
<td>Chapter 5, Section 5.1</td>
</tr>
<tr>
<td>C6</td>
<td>Evaluation of each risk container type's ability to predict and isolate implementation error-proneness.</td>
<td>RQ1</td>
<td>Chapter 5, Sections 5.2-5.6</td>
</tr>
<tr>
<td>C7</td>
<td>Formally defined metrics for container level change propagation.</td>
<td>RQ1</td>
<td>Chapter 6, Section 6.1</td>
</tr>
<tr>
<td>C8</td>
<td>Evaluation of each risk container type's ability to predict and isolate implementation change propagation.</td>
<td>RQ1</td>
<td>Chapter 6, Sections 6.2-6.4</td>
</tr>
<tr>
<td>C9</td>
<td>Evidence about whether smaller risk container diagrams help practitioners locate error inducing design flaws and impact changes more easily.</td>
<td>RQ2</td>
<td>Chapter 7, Sections 7.1-7.6</td>
</tr>
<tr>
<td>C10</td>
<td>An architecture risk model.</td>
<td>RQ3</td>
<td>Chapter 8, Sections 8.1-8.4</td>
</tr>
</tbody>
</table>

1.11 Published Work

Concepts and results from the work in this dissertation have appeared in five publications (Leigh 2017; Leigh, Wermelinger, and Zisman, 2016, 2017, 2019, and 2021). This section briefly introduces how each of those publications relates to the scope of the work.
‘An Evaluation of Design Rule Spaces as Risk Containers’ introduces the high-level concept of a risk container and provides a basic evaluation of creating Xiao, Cai, and Kazman’s (2014) Design Rule Spaces from UML design diagrams and their usage to isolate error-proneness risks in an industry project (Leigh, Wermelinger, and Zisman, 2016).

‘Architecture Analysis to Predict Project Risks – Using Containers to Aid Risk Assessment’ summarises the research undertaken by this dissertation (Leigh, 2017). It explains the problem and its importance, previous work, proposed approach, evaluation plan, and expected results. This paper was presented and discussed at the European Conference on Software Architecture’s Doctoral Symposium.

‘Software Architecture Risk Containers’ more fully explores the concept of risk container in literature to identify alternative types of risk container (Leigh, Wermelinger, and Zisman, 2017). In addition, the paper extended the error-proneness risk isolation evaluation to a second industry project and the three types of risk containers chosen for the dissertation: Design Rule Containers, Use Case Containers, and Resource Containers.

‘Risk Containers: A Help or Hindrance to Practitioners?’ introduces the initial definition of risk containers being meaningful to practitioners (Leigh, Wermelinger, and Zisman, 2019). The full definition is presented in section 1.5 of this dissertation. The paper also describes the early results of an experiment designed to test whether participants find it easier to use smaller risk container diagrams than larger architecture diagrams to: a.) locate risk inducing design flaws; and b.) correctly identify the impact of making changes.

‘Evaluating the effectiveness of risk container types to isolate Change Propagation in Software Architectures’ is the first publication to provide an algorithmic description of the Design Rule, Use Case, and Resource Container population methods, and formal definitions of the container element isolation and change propagation metrics (Leigh, Wermelinger, and Zisman, 2021). This paper presented the results of testing the three container types for their ability to isolate change propagation risks in the same two industry projects used to test error-proneness as well as in two additional student research projects.

1.12 Dissertation Structure

The remainder of this dissertation is structured as follows. Chapter 2 reviews related literature to establish: (i) what techniques already exist for identifying maintainability risks in
software architectures; (ii) what design metrics are already available for predicting maintainability risks; before (iii) summarising the gap addressed by this research.

Chapter 3 explains the overall method and summary of work done to answer the research questions.

Chapter 4 explains the method used to evaluate whether risk containers can be extracted from designs, whether container types are element isolating, and the results obtained (contributions C1, C2, C3, C4).

Chapter 5 presents the method used to determine whether risk container metrics predict error-proneness risks and the results obtained (contributions C5 and C6).

Chapter 6 describes the method used to determine whether risk container metrics predict change propagation risks and the results obtained (contributions C7 and C8).

Chapter 7 presents the method and results of the online experiment to determine whether smaller container diagrams help practitioners to find error inducing design flaws and impact changes more easily (contribution C9).

Chapter 8 proposes an architecture risk model based on requirements derived from the literature review and an evaluation of it based on a practitioner survey (contribution C10).

Chapter 9 discusses the overall research findings, the practical applications of the research, potential future research possibilities, and concluding remarks.
Chapter 2 - Literature Review

This chapter presents the results of a systematic search to find existing software architecture maintainability analysis techniques, a detailed review of the analysis techniques found that can predict error-proneness and change propagation, a review of the existing software design metrics that have been associated with predicting error-proneness and change propagation, and a review of previous work relating to architecture risk models. The chapter ends by identifying the gap between the existing approaches found and the approach investigated by the research described in this dissertation.

2.1 Identification of Maintainability Risks in Architectures

This subsection describes the existing techniques that can be used to identify maintainability risks such as error-proneness and change propagation in software architectures.

2.1.1 Search Criteria

Using the definition of risk introduced in section 1.3, techniques can determine a risk if they can indicate the likelihood of a desired achievement level of a quality attribute not being attained. Lots of techniques simply determine whether one alternative is better than another by deriving some measure of quality attribute support. Whilst that might stimulate a redesign if attainment is predicted to be low, it does not identify which parts of the architecture carry more risk than others. Given the overall goal to produce models and views of isolated risks within software architectures, the literature is assessed using the following criteria:

1. Risk Output – does the technique provide an explicit risk output for maintainability risks such as error-proneness or change propagation? I.e., some prediction of the likelihood of an undesired event, e.g. a relative risk indicator.

2. Risk Mapping – does the technique allow specific risks to be attributed to architecture subsets?

The review begins by searching the literature to identify architecture analysis techniques that satisfy criteria 1 and 2.
2.1.2 Search Method

Okoli’s (2015) systematic method of literature reviewing was used to plan, select, extract, and execute, the review of existing techniques capable of identifying and isolating maintainability risks in software architectures. During Okoli’s planning stage, the purpose of the literature review was developed and expressed as the following research question for selecting techniques that satisfy the Risk Output and Risk Mapping search criteria introduced in the preceding section:

**To what extent can existing architecture analysis techniques help identify risks (Risk Output) in software architectures and relate the risks to Architecture Description elements (Risk Mapping)?**

Software architecture is a mature research field, and as such, many literature surveys have already been published. Therefore, the Okali’s selection stage was initially based on a set of widely cited surveys by Dobrica and Niemelä (2002), Babar and Gorton (2004), Babar, Zhu, and Jeffery (2004), Mattsson, Grahn, and Mårtensson (2006), Bernardi, Merseguer, and Petriu (2012), and Malhotra and Chug (2016). The Malhotra and Chug survey was of particular interest owing to it being up to date at the beginning of this research project and being based on a very similar research question: “what techniques have been used for the software maintainability predictions?”, p. 1226. At the time of writing, these surveys had more than one thousand Google Scholar Citations. The Open University’s library search tool was used to fill in the gap since 2016 by searching items containing the terms “‘software architecture’ AND (‘bug-proneness’ OR ‘error-proneness’ OR ‘fault-proneness’ OR ‘Change propagation’)” in articles, conference proceedings, and journals. Only papers less than twenty years old were included. “Software architecture” was used as a search term in preference to “software design” because the latter produced less relevant results including implementation-based studies and user interface design studies. The approach of using surveys and a structured search is an example of protocol-driven searching (Greenhalgh and Peacock, 2005). The papers found in the surveys and search results were augmented with additional papers found by snowballing. Snowballing involves checking the references of the initial results to see if they yielded additional papers of interest as explained by Greenhalgh and Peacock. The superset of the papers described in the six surveys and search results contained 268 papers for consideration. Basing the review on these widely cited surveys and searches provides confidence that the research presented in this dissertation addresses a gap in the literature.
Okali’s extraction stage was performed by eliminating techniques that do not satisfy the risk output and risk mapping criteria. Techniques were eliminated for the following reasons: not risk outputting, not useful for maintainability risks, a metric definition rather than an analysis technique, requires input not available in the design, and being a duplicate of another proposal. These eliminations reduced the set of techniques to those shown in Table 2-1, all of which were found to satisfy the Risk Output and Risk Mapping criteria defined in section 2.1.1.

Table 2-1 Results of Literature Search

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Identifier</th>
<th>Quality Attribute</th>
<th>Risk Output</th>
<th>Risk Mapping (Architecture Subset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1994</td>
<td>Kazman et al.</td>
<td>SAAM</td>
<td>Multiple</td>
<td>QA not sufficiently satisfied</td>
<td>Arbitrary architecture subset</td>
</tr>
<tr>
<td>1997</td>
<td>Lung et al.</td>
<td>SAAMER</td>
<td>Reusability</td>
<td>QA not sufficiently satisfied</td>
<td>Arbitrary architecture subset</td>
</tr>
<tr>
<td>1999</td>
<td>Lassing, Rijsenbrij, and van Vliet</td>
<td>SAAMCS</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Arbitrary architecture subset</td>
</tr>
<tr>
<td>1999</td>
<td>Molter</td>
<td>ISAAMCR</td>
<td>Reusability</td>
<td>QA not sufficiently satisfied</td>
<td>Arbitrary architecture subset</td>
</tr>
<tr>
<td>2000</td>
<td>Kazman, Klein, and Clements</td>
<td>ATAM</td>
<td>Multiple</td>
<td>Generic Risk Assessment</td>
<td>Scenario</td>
</tr>
<tr>
<td>2000</td>
<td>Clements</td>
<td>ARID</td>
<td>Multiple</td>
<td>QA not sufficiently satisfied</td>
<td>Arbitrary architecture subset</td>
</tr>
<tr>
<td>2002</td>
<td>Kazman, Asundi, and Klein</td>
<td>CBAM</td>
<td>Multiple</td>
<td>QAs not sufficiently satisfied</td>
<td>Scenario</td>
</tr>
<tr>
<td>2003</td>
<td>Genero et al.</td>
<td>Maintainability</td>
<td>Difficult to maintain</td>
<td>Arbitrary architecture subset (UML Class Diagram)</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Bengtsson et al.</td>
<td>ALMA</td>
<td>Maintainability</td>
<td>Change Propagation</td>
<td>Change Scenario (Arbitrary architecture subset)</td>
</tr>
<tr>
<td>2005</td>
<td>Thwin and Quah</td>
<td>Maintainability</td>
<td>Error-proneness</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>van Koten and Gray</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>Zhou and Leung</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Shaik, Abdelmoez, and Ammar</td>
<td>SARA</td>
<td>Reliability, Maintainability</td>
<td>Error-propagation, Change propagation</td>
<td>Component</td>
</tr>
<tr>
<td>2009</td>
<td>Wang et al.</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>Dubey, Rana, and Dash</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>Ye, Zhu, and Wang</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Type</td>
<td>Description</td>
<td>Details</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>Kaur, Kaur, and Pathak</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>&quot;Modules&quot;, presumed to be Classes</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>Malhotra and Chug</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>Xiao, Cai, and Kazman</td>
<td>Maintainability</td>
<td>Error-proneness</td>
<td>DRSpaces</td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>Kumar and Rath</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Object Oriented System</td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>Stevanetic and Zdun</td>
<td>Maintainability</td>
<td>Implicitly error-proneness and change propagation due to lack of understandability</td>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>Weifeng et al.</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Whole architecture</td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>Li et al.</td>
<td>Maintainability</td>
<td>QA not sufficiently satisfied</td>
<td>Influential Modules</td>
<td></td>
</tr>
</tbody>
</table>

Whilst some of the techniques can attribute risks to specific types of architecture subsets, for example Xiao, Cai, and Kazman’s (2014) method attributes error-proneness risks to Design Rule Spaces, other techniques can be used to compare arbitrary architecture subsets, for example Kazman et al. (1994) demonstrated how the Software Architecture Analysis Method could be used to compare alternative user interfaces.

The approach of basing the literature review on: (i) six widely cited surveys, including Malhotra and Chug’s more recent survey focusing on techniques for predicting maintainability risks; (ii) using The Open University’s library search to fill in the gap since 2016; (iii) snowballing to find additional of papers of interest in the references of (i) and (ii); and then (iv) filtering the techniques found in (i-iii) by the Risk Outputting and Risk Mapping criteria defined in section 2.1.1, is justifiable, repeatable, and should be largely complete. It is justifiable because the Risk Outputting and Risk Mapping criteria determine the extent to which any existing software architecture analysis techniques support the risk container proposition, i.e. these criteria demonstrate that the research presented in this thesis represents a gap in the literature. It is repeatable because the surveys provide an invariant starting set and the search used to find techniques since 2016 was based on specific and structured search terms. No literature review can be considered complete, and as explained by Greenhalgh and Peacock (2005) all approaches including protocol-driven search strategies, snowballing, as well as informal approaches such as “asking around” have pros and cons. However, by combining surveys, protocol-driven searching, and snowballing within Okali’s systematic method, we can be confident the literature review exhibits a high degree of completeness.
The next four subsections critically review each of the down selected techniques shown in Table 2-1 to determine whether they isolated error-proneness and change propagation and already satisfy the risk container proposition.

### 2.1.3 Generic Risk Identification

The Software Architecture Analysis Method (SAAM) developed by Kazman et al. (1994) evaluates architecture alternatives using scenarios that are representative of quality attribute satisfaction. It is a general-purpose technique that could be used with architectural descriptions of different levels of abstraction and applied to a wide range of quality attributes including maintainability. Kazman et al. demonstrated how three alternative user interface architectures can be tested for maintainability using SAAM. The concrete tasks they used were changing the toolkit and adding a menu item. The assessor works out what must be done to accomplish each task. The architecture that allows these tasks to be done with the least effort is judged the most maintainable. Whilst highly flexible, its effectiveness is limited by the skill of the architect to develop scenarios that rigorously exercise the architecture. In addition, Bengtsson et al. (2004) criticised SAAM as being not repeatable because it is too dependent upon the experience of the assessors. These limitations could be overcome by using a broad range of assessors and independent assessors. Another way to add rigour to the method would be to combine the inputs of the assessors with Multi Criteria Decision Analysis techniques (Ishizaka and Nemery, 2013) to reduce the effect of subjective opinions. However, these mitigations to increase repeatability would increase project costs and still fail to address the problem that weak scenarios will lead to a weak evaluation.

Kazman et al. highlighted two difficulties facing evaluation and used those difficulties to justify flexible scenario-based approaches such as SAAM. Firstly, there was no common language for describing different architectures, and secondly, there is no clear way of understanding an architecture with respect to quality attributes. Almost three decades later, and these limitations in support of SAAM are no longer true. ISO 42010 (2011) has since standardised how architectures are described, and it relates stakeholder concerns (including quality attributes) to the architecture so that the architectural description can convey how support for quality attributes is required and achieved by the architecture. In addition, and as detailed in section 2.2 of this chapter, there are now many metrics proven to be indicators of maintainability satisfaction (McCabe 1976; Chidamber and Kemerer, 1994; Mitchell and Mancoridis, 2006). Furthermore, UML (Object Management Group, 2017) is now a widespread and popular language for describing different architectures. Although the emergence of agile has limited
UML use in many projects, Petre (2013) found 22% of developers surveyed use UML to some degree. Thus, the emergence of standard maintainability metrics, a standard for describing architectures, and a language for modelling architectures means the limitations justifying the SAAM approach are no longer present.

The user interface SAAM identified as having the least support for maintainability in Kazman et al.’s (1994) evaluation carries the most risk with respect to the concern of maintainability. Therefore, SAAM satisfies the Risk Output and Risk Mapping criteria because it can identify the risk due to lack of support for maintainability associated with the user interface subset of the architecture. However, being able to identify that the user interface contains a maintainability risk is very coarse, tantamount to being able to locate something within a continent as opposed to a country, province, or neighbourhood. A more modern technique called Design Rule Spaces (Xiao, Cai, and Kazman, 2014), albeit based on architecture relationships reverse engineered from code, can identify maintainability risks with much more granularity. Furthermore, creating a maintainability risk view that covers the whole architecture would need multiple SAAM evaluations, one for each subset of the architecture, which would become very labour intensive by comparison to the automated maintainability prediction techniques described in Malhotra and Chug’s (2016) survey.

Kazman, Klein, and Clements (2000) proposed the Architectural Tradeoff Analysis Method (ATAM) to improve upon SAAM by the addition of trade-off analysis. The novelty of ATAM suggested by Kazman, Klein, and Clements is that it was first to recognise relationships between quality attributes. Kazman, Klein, and Clements provided a case study whereby a performance, availability, and security analysis is performed and traded off for three architectures. In their example, they showed that both performance and availability were improved by increasing the number of servers, but security was negatively affected by the number of servers. Bass et al. (2007) retrospectively reviewed the results of eighteen ATAM evaluations. The results showed that the most prevalent risk themes were performance, requirements, awareness, and unrecognized needs, as these were exhibited in 50% of evaluations. It is however unclear whether these risk categories are more important to the generalised success criteria of software projects, or whether ATAM is better at uncovering these risks than others. Bass et al. uncovered three pieces of advice for software architects. Firstly, to carefully consider which activities and entities they should be coordinating. Their results showed that risks due to omission (not in the architectural description) were more numerous than risks of commission (risks due to architectural decisions). Secondly, that there are lots of risks in the awareness category which means the architect must communicate the implications of the architecture to the organisation.
Thirdly, interfaces should be more abstract to deal with risks stemming from late or unspecified requirements. Bass et al. concluded it is important to express risk themes in terms of the explicit consequences of what could happen if the risk is not addressed. The ATAM demonstrations presented by Kazman, Klein, and Clements and Bass et al. confirm that ATAM satisfies the first selection criterion (Risk Output) and it inherits being Risk Mapping from being an enhancement of SAAM. ATAM has the same drawbacks as SAAM: reliance on assessor skill, being labour intensive, and lack of reproducibility. Furthermore, the addition of trade-off analysis between quality attributes offers no particular advantage for the production of maintainability risk views.

In Active Design Reviews (ADR) participants answer carefully phrased questions to test their knowledge of a design. Participants will therefore be unable to answer the review question if they haven’t read the design. Clements (2000) blended ATAM with ADR to develop a new and more rigorous technique called Active Reviews for Intermediate Designs (ARID). ARID uses ATAM-like prioritised stakeholder scenarios to exercise the design rather than questions. Several limitations were identified by Clements such as the architect’s presence in the review is crucial, its output is dependent upon the quality and experience of the reviewers, and it won’t work if architectural design documentation isn’t available. ADR satisfies the Risk Output and Risk Mapping criteria by virtue of it being based on ATAM, however that also means ARID has the same limitations with respect to creating maintainability risk views already described.

Kazman, Asundi, and Klein (2002) proposed the Cost Benefit Analysis Method (CBAM) as a derivative of ATAM. CBAM compares how well Architectural Strategies (AS) achieve a set of quality attributes for a set of prioritised scenarios. Participants then choose ASs based on return on investment, subject to cost and schedule constraints. Finally, a sanity check to confirm the recommended AS aligns with the organisation’s business goals is performed by intuition. Moore et al. (2003), highlighted several limitations found when applying CBAM to NASA’s Earth Observing System Data Information System Core System project. The difficulties encountered were dealing with large numbers of AS, short attention span of stakeholders, subjective interpretations of quality attributes, faking scores to bias their own priorities, and lack of common understanding of current response levels by stakeholders. They proposed enhancements in CBAM2. Firstly, changing it to an iterative process enabling more detail to be captured without requiring long sessions with stakeholders. Secondly, using more concrete scenarios to avoid differing interpretations of quality attributes. Thirdly, the current state was elicited and shared so all stakeholders had a better view of the status quo before considering their inputs. Fourth, a sensitivity of utility for each quality attribute response level was introduced. Despite these improvements, CBAM2 does not address the limitations it inherits
from SAAM and ATAM (reliance on assessor skill, being labour intensive, and lack of reproducibility). However, an advantage of CBAM2 over its predecessors is that it recognises the financial consequences of risk. Furthermore, CBAM includes more explicit affordance for risk assessment by including an activity whereby important risk factors that pertain to the ASs are discussed. The impact, probability of occurrence, and potential mitigations are assessed for each risk. The relative risk probabilities are then used to determine whether each AS is considered riskier than the others. Therefore, the AS return on investment ranking produced by CBAM is modulated by riskiness.

One difficulty assessing architectures for risks is that if the requirements are not well understood or conflicting, those risks cannot be mitigated by changing the architecture. Therefore, Kazman, In, and Chen (2005) added a win-win requirements negotiation step to CBAM creating a new method called WinCBAM which begins by eliciting the win conditions before identifying conflict issues. The conflict issues are then fed into the first stage of CBAM during which the ASs are explored. WinCBAM concludes by a further additional step to reach agreements at the end of the main CBAM process. If agreement cannot be reached another iteration is conducted. Kazman, In, and Chen conceded that testing in a large scale industrial setting would be difficult to achieve and so it is difficult to draw any firm conclusions based on the limited case study presented. However, the rationale behind the WinCBAM proposition highlights a clear limitation of the work presented in this dissertation which is that risk container analysis cannot help with risks introduced before the design stage of the software development life-cycle.

Zhu, Staples, and Jeffery (2008) contest that approaches such as ATAM and SAAM don’t scale to Ultra Large Scale (ULS) systems because of their strict step by step processing and focus on structural architecture. Zhu, Staples, and Jeffery assert that for such ULS systems it is more important to focus on quality-centric architectural rules rather than structural views of components and connectors. That belief is born from their work with the mortgage industry where problems faced are decentralisation, conflicting requirements, continuous evolution, and blurred system/people boundaries. To evaluate rule-centric architectures, they have proposed the Evaluation Process for Rule Centric Architectures (EPRA) a four phase process of: defining business goals and quality scenarios, rule analysis, elicit architectural tactics, and trade-off analysis between the rules. However, with respect to maintainability and in contradiction to Zhu, Staples, and Jeffery’s view (2008), it is clear that focusing on components and connectors is important because studies have shown that quality metrics calculated from components and connectors are predictive of maintainability problems (Munson and Khoshgoftaar, 1992;
Coleman et al., 1994; Schroeder, 1999; Basili, Briand, and Melo, 1996; Briand et al., 1998, 2000; Yu, Systa, and Muller, 2002; Subramanyam and Krishnan, 2003; and Gyimothy, Ferenc, and Siket, 2005). Furthermore, the survey by Malhotra and Chug (2016) identifies many maintainability analysis techniques based on these metrics that have been demonstrated to predict maintainability problems. In addition, this dissertation (subsection 6.6.2), presents some evidence for component and connector based maintainability techniques being suitable and effective for Java systems classified as large by Zhang and Tan (2007). EPRA has not been verified and so whether its suggested benefits are realised remains to be seen. However, even if EPRA’s benefits were proven, EPRA does not help with the goal of this dissertation because EPRA is neither Risk Outputting or Risk Mapping.

Sobhy et al. (2016) tested a CBAM specialisation based on real options theory that evaluates how diversity in architectural decisions can achieve sustainability. Their specialisation demonstrates that specific risks may need specific methods. This observation suggests that whilst the generic techniques discussed in this subsection (SAAM, ATAM, ARID, CBAM, and WinCBAM) may offer some capability for identifying maintainability risks, they are not optimised for maintainability, and therefore may be less effective than the more specific maintainability techniques discussed in the following subsections.

Whilst they could be used to produce maintainability risk views to some degree, all the techniques covered in this subsection lack automation and a consistent way to split the architecture into risk isolating containers. This means using them to create risk views will be labour intensive and the risk views produced may be coarse and inconsistent. In conjunction with Malhotra and Chug’s (2016) survey that identifies many techniques that automate maintainability risk analysis, these limitations suggest that SAAM, ATAM, ARID, CBAM, and WinCBAM are not the best approaches on which to base container based risk assessment due to the scale of modern architectural designs (Zhang and Tan, 2007).

### 2.1.4 Maintainability Risk Identification

Several maintainability specialisations of SAAM have been proposed: SAAM for Evolution and Reusability (SAAMER) by Lung et al. (1997), SAAM for Complex Scenarios (SAAMCS) by Lassing, Rijsenbrij, and van Vliet (1999), and integrating SAAM in domain-Centric and Reuse-based development (ISAAMCR) by Molter (1999). Lung et al.’s (1997) main contribution is to present a framework and set of architectural views for reusability analysis which would not identify the specific risks of interest to the research carried out in this dissertation. Lassing,
Rijsenbrij, and van Vliet (1999) criticised previous flexibility analysis methods for concentrating on the average number of changes incurred per scenario which have the potential to overlook complex change scenarios where the risk could be disproportionate. They proposed SAAMCS, a two-dimensional framework that extends SAAM to find those complex scenarios to counteract this problem, but they did not provide an evaluation which means it is unknown whether it could identify maintainability risks that could give rise to error-proneness or change propagation risks. Molter (1999) suggests that “the reduction of inherent risks is usually the prevalent motivation for performing architecture analysis”, p. 9. The purpose of Molter’s ISAAMCR is to identify flaws early in the life-cycle of reusable architectures, i.e. discover flexibility limitations that could introduce risks when the reusable architecture is applied to new scenarios. SAAM is extended by the introduction of reusability based protoscenarios to identify these flaws that could lead to reusability risks.

SAAMER, SAAMCS, and ISAAMCR support the Risk Output and Risk Mapping criteria by virtue of the same rationale that SAAM supports these criteria, i.e. Risk Output is supported because the technique can provide some indication of maintainability risk and Risk Mapping is supported because these techniques can be applied to a subset of the architecture and a risk view can be constructed by conducting many evaluations, one per subset of the architecture. However, they also inherit the same drawbacks for creating risk views: reliance on assessor skill, being labour intensive, and lack of reproducibility.

Genero et al. (2003) suggest greater size and structural complexity of software architectures makes software architectures more difficult to understand, analyse and modify due to greater cognitive complexity, which in turn causes complex and large architectures to be more difficult to maintain. Genero et al.’s suggestion is supported by the findings of Wolfe and Horowitz (2017) who reported that limits on human visual processing make it impossible to recognise everything at once and that objects of interest are harder to find in more complex search fields. To help architects deal with complexity, Genero et al. calculated complexity and size metrics from class diagrams. In addition, subjective measures were obtained by asking students to perform understandability and modifiability tasks based on the same class diagrams. The Kendall (1938) and Spearman (1904) coefficients were used to test the significance of the correlation between the metrics and subjective results to determine whether the metrics are good predictors of understandability and modifiability. Based on two independent experiments, Genero et al. concluded that NAssoc (total number of relationships in a class diagram) and maximum Depth of MaxDIT (maximum depth of inheritance tree in a class diagram) are most effective maintainability predictors of the metrics they tested. In the one experiment the
NAssoc/MaxDIT correlations were weak but significant to the $\alpha=0.05$ level, and in the other experiment they were strong and significant to the $\alpha=0.01$ level. Genero et al. suggest this difference was due to only one experiment being conducted using the English language.

Genero et al.’s method better supports the Risk Output criterion because a metric indicating relative risk is calculated and the Risk Mapping output criterion because each class diagram analysed is a subset of the architecture. However, there are three problems with Genero et al.’s method with respect to the research questions posed in this dissertation: firstly, whole class diagrams could be quite arbitrary, i.e. the same design could be split up into diagrams in different ways; secondly risk containers based on class diagrams could potentially be overlapping depending upon how much thought the architects gave to structuring the architecture into different class diagrams; thirdly, no evidence is provided about how the maintainability characteristic of the architecture translates into error-proneness and change-propagation; and fourthly, the study was conducted with students as opposed to practitioners. However, their results suggest that NAssoc and MaxDIT, or other metrics (McCabe 1976; Henry and Kafura, 1981; Chidamber and Kemerer, 1994; Mitchell and Mancoridis, 2006), should be considered when developing a method to test whether error-proneness and change propagation could be predicted from risk containers. Whilst Genero et al.’s approach improves upon SAAM, ATAM, ARID, CBAM, SAAMER, SAAMCS, and ISAAMCR by reducing the reliance on assessor skill and being less labour intensive through automation, basing risk containers on arbitrary class diagrams fails to address the lack of reproducibility.

Van Koten and Gray (2006) used a Bayesian network to predict software maintainability. Their technique could, in theory, be used to create architecture risk views because an overall object-oriented system could be broken down into subsystems, and the technique used to predict maintainability of each sub-system. Their approach correlates design metrics to lines of code changed over a 3-year maintenance period. Zhou and Leung (2007) used the same experimental construct, metrics, measure of maintainability (lines of code), and training projects as Van Koten and Gray (2006) to evaluate the performance of a multiple adaptive regression splines algorithm. Zhou and Leung concluded their method is at least as accurate, if not more accurate, than other multivariate linear regression models, artificial neural network models, regression tree models, and support vector models. Furthermore, Wang et al. (2009) and Dubey, Rana, and Dash (2012) evaluated additional algorithms called Project Pursuit Regression and Multilayer Perceptron respectively, again using the same experimental construct, metrics and software projects. Wang et al. and Dubey, Rana, and Dash also concluded their algorithms are effective at predicting maintainability when measured in terms of lines of code.
Ye, Zhu, and Wang (2013) furthered the application of machine learning algorithms to the problem of predicting software maintainability by combining multiple machine learning models in a technique known as Multiple Classifiers Combination. They evaluated their approach using an open-source object-oriented system composed of 300 C++ classes and 21 different software design metrics that are calculable from UML. Ye, Zhu, and Wang concluded that their method has “adequate” predictive capability, and that it is more effective than relying upon any single classifier.

Malhotra and Chug (2014) collected the old and new source code versions of two C# web applications to evaluate a novel machine learning method called Group Method of Data Handling by comparison to Feed Forward 3-Layer Back Propagation Network and General Regression Neural Network models. Again, design metrics were used to predict maintainability measure in terms lines of code changed between versions. Malhotra and Chug concluded that Group Method of Data Handling was more effective than the other two models, and that developers can use it to judge maintainability whilst designing and coding.

Kumar and Rath (2015) used a hybrid neural network and genetic algorithm (a Neuro-Genetic algorithm) based on design metrics to estimate the maintainability of the same ADA software products used by Van Koten and Gray (2006) and Zhou and Leung (2007). They argue that the incorporation of rough set analysis and principal component analysis improves prediction accuracy.

There are several problems with these approaches (Van Koten and Gray, 2006; Zhou and Leung, 2007; Wang et al., 2009; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013; Malhotra and Chug, 2014; Kumar and Rath, 2015). The first problem is that lines of code is a discredited measure of software developer output (Sadowski and Zimmermann, 2019, p. 13), and therefore of maintainability in this context. The second problem is that only two ADA projects (Van Koten and Gray, 2006; Zhou and Leung, 2007; Wang et al., 2009; Dubey, Rana, and Dash, 2012; Kumar and Rath, 2015), two web applications (Malhotra and Chug, 2014), or one C++ project (Ye, Zhu, and Wang, 2013) have been used to train their predictive models, i.e. there is no way to know if they would be as predictive if used for other software projects using different maintenance practices and different programming languages. Thirdly, even if they do work for other projects they will have to be retrained. Fourthly, whilst they could potentially be used to create risk views by performing separate analyses on different parts of an OO system using these approaches, there remains the question of how to best split the architecture into subsets to be used as risk containers. Fifthly, using machine learning models (Van Koten and
Gray, 2006; Dubey, Rana, and Dash, 2012; Malhotra and Chug, 2014; Kumar and Rath, 2015), and especially using multiple classifiers (Ye, Zhu, and Wang, 2013), abstracts the rationale behind the prediction from the architecture input. That means architects will not understand how to improve their design if it is predicted to be difficult to maintain using machine learning techniques because they offer no traceability back to the design metrics used to make the prediction.

Kaur, Kaur, and Pathak (2014) used a much greater number of metrics (25 in total) than previous machine learning studies (which typically used fewer than 10) and used multiple classifiers as per Ye, Zhu, and Wang’s proposition. The classifiers used by Kaur, Kaur, and Pathak were Naïve Bayes, Bayes Network, Logistic Regression, Multilayer Perceptron, and Random Forest. Again, maintainability was measured in terms of lines of code changed. Their evaluation was based on four open-source projects (Lucene, JHotdraw, JEdit, and JTreeview) which ranged in size between 60 and 359 classes and 11988 and 135241 lines of code. Kaur, Kaur, and Pathak’s precision and recall results demonstrated that they could create accurate prediction models for predicting which modules are more likely to be change-prone. Reliance on open-source projects fails to provide evidence about whether their approach would help industry practitioners. Kaur, Kaur, and Pathak’s approach has all the same drawbacks as the other studies that use lines of code changed as a measure of maintainability effort and machine learning (Van Koten and Gray, 2006; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013, Malhotra and Chug, 2014; Kumar and Rath, 2015).

Whilst these techniques have applied statistical analysis (Zhou and Leung, 2007; Wang et al., 2009) and machine-learning (Van Koten and Gray, 2006; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013, Malhotra and Chug, 2014; Kumar and Rath, 2015) to further optimise the use of metrics for predicting the maintainability characteristics of software, they fail to improve upon Genero et al.’s contribution as a basis for container based architecture risk assessment, i.e. basing risk containers on arbitrary object-oriented sub systems still fails to address lack of reproducibility.

Stevanetic and Zdun (2016) also calculated design metrics from UML to indicate the understandability of components. They hypothesised that the metrics would predict the amount of time needed to correctly understand components. Their hypothesis is consistent with the findings of Wolfe and Horowitz (2017) who reported that limits on human visual processing make it impossible to recognise everything at once and that objects of interest are harder to find in more complex search fields, i.e. more complex components should be more difficult to
understand. They advanced previous work by considering hierarchical component metrics (module size in classes, number of API classes, direct module coupling, number of disjoint clusters, cohesion by rest of world, and depth in module hierarchy). They concluded that hierarchical metrics are significantly better predictors of component understandability than previous metric models and models based solely on the participant’s level of experience. Their results show that if the internal relationships of a component are difficult to comprehend it might increase the risk associated with maintaining that component. Although class diagrams and components are different, Stevanetic and Zdun’s results are consistent with the findings of Genero et al. because they are both complex structures that represent some or all of the structural view of a software architecture. This means both studies suggest the internal complexity of complex composite structures affects the understandability of those structures. Stevanetic and Zdun’s technique is a good fit for the Risk Output and Risk Mapping criteria because a relative indicator of understandability risk (Risk Output) is reported for components which are specific types of architecture subsets. Thus, a risk view whereby each component is ranked by understandability risk could be produced by calculating the understandability of each component and ranking components by their relative maintainability risk. This makes Stevanetic and Zdun’s technique the nearest match to the risk container proposition discussed so far because basing risk containers on components addresses the lack or reproducibility limitation that approaches based on arbitrary architecture subsets have (ATAM, ARID, CBAM, SAAMER, SAAMCS, ISAAMCR, Genero et al., 2003; Van Koten and Gray, 2006; Zhou and Leung, 2007, Wang et al., 2009; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013, Malhotra and Chug, 2014; Kaur, Kaur, and Pathak 2014; Kumar and Rath, 2015). However, despite Stevanetic and Zdun’s results suggesting that components could be an effective risk container type, they offer no performance comparison to other types of architecture subset, i.e. there is no evidence to suggest using components as risk containers would be the most effective container type for maintainability risks. Furthermore, whilst Genero et al. (2003) and Stevanetic and Zdun’s (2016) research is supportive of the hypothesis that design complexity could lead to errors being introduced and changes inaccurately analysed because the design and/or code is not fully understood, neither presented any direct evidence for the isolation of error-proneness and change propagation in upfront designs.  

2.1.5 Error-proneness Risk Identification 

Thwin and Quah (2005) applied machine learning to the problem of predicting the number of defects classes will have. In a comparison between Ward and General Regression Neural
Networks using Chidamber and Kemerer’s (1994) object-oriented metrics, Thwin and Quah found the latter to provide marginally more accurate predictions. Thwin and Quah’s results showed correlation strength between predicted faults and actual faults is approximately $\rho=0.95$ with a significance of $\alpha<=0.001$ for both networks. Despite the promising potential, there are several issues with Thwin and Quah’s approach with respect to isolating error-proneness risks. Firstly, the input to their experiment was three human machine interface subsystems, so it remains unproven whether their approach is effective for other types of software. Secondly, subsystems is a vague term and it is not reported whether they overlap which means it is unknown as to whether the faults predicted are really isolated in the subsystems, i.e. faults could be double counted. Furthermore, the Thwin and Quah technique suffers from the same drawbacks as the other machine learning techniques already discussed (Van Koten and Gray, 2006; Zhou and Leung, 2007; Wang et al., 2009; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013, Malhotra and Chug, 2014; Kaur, Kaur, and Pathak 2014; Kumar and Rath, 2015).

Xiao, Cai, and Kazman (2014) investigated whether Design Rule Spaces (DRSpaces) could be used to glean architectural insight. Xiao, Cai, and Kazman extracted DRSpaces from source code using Wong et al.’s (2009) Design Rule Hierarchy algorithm. Each DRSpace is based on a design rule which are the key interfaces that split an architecture into independent modules. The authors explain that each DRSpace is a graph whose vertices are the related classes the DRSpace is composed of and the edges are the relationships between those related classes. It is therefore possible for DRSpaces to overlap because a DRSpace may depend on another DRSpace and in such a case the leading class (design rule class) will be in both spaces. DRSpaces are similar to the Concern Graphs proposed by Robillard and Murphy (2002). The difference being that DRSpaces strictly model the classes that realise a modularising design rule, whereas concern graphs can represent any concern, even a cross-cutting concern such as logging, and they don’t have to be related to modularisation. A tool called TITAN was used to develop the DRSpaces from the source code of large open source projects: JBoss, Hadoop Common, and the Eclipse IDE. Xiao, Cai, and Kazman’s method used error and change proneness of files to indicate which DRSpaces contain architectural problems. Each file with at least $N$ bugs or changes is included in a bug/change space respectively. DRSpaces are then ranked in terms of error/change proneness by expressing the percentage of the bug/change space they occupied. Three conclusions were drawn by Xiao, Cai, and Kazman (2014) as follows. Firstly, if a leading file of a DRSpace is error-prone, a large proportion of the other DRSpace files are likely to be error-prone. Secondly, most error-prone files will be found in just a few DRSpaces. Thirdly, all error-prone DRSpaces exhibit multiple structural and evolutionary issues. Upon deeper analysis in a
follow up paper by Mo et al. (2015), the researchers identified two specific common causes of the error-proneness to be unstable interfaces and implicit cross module dependencies.

DRSpaces are a good fit for both the Risk Output and Risk Mapping criteria because relative error-proneness (Risk Output) is attributed to DRSpaces which are architecture subsets based on modularising design rules (Risk Mapping) which are created algorithmically and therefore reproducibly. Furthermore, they offer an alternative to using components as risk containers based on Stevanetic and Zdun’s (2016) research. However, in terms of this dissertation’s goal, the work of this research group has some limitations. Firstly, whilst Xiao, Cai, and Kazman have shown how DRSpaces can be effective in locating existing error-proneness in implementations, they have not established whether calculating metrics known to predict maintainability problems (McCabe 1976; Henry and Kafura, 1981; Chidamber and Kemerer, 1994; Mitchell and Mancoridis, 2006) would enable DRSpaces to be used to mitigate such problems during design, i.e. DRSpaces are dependent on static analysis of source code and the goal of this dissertation is to identify risks in designs. Secondly, much of the insight reveals evolutionary coupling issues which are not related to the upfront design, and which cannot be predicted before implementation. Unless of course the root cause of such evolutionary coupling and implicit cross module dependencies stems from limitations of the architectural design. Overall, DRSpaces are a close fit to the risk container concept and hence used as the basis of Design Rule Containers for further evaluation as a risk container type in this dissertation. Arguably DRSpaces, as well as the Design Rule, Use Case, and Resource Containers evaluated in this dissertation, are more specialised forms of Robillard and Murphy’s (2002) Concern Graphs.

2.1.6 Change Propagation Risk Identification

Architecture Level Modifiability Analysis (ALMA) was proposed by Bengtsson et al. (2004) and has five main steps: goal selection, software architecture description, change scenario elicitation, change scenario evaluation, and interpretation. Inside the goal selection step is a risk assessment sub step to identify the types of change for which the architecture is inflexible. ALMA was conceived to be used for one of three goals: maintenance prediction, risk assessment, and software architecture comparison. The authors suggest that for risk assessment, the architectural description must contain information to determine whether a change scenario will be complex to implement. They identify four criteria for identifying change scenario complexity, which are: initiator of the changes, impact level, multiple owners (negotiation may increase risk), and version conflicts. They suggest that changes to the environment in which the system
operates are also important to risk assessment and hence should be included in the architectural description.

Not only is ALMA more subjective than metric-based techniques (Genaro et al., 2003; Stevanetic and Zdun 2016; Abdelmoez et al., 2005), because it relies on interviews of the designers to gather the change scenarios, it can only deal with known business change scenarios. In addition to ALMA being Risk Outputting by design it could satisfy the Risk Mapping criterion by performing several ALMA evaluations on distinct parts of an architecture to produce a map of change complexity risk. However, architects would have to decide how to split up the architecture in order to use ALMA to create effective risks containers, which is a limitation by comparison to using Xiao, Cai, and Kazman’s (2014) DRSpaces and components as Stevanetic and Zdun (2016) did. Furthermore, whilst modifiability is a close fit for the goal of the research presented in this dissertation, ALMA does not specifically isolate change propagation risks into architecture subsets.

Unlike ALMA, Abdelmoez et al.’s (2005) does not rely on subjective change scenarios because it uses change propagation probability calculations based on component dependencies to determine change propagation risk. The calculations are based on the usage of public attributes and function (method) parameters of class A by class B. Shaik et al. (2006) reported that Abdelmoez et al.’s change propagation metric of is “helpful and effective in assessing the design quality of software architectures”. Abdelmoez et al.’s technique satisfies the Risk Output and Risk Mapping criteria because a relative measure of change propagation probability is output per component thus producing a risk view whereby each component ranked by risk. Therefore, like the approaches proposed by Xiao, Cai, and Kazman (2014) and Stevanetic and Zdun (2016), the Abdelmoez et al. approach is a close fit to the risk container proposition.

Abdelmoez, Goseva-Popstojanova, and Ammar (2006) used successive system releases to estimate requirement maturity in conjunction with the change propagation metric proposed in Abdelmoez et al. (2005). The requirement maturity is traced to components to determine the Initial Change Probability of components (ICP). The ICP is then multiplied by component change propagation probabilities to give an Unconditional Change Probability (UCP). The UCP is then multiplied by an estimate of size of change to determine the maintainability risk. The improved method underwent limited testing using a case study by introducing a theoretical change into a sequence diagram. However, the reliance on multiple system releases to determine requirement stability means the enhanced technique is not suitable for design time risk analysis.
Shaik, Abdelmoez, and Ammar (2008) have furthered the work of Goseva-Popstojanova et al. (2003) and Abdelmoez et al. (2005, 2006) by automating the methods they proposed to develop the Software Architecture Risk Assessment (SARA) tool. Change propagation probability estimates are calculated using the methods described by Abdelmoez, Goseva-Popstojanova, and Ammar (2006). The tool assumes all files have an initial change probability of one. However, the tool also allows the user to input a source code error report alongside the UML that can be used to calibrate change probabilities the tool uses. The tool also calculates error propagation probability based on the methods described by Goseva-Popstojanova et al. (2003).

Automation overcomes a major limitation in the application of their work to industry as it enables software architects to use their methods without needing special training outside of their core discipline. Furthermore, it means the results will be more reproducible and allows architects to focus on analysing the output of SARA rather than using their time to calculate component change propagation probabilities. Architects only need to learn how to use the tool which takes input in the form of a UML model and outputs the results in a Graphical User Interface (GUI). The SARA tool requires comprehensive UML sequence, class, and state chart diagrams, which may not always be available, especially in agile projects that favour working software over comprehensive documentation. The reliance on state charts is a major limitation for use in practice because Petre (2013) found that less than 30% of UML users work with state chart diagrams. In terms of output, the SARA GUI displays the relative change and propagation probabilities between pairs of components which allows the architect to compare components for relative risk (i.e., a component based risk view). Notwithstanding that SARA needs an optional source code input, SARA is the closest match to a technique that can already create risk containers. However, whether components really isolate change propagation risks is unknown because components may share classes and thus components may not be the most effective way of entrapping change propagation risk.

Rostami et al. (2015) attempt to improve change propagation prediction accuracy 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 which means that Rostami et al.’s approach does not support the goal of this dissertation.

More recently Weifeng et al. (2019) have used simulation to predict the change propagation characteristics of software. A class coupling network (graph) is used to model software structure before simulations are applied over the network to calculate their novel metric termed Software Stability. Software Stability is the average number of classes not changed in the change simulation scenarios, thus the higher the Software Stability, the fewer classes have to be modified to implement changes. Weifeng et al. theoretically validated their technique using Weyuker’s (1998) properties and empirically validated it using a set of open-source Java software systems. Weifeng et al. found that most of Weyuker’s properties are satisfied and that the Software Stability metric is an effective indicator for software quality improvement and class importance. Knowing the Software Stability for a software project’s class coupling network does not support the goal of isolating which parts of an architecture contain more change propagation than others. Nor does being able to determine which classes are more important across the different change simulations (god classes). Furthermore, there is also the question of whether the simulations used to exercise the class coupling network are representative of the change scenarios that would occur in practice. However, if Weifeng et al.’s technique were applied to clustered subsets of an architecture (e.g. clustered using Wong et al.’s, 2009, Design Rule Hierarchy Algorithm), it might be possible to create an architecture risk view by ranking the clusters by the Software Stability Metric, however that remains to be proven.

Li et al. (2019) have also considered trying to identify the areas of change propagation risk through identification of the most influential functional modules. Li et al.’s technique uses the weighted LeaderRank algorithm and susceptible-infected-recovered model of weighted and directed complex networks to identify the influential function modules of modular systems at the design stage. Li et al.’s technique is not specific to software and further evaluation is needed to determine whether their approach could be used to predict influential modules within a software architecture. It also suffers from the same problem as Weifeng et al.’s because knowing the god classes does not provide a risk view of the whole architecture. Once again, it would have to be used in combination with a clustering algorithm to produce a risk view.

2.1.7 Section Summary

The goal of section 2.1 was to find existing techniques that can identify maintainability risks in architectural designs to see whether they can isolate error-proneness and change propagation
risks into architecture subsets that could support the risk container concept. The search method introduced in sub-section 2.1.2 was based on Okoli’s (2015) systematic method and resulted in 268 papers for consideration. Of those found, 25 were selected for detailed review by applying the search criteria defined in sub-section 2.1.1. Of the 25 that satisfied the selection criteria, only the techniques proposed by Thwin and Quah (2005) and Xiao, Cai, and Kazman (2014) provided evidence that their methods could support the concept of error-proneness risk containers. Only Abdelmoez et al. (2005, 2006) and Shaik, Abdelmoez, and Ammar (2008) provided evidence that their methods could support the concept of change propagation risk containers.

2.2 Software Metrics

Having reviewed techniques that can be used to identify risks in software architectures to identify where the gaps lie with respect to producing risk views of software architectures, we must next consider what metrics can be used to indicate error-proneness and change propagation risks to denote which architecture subsets are riskier than others. That is because any metrics used in this dissertation should be based on standard metrics that are already known to indicate error-proneness and change propagation rather than reinventing the wheel if possible. In other words, risk container metrics should build upon the body of knowledge because software engineering maintainability metrics is a mature field as shown in this section.

2.2.1 Error-proneness Indicators

The concepts of coupling, cohesion, and complexity all have a bearing on how easy or difficult software is to develop and maintain due to limits on human visual processing (Wolfe and Horowitz, 2017). Low coupling between software components means changes are more self-contained within software components due to the absence of the ripple effect (Lindvall, Tvedt, and Costa, 2003; Bass, Clements, and Kazman, 2012). As explained in section 1.1, failure to pre-empt the ripple effect by developers can lead to errors because changes are not always fully implemented. High cohesion within software components means the component better obeys the single responsibility principle. The single responsibility principle proposed by Martin (2002) asserts that each class should have only one reason to change, or in other words separate concerns should not be conflated into one class by coupling different responsibilities together. Keeping them separate means a change to one responsibility cannot have an impact (e.g., introduce an error by mistake) on the other which means the systems that follow the single responsibility principle are more maintainable overall. Complexity can result from high coupling
and low cohesion. Many standard software engineering metrics to assess the relative coupling, cohesion, and complexity of software have been proposed and previous research has investigated how effective they are at predicting error-proneness.

2.2.1.1 Coupling

Coupling Between Objects (CBO) proposed by Chidamber and Kemerer (1994) is a measure of how many other classes a class is directly dependent upon in order to fulfil its own responsibilities. Research by Basili, Briand, and Melo (1996), Briand et al. (1998; 2000), Yu, Systa, and Muller (2002), Subramanyam and Krishnan (2003), and Gyimothy, Ferenc, and Siket (2005) suggests CBO is a predictor of error-proneness. Gyimothy, Ferenc, and Siket observed that a range of other metrics (Weighted Methods per Class, Lines of Code, Request for Class, Lack of Cohesion of Methods, and Fan In) are significant in their ability to predict error-proneness but they concluded their study by stating that "The CBO metric seems to be the best in predicting the error-proneness of classes", p. 909. Further evidence is provided by Thwin and Quah (2005) who found that using Chidamber and Kemerer’s metrics suite, including CBO, in combination with machine learning could make accurate predictions about the number of defects classes will have. It should however be noted that other research has failed to demonstrate a correlation between coupling and error-proneness (Binkley and Schach, 1998; Harrison, Counsell, and Nithi, 1998; Tang, Kao, and Chen, 1999). Another criticism of CBO as a predictor of error-proneness is that El Emam et al. (2001) demonstrated a confounding effect of class size. El Eman et al. demonstrated how stratifying by class size reduces the association between coupling and error-proneness. Furthermore, they highlight studies (Briand et al., 2000) that have observed associations between coupling metrics and size, and other studies (Hatton, 1996; Cartwright and Shepperd, 2000) that observed associations between size and error-proneness directly. These studies support El Eman et al.’s conclusion that previous validation studies have over-estimated the ability of object-oriented metrics such CBO to predict error-proneness. El Eman et al. therefore recommend that future studies control for the effect of size when using CBO to predict error-proneness.

Henry and Kafura (1981) first defined Fan In (FI) when considering metrics for information flow as: fan-in of procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information. Since the emergence of object-oriented software development FI has been adapted to be an alternative coupling metric to CBO. For example, more recent research by Mubarak, Counsell, and Hierons (2010) defines FI the incoming coupling to a class, i.e. it is the opposite of CBO. Whereas CBO is a count of the
number of other classes a class is dependent upon to fulfil its own responsibilities, $FI$ is a count of other classes that are dependent upon a given class to fulfil their responsibilities. In their comparative study of different metrics for predicting error-proneness, Gyimothy, Ferenc, and Siket (2005) suggested $FI$ is predictive of error-proneness, but that it is not as effective at predicting error-proneness as CBO is.

Mo et al. (2016) proposed the Decoupling Level (DL) metric to assess how decoupled architectural modules are from each other. Their DL research extends Wong et al.’s (2009) research with the Design Rule Hierarchy (DRH) clustering algorithm. The overall DL for an architecture is the sum of the DLs for each layer in the DRH. As specified, DL does not support the research goal because its formula is based on source code files rather than design diagrams. DL also does not support the 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. No direct evidence of whether DL can be used to predict error-proneness has been provided.

### 2.2.1.2 Cohesion

Lack of Cohesion in Methods (LCOM) proposed by Chidamber and Kemerer (1994) represents the cohesiveness of a class as the degree of similarity between the methods of a class, i.e. if the methods of a class are all considered to be similar because they relate to the same responsibility, the class is cohesive. Cohesiveness is determined by the number of methods that access the same instance variables. Both Yu, Systa, and Muller (2002) and Gyimothy, Ferenc, and Siket (2005) presented evidence for LCOM being predictive of error-proneness, albeit based on different methods of calculation. Basili, Briand, and Melo (1996) however found LCOM not to be predictive of error-proneness and Subramanyam and Krishnan (2003) didn’t test it. LCOM is unlikely to be useful for analysis methods applied to upfront designs because UML class diagrams do not detail which instance variables are used by different methods unless they are annotated with Object Constraint Language (Object Management Group, 2014), and state machine diagrams are rarely used (Petre, 2013).

Mitchell and Mancoridis’ (2006) Cluster Factor (CF) is an alternative cohesion measure. Whereas LCOM represents the cohesion of a class, CF represents the cohesion between a collection of classes that comprise a module. CF is equal to the number of internal dependencies within a module divided by the number of internal dependencies plus half the external dependencies. Internal dependencies are between two classes of the module, whereas external
dependencies are from a module class to a class that is not part of the module. CF could be calculated for risk containers by considering a risk container to be interchangeable with a module when calculating CF. Given that CF is a measure of module cohesion, and that Xiao, Cai, and Kazman’s (2014) DRSpaces use the DRH algorithm to separate an architecture into independent modules, Xiao, Cai, and Kazman missed an opportunity to evaluate DRH and DRSpaces with CF. That is because if DRH separates an architecture into independent modules as intended, it would be expected that the modules extracted should have high CF.

2.2.1.3 Complexity

McCabe (1976) proposed a software complexity measure based on cyclomatic number or a graph: “In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits”, p. 308. McCabe adapted the cyclomatic number to be a software metric, McCabe’s Cyclomatic Complexity (MCC), by defining how standard software control structures (sequence, if then else, while and until) are represented in a graph. Doing so enables the transformation of program code into a graph, which in turn enables the cyclomatic number of the transformed graph to be calculated. The complexity of a program is considered to be the cyclomatic number of the graph the program code has been transformed into. Research by Munson and Khoshgoftaar (1992), Coleman et al. (1994), and Schroeder (1999) is supportive of using MCC to predict error-proneness. As well as a correlation to error-proneness, studies have also demonstrated that programs with greater complexity have more lines of code. Hatton (2008) suggested that MCC is no more effective than lines of code as a predictor or error-proneness. Thus, similar to the way class size in terms of lines of code confounds CBO, lines of code also confounds complexity because larger programs are more complex and therefore likely to have more defects. The goal of the research presented in this dissertation conflicts with the way MCC is calculated. That is because upfront architectural designs rarely specify code structures (except perhaps UML state machine diagrams) and so the control structure information required to calculate MCC is unlikely to be available. Petre (2013) found that only 3 of 11 UML users were using state machine diagrams from which MCC could be calculated. MCC is therefore limited in practical terms to the analysis of program code and had be adapted for use in the kinds of upfront designs that are typically available for use in this dissertation.

Chidamber and Kememer (1994) proposed an alternative complexity metric for Object-Oriented (OO) software called Weighted Methods per Class (WMC). The complexity of each method must be calculated to calculate the WMC of a class. Chidamber and Kemerer elected not to prescribe how to calculate method complexity, leaving method complexity open to
interpretation. For example, it could be method lines of code, number of parameters, or number of class types referenced in the method signature. WMC is calculated by summing all the method complexities together. Basili, Briand, and Melo (1996), Yu, Systa, and Muller (2002), Subramanyam and Krishnan (2003), and Gyimothy, Ferenc, and Siket (2005) all demonstrated WMC can be predictive of error-proneness. Using WMC may not be helpful for risk containers extracted from designs because not all class diagrams will include all of the methods, i.e. WMC requires more detailed class diagrams than CBO for example and therefore may be less useful in practice.

Response for a Class (RFC) proposed by Chidamber and Kemerer (1994) is also an alternative complexity metric. RFC is the set of methods that can potentially be executed in response to a message received by an object of that class. Chidamber and Kemerer explained that “the larger the number of methods that can be invoked from a class, the greater the complexity of the class”, p. 487. Basili, Briand, and Melo (1996), Yu, Systa, and Muller (2002), and Gyimothy, Ferenc, and Siket (2005) all concluded that RFC is predictive of error-proneness. Subramanyam and Krishnan (2003) did not test RFC. Calculating RFC from designs requires sequence diagrams which are used in 55% of UML designs (Petre, 2013). If sequence diagrams provide comprehensive coverage of the design, RFC could therefore be used in combination with risk containers to analyse designs for error-proneness risks.

Chidamber and Kemerer (1994) also proposed the Depth of Inheritance Tree (DIT) metric which they suggest can also be interpreted as a complexity metric in OO software. They state, “the deeper the class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour”, and that, “deeper trees constitute greater design complexity, since more methods and classes are involved”, p. 483. Gyimothy, Ferenc, and Siket (2005) observed that DIT was not as predictive of error-proneness as the other metrics they observed to be predictive of error-proneness (CBO, WMC, Lines of Code, Request for Class, LCOM, and Fi). Gyimothy, Ferenc, and Siket also concluded that whilst DIT is predictive of error-proneness it is not as trustworthy as the other metrics that predict error-proneness. That is because the completeness of DIT, i.e. the number of faults in faulty predicted classes divided by the number of faults in all classes, was lower than the completeness of the other metrics that predict error-proneness. Basili, Briand, and Melo (1996) also found DIT to be predictive of error-proneness, however the results obtained by Subramanyam and Krishnan (2003) are contradictory because they observed DIT to be insignificant with respect to predicting error-proneness. Whilst DIT could be calculated from design diagrams, relying on it as a predictor of error-proneness would limit the applicability of risk containers to object-oriented software.
Number of Children (NOC) is the number of immediate sub-classes of a class (Chidamber and Kemerer, 1994). NOC cannot be neatly categorised as a coupling, cohesion, or complexity metric because its theoretical basis encompasses all three properties: coupling because it is a specialised version of FI whereby the incoming dependencies are the inheritance relationships; cohesion because a large NOC increases the likelihood of an improper (and incohesive) abstraction of the parent class; and complexity because NOC indicates the potential influence a class has on a design. Gyimothy, Ferenc, and Siket (2005) concluded that NOC is not predictive of error-proneness, whereas Basili, Briand, and Melo (1996) concluded that higher NOC was associated with lower fault detection even though Yu, Systa, and Muller (2002) concluded that the higher the NOC of a class the more faults it will have. Here the results of all three studies are contradictory which suggests NOC is not a reliable predictor of error-proneness. Subramanyam and Krishnan (2003) didn’t test NOC. Again, like DIT, relying on NOC would limit the applicability of risk containers to object-oriented software.

Thwin and Quah (2005) used WMC, RFC, DIT, and NOC (as well as CBO) with machine learning and found that combining machine learning with Chidamber and Kemerer’s (2004) metrics suite was effective for predicting the error-proneness of classes. This provides additional confidence that these metrics could be used with risk containers for isolating error-proneness in designs.

### 2.2.2 Change propagation indicators

We have already reviewed Weifeng et al.’s (2019) Software Stability metric in subsection 2.1.6 and discounted it from use in this dissertation because it indicates the overall stability of software rather than change propagation between classes. This subsection discusses other indicators found in the literature that can be used to predict change propagation risks.

The approach taken to calculate change propagation probability by Clarkson, Simons, and Eckert (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 risk of change propagating to b, starting at a, is shown in Equation 2-1.

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

In Equation 2-1 $\rho(b,u)$ is 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 $b$. The initial probabilities for change propagating between two components are derived from observed data. This method conflicts with the goal of design-time risk assessment because data on observed co-change is not available prior to implementation. Furthermore, it remains to be demonstrated as being effective in software architectures.

Abdelmoez et al.'s (2005) approach calculates change propagation probability based on the usage of public attributes and function parameters of class $A$ by class $B$. Their formula for the risk of change propagating to $B$, starting at $A$, is shown in Equation 2-2.

$$CP(A, B) = \Pr \left( (\lnot [B] \neq [B']) | (IA, B \neq IA', B') \land ([A \cdot B] = [A' \cdot B']) \right)$$


$IA, B$ in Equation 2-2 is the interface between $A$ and $B$. In this case, change propagation is assessed to a higher fidelity than the cruder overall component level dependencies used by Clarkson, Simons, and Eckert because the probability calculated takes into account the specifics of class $A$ used by class $B$ rather than assuming that when $A$ changes $B$ will likely change. The usage coefficient is $1$ or $0$ between each variable of component $A$ and every other component in the system depending on whether it is used or not. Therefore, the one step change propagation between any two components is estimated using the formula shown in Equation 2-3.

$$CP(C_i, C_j) = \frac{1}{|V_i|} \sum_{v \in V_i} \pi_{vj}$$

Equation 2-3 Abdelmoez et al.'s (2005) One Step Change Propagation between two Components

Where $V_i$ in Equation 2-3 is the set of public attributes and function parameters of component $C_i$ and $\pi_{vj}$ is the variable usage of each member of the set by the other component. Abdelmoez et al. explain that the one step change propagation probability sets an upper bound to the multi-step propagation. The trade-off for the increased fidelity of the Abdelmoez et al. calculation is the need for more detailed architecture input. Whereas the Clarkson, Simons, and Eckert calculation could be applied to dependencies extracted from a class diagram, the Abdelmoez et al. would require a diagram such as an object sequence diagram to show aspects of the interface being used.
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 the one based on the call graphs (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.

2.3 Modelling Risks

The ISO 42010 (2011) standard for architectural descriptions is limited in its support for risk management. The concept of risk is buried as an optional sub type of the Concern entity, and no further guidance about how to describe risks to support effective management is provided. In ISO 42010 a concern is defined as follows:

“a concern could be manifested in many forms, such as in relation to one or more stakeholder needs, goals, expectations, responsibilities, requirements, design constraints, assumptions, dependencies, quality attributes, architecture decisions, risks or other issues pertaining to the system” (p. 6).

Having risk buried inside the Concern concept does not reinforce the need for architects to prioritise risk management as Poort and van Vliet (2012) proposed in their Risk and Cost Driven Architecture approach.

Patterson and Neailey (2002) identified five stages of iterative risk management, namely: identification, assessment, analysis, reduction/mitigation, and monitoring. They derived their risk management model from the automotive industry, but it is representative of general project management practice.

Identification aims to elicit the risks associated with completing a project. ISO 42010 limits the identification step because risks cannot be identified in terms of the elements in the architecture that cause the risks.

Assessment aims to establish the overall probability of occurrence and impact. Analysis is optional and applies quantitative techniques such as Monte Carlo simulation, sensitivity analysis, and decision tree analysis to determine whether the risk level varies due to different project
choices. Together, the assessment and analysis stages prioritise risks for the reduction/mitigation step. Assessment and analysis are constrained by ISO 42010 because the impact in terms of how much of the architecture would be affected, and probability of occurrence of one risk, cannot be compared to other risks. For example, suppose a stakeholder was concerned about system maintainability; there is no way to record maintainability metrics for architecture elements to indicate the relative risk associated with them. This means the risk assessor cannot easily assess the relative impact of different risks and prioritise risks for the reduction/mitigation step.

Not knowing which parts of the architecture cause, or are impacted by risks, also constraints the reduction/mitigation stage. For example, suppose error-proneness risks were to be mitigated by the prioritisation of testing resources. If the risk description does not include the architectural description (AD) elements predicted to be error-prone, there is no easy way to prioritise subsets of the architecture for testing. Similarly, not knowing which AD elements are affected by the error-proneness, prevents the cost of the mitigation being compared to the importance of the use cases those AD elements support. Therefore, the absence of this information prevents a cost benefit analysis being performed for potential risk mitigations. Together, the above limitations demonstrate that ISO 42010 offers limited support for risk management.

ISO 31000 is the international standard for risk management and was first published in 2009 after Patterson and Neailey (2002) proposed their iterative risk management process. ISO 31000 (2018) is the most recent version of the standard. ISO 31000 is in agreement with Patterson and Neailey that risk management should be an iterative process. ISO 31000 includes several principles that further strengthen the argument for ISO 42010 being inadequate from a risk management point of view. Those principles are that risk management should be: a) integrated; b) structured and comprehensible; c) customised; d) inclusive; e) dynamic; f) use the best available information; g) consider human and cultural factors; and h) involve continual improvement. The principle that risk management should be integrated (a) as part of an organisation’s activities suggests that risk management should be integral to architecting teams and organisations as advocated by Poort and van Vliet (2012). The principles that risk management should be structured and comprehensible (b) and based on the best available information (f) suggests that architectural descriptions should be analysed to identify risks and that identified risks should be traceable back to their precursors in the architectural description. Thus, principles a, b, and f all support the idea that architecting should involve risk management and that architectural descriptions should include risk concepts. Furthermore, ISO 31000 defines
several concepts that ought to form the basis of any attempt to propose a standard risk model. Those concepts are risk, risk management, stakeholder, risk source, event, consequence, likelihood, and control.

The idea of modelling risks is not new, and Fabian et al. (2010) explain common criteria for modelling security risks. These common criteria are now formalised into ISO 15408-1 (2009) and are used as the basis of Fabian et al.’s conceptual framework for security requirements engineering. As illustrated in Figure 2-1, Stakeholders are subject to Risks and Risks constitute Potential Losses, but Risks can be reduced by Countermeasures. Fabian et al.’s model achieves traceability to the architectural design because Security Property is an abstract entity which has a sub class called Design Property (and Implementation Property) as shown in Figure 2-1. Violation of security properties implies a Potential Loss will be suffered by the Stakeholder. Security properties are potentially violated by Vulnerabilities, potentially exploited by Threats, and actually exploited by Attacks.

![Figure 2-1 Security Concepts Model (adapted from Fabian et al., 2010, Fig. 2)](image)

More recently Mayer et al. (2019) proposed how an information system security risk model could be integrated with enterprise architecture management. Their model shown in Figure 2-2 contains similar concepts to that of Fabian et al.’s model shown in Figure 2-1. Risk and Vulnerability are the same in both models. Mayer et al. further decompose Fabian et al.’s Countermeasure into Risk Treatment, Security Requirement, and Control. Furthermore, Potential Loss is replaced with a more generic Impact concept. Perhaps the most important change in Mayer et al.’s model is the recognition that for a security risk to be present, a Threat must interact with a Vulnerability, i.e. vulnerabilities are inconsequential unless they are threatened. Therefore, Mayer et al. introduce a new entity called Event that leads to an Impact.
when a Threat exploits a Vulnerability. Mayer et al.’s model achieves traceability from risks to architecture elements through an abstract entity called Asset because Application Element and Technology Element are sub types of Asset.

![Figure 2-2 Risk Model (abridged from Mayer et al., 2019, Fig. 6)](image)

### 2.4 The Research Gap

It is clear from the preceding sections that architecture analysis is a mature field with many techniques proposed. Mature both in terms of metrics for indicating relative areas of error-proneness and change propagation risk, and techniques that can attribute risks to different architecture subsets. Therefore, with respect to goals, scope, and research questions of this work, we must next consider where the knowledge gap lies.

Section 2.2 explained the set of standard software engineering metrics that are associated with indicating error-proneness and change propagation. As these metrics can be used to answer RQ1 there is no need to investigate the design of new metrics. However, the standard metrics have been invented for calculating the error-proneness and change propagation associated with classes and components as opposed to risk containers. Therefore, the work in this dissertation reports novel results by demonstrating how to calculate these standard metrics for Design Rule, Resource, and Use Case Containers (contributions C3, C5, and C7).

To determine the existing techniques for producing relevant risk views and find the nearest matches to the proposition of this thesis, the list of techniques that predict error-proneness and change propagation needs to be intersected with the list of techniques that can isolate risks into architecture subsets. Whilst some of the techniques reviewed in section 2.1 can be used to produce general maintainability risk views (SAAM; SAAMER; SAAMCS; ISAAMCR; ATAM; ARID;
CBAM; Genero et al., 2003; van Koten and Gray, 2006; Zhou and Leung , 2007; Wang et al., 2009; Dubey, Rana, and Dash, 2012; Ye, Zhu, and Wang, 2013; Kaur, Kaur, and Pathak, 2014; Malhotra and Chug, 2014; Kumar and Rath, 2015; Stevanetic and Zdun, 2016) there is no specific evidence that they can be used to predict error-proneness and change propagation risks. Bengtsson et al.’s (2004) ALMA technique is a close fit for the scope of this dissertation because it could support the production of change scenario complexity risk views based on arbitrary architecture subsets. However, identifying areas of change complexity is not the same as identifying the risk of change propagation which is the subject of this work.

Only Thwin and Quah (2005) and Xiao, Cai, and Kazman (2014) provided evidence that error-proneness risks could be attributed to object-oriented systems and DRSpaces respectively. However, the input to Thwin and Quah’s method is an object-oriented subsystem which does not support the risk container concept because the architect would not know how to reproducibly split the architecture into risk containers. Xiao, Cai, and Kazman’s approach addresses this limitation by splitting up architectures into independent modules using the Design Rule Hierarchy algorithm (Wong et al., 2009), but unlike Thwin and Quah’s approach it does not predict error-proneness from designs. Therefore, what is needed to support the risk container proposition is a technique that is predictive of error-proneness like Thwin and Quah’s approach, based on architecture subsets that can be repeatedly created like Xiao, Cai, and Kazman’s DRSpaces. Thus, part of the novelty of the research in this dissertation is evaluating whether calculating design metrics for Design Rule Containers can bridge this gap in order to create error-proneness risk views from architectural designs.

Only Abdelmoez et al. (2005, 2006) and Shaik, Abdelmoez, and Ammar (2008) provided evidence that change propagation could be attributed to components. Abdelmoez et al.’s (2005) metric can predict change propagation and components are architecture subsets and so it could be argued that their method already supports the risk container proposition. In addition, Shaik, Abdelmoez, and Ammar’s (2008) SARA already provides a GUI tool to identify and produce component change propagation risk views based on Abdelmoez et al.’s change propagation metric. However, UML components can share classes and so there is no guarantee that the risk of change propagation is isolated within components. It will depend upon how cohesive the components are and what level of class sharing is involved. Therefore, there might be more effective ways to isolate change propagation into risk containers. Hence, another part of the novelty of the research in this dissertation is evaluating the effectiveness of different types of risk containers (Design Rule, Resource, and Use Case Containers).
Overall, the closest matches to the research in this dissertation are considered to be Xiao, Cai, and Kazman’s (2014) DRSpaces for error-proneness, albeit with the limitation that they are not predictive, and Shaik, Abdelmoez, and Ammar’s (2008) SARA, albeit with the limitation that it is not known how well components isolate change propagation due to class sharing (overlap). Although DRSpaces can overlap, the degree of overlap should be low because the Design Rule Hierarchy algorithm on which they are based was designed to cluster classes into independent modules to increase developer parallelism (Wong et al., 2009). Table 2-2 provides a comparison of the research presented in this dissertation to the nearest matches.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>SARA</th>
<th>DRSpaces</th>
<th>This Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earliest Life-Cycle stage</td>
<td>Design</td>
<td>Implementation</td>
<td>Design</td>
</tr>
<tr>
<td>Architecture Input</td>
<td>UML sequence, class, and state chart diagrams</td>
<td>Implementation source code</td>
<td>Class, resource, and use case dependency graphs</td>
</tr>
<tr>
<td>Architecture Perspectives</td>
<td>Structure and behaviour</td>
<td>Structure</td>
<td>Structure, behaviour, and resource usage</td>
</tr>
<tr>
<td>Risk Types Supported</td>
<td>Change propagation</td>
<td>Error-proneness</td>
<td>Error-proneness and change propagation</td>
</tr>
<tr>
<td>Risk Attribution (Containers)</td>
<td>Components</td>
<td>DRSpaces</td>
<td>Design Rule, Resource, and Use Case Containers</td>
</tr>
<tr>
<td>Risk Predicting Evidence</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Element Isolating Evidence</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

DRSpaces were the inspiration behind the risk container concept evaluated in this dissertation. That is because Xiao et al. showed that DRSpaces can be used to split an architecture to separate areas of relatively high error-proneness risk from areas of relatively low error-proneness risk, where the risk is isolated within the bounds of the riskier DRSpaces. Likewise, Shaik, Abdelmoez, and Ammar demonstrated riskier components can be identified using SARA. Therefore, both DRSpaces and the components used in SARA satisfy some aspects of the risk container definition in section 1.3 because they are architecture subsets in which risks can be isolated. So, if both DRSpaces and components are close fits to being considered risk
containers, what distinguishes the risk container concept proposed in this dissertation from the existing concepts of DRSpace and component?

Firstly, risk container is a more abstract concept than DRSpace and component because any architecture subset into which risks can be isolated (including DRSpaces and components) are considered risk containers. Secondly, the definition of risk container requires that containers have minimal overlap so that the risk associated with a container is truly attributable to its member elements (i.e., is risk isolating). Here lies the novelty of the risk container concept and research in this dissertation because we don’t know how overlapping DRSpaces and components typically are. Thus, DRSpaces are only potential implementations of the abstract risk container concept: Xiao et al. extracted them from source code rather than designs, did not determine whether design metrics are predictive of the risk isolated, and did not determine the typical degree of overlap between DRSpaces. Likewise, components are only potential risk containers because Shaik, Abdelmoez, and Ammar did not determine the typical degree of overlap between components. Furthermore, SARA was not tested for error-proneness and DRSpaces were not tested for change propagation.

Overall, a risk container is an abstract concept, whereas DRSpaces and components are more concrete, and we cannot say whether DRSpaces and components are valid risk containers for change propagation and error-proneness without further research because neither of the nearest matches provide answers to RQ1 and RQ2. Thus, it is not known whether they support the goal of isolating risks in upfront architectural designs into architecture subsets to produce risk views of architectures that would support practitioners. In later sections this dissertation evaluates how well Design Rule Containers extracted from designs (as opposed to source code that is used with DRSpaces) and two other types of potential risk containers can be used to predict and isolate risks using upfront designs.

The literature review has also identified that further work is necessary in terms of relating risks isolated into containers to the architecture elements that precipitate them. Whilst ISO standards guide practitioners towards the important concepts for architecture descriptions (ISO 42010) and risk management (ISO 31000) they provide only limited advice about how the standards should integrate the practices of software architecture and risk management. Furthermore, although architecture models have been proposed by other researchers (Fabian et al., 2010; Mayer et al., 2019) they have only been evaluated in the security domain and as such contain some security specific terminology and concepts. As explained in Chapter 1, container-based architecture risk assessment (Figure 1-1) is a generic process applicable to all risks that
stakeholders might be concerned about. Whilst this dissertation has only evaluated risk containers for their ability to identify maintainability risks, the wider proposition is that containers could be used for other types of risks such as efficiency, safety, and security risks. That means the risk container proposition needs a more generic risk model to establish traceability from the architecture elements that may give rise to risks, and the architecture elements impacted should those risks materialise. Therefore, the relevant concepts already proposed by ISO 42010, ISO 31000, Fabian et al., and Mayer et al. need to be compiled into a generic architecture risk model to support the risk container proposition. As such, this dissertation addresses that gap by proposing an architecture risk model that is not only based on standard architecture (ISO 42010) and risk concepts (31000), but can describe the risk output of any architecture analysis techniques in order to answer RQ3.

By providing contributions C1-C10 (Table 1-1) this dissertation provides evidence to determine whether the proposition of risk containers can be used to isolate different risks and produce risk views within the constraints of an achievable research project. If the research provides evidence to suggest that isolation is possible and generalisable for error-proneness and change propagation with different containers (Design Rule, Use Case, and Resource), and that the risk containers are meaningful and useful to practitioners, it may support the future development of a process that enables: (i) risk instances to be managed holistically as opposed to treating the class level symptoms in ignorance of a higher-level cause; (ii) an understanding of whereabouts within the overall architecture the risks are to be found; and (iii) determination of the scope of the risks identified in terms of architecture elements. The advantages of (i-iii) are that the risk scope is better understood which means its impact and cost of mitigations can be better estimated enabling practitioners to manage the error-proneness and change propagation risks more effectively. Further future research beyond the work in this dissertation would be needed to establish whether using such a process would provide a return on investment for practitioners.

2.5 Chapter Summary

This chapter began by presenting the results of a systematic literature review of architecture analysis techniques based on Okoli’s (2015) method. The goal of the review was to find existing techniques that can identify maintainability risks in architectural designs to see whether they can isolate risks into architecture subsets and support the risk container concept. The chapter next described a review of standard software design metrics that have the potential to be used with risk containers in order to predict and isolate error-proneness (McCabe, 1976; Chidamber
and Kemerer, 1994; Mitchell and Mancoridis, 2006) and change propagation (Clarkson, Simons, and Eckert, 2004; Abdelmoez et al., 2005, 2006). A third review considered existing methods for modelling risks associated with software architectures to support the goal of producing architecture risk views based on risk containers.

Many of the techniques do not satisfy the risk container concept because they are based on arbitrary architecture subsets. Although the closest matches do associate risks with specific types of architecture subsets (Shaik, Abdelmoez, and Ammar, 2008; Xiao, Cai, and Kazman, 2014), none of the previous research considers the degree of overlap between the architecture subsets with which the risks are associated, and none of the previous research considers which types of architecture subset are the most risk isolating. Therefore, here lies the primary novelty of the research presented in this dissertation: to determine how risks can be isolated into different types of architecture subsets, which are called risk containers. The existing approaches for modelling risks associated with software architectures (Fabian et al., 2010; Mayer et al., 2019) were found to be limited to security risks and not generic enough to be used with the generic container-based software architecture risk assessment process introduced in Figure 1-1.
Chapter 3 - Methodology

This chapter presents the research approach including an overview of the work undertaken, and explains how data was collected in order to evaluate whether error-proneness and change propagation risks can be isolated into different types of risk containers.

3.1 Research Approach

3.1.1 Research Philosophies

The research presented in this dissertation is both postpositivist and pragmatic according to the definitions provided by Creswell and Creswell (2018). There are two reasons why the research is postpositivist. Firstly, because it assumes causes probably determine outcomes, i.e. it tests the theory that risk container properties (coupling, cohesion, and complexity) result in risks (error-proneness and change propagation). Secondly, because it is reductionist, i.e. it assumes that the overall risk container proposition can be evaluated by reducing it into three separate research questions that were independently tested.

The research is also pragmatic because it is problem and real-world practice-oriented. Seeking to increase the chance of project success when measured in terms of budget and schedule is problem-oriented. The emphasis on isolating real project risks (meaningful definition M1), by forming risk containers from design artefacts used in practice (meaningful definition M2), to help practitioners locate risks inducting architectural flaws more easily (meaningful definition M3), in the research definition of risk containers being meaningful to practitioners (section 1.3), makes the research real world practice-oriented as well as being problem-orientated.

According to Creswell and Creswell the overall research approach is an example of mixed methods. Table 3-1 summarises the approaches used to gather quantitative and qualitative evidence as well as providing a reminder of what are the three research questions. The nomenclature used in Table 3-1 (explanatory sequential, convergent, mixed methods, nonexperimental) is defined in Creswell and Creswell (2018). The following three subsections summarise the method taken to answer each question.
<table>
<thead>
<tr>
<th>Research Question</th>
<th>Research Approach</th>
<th>Quantitative Data</th>
<th>Qualitative Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1: How effective are Design Rule, Resource, and Use Case Containers at predicting and isolating the risks of implementation error-proneness and implementation change propagation?</td>
<td>Explanatory sequential mixed methods</td>
<td>Statistical analysis of practitioner design review activity experiment, i.e. time taken and number of error-inducing cyclic dependencies and change impacts identified correctly.</td>
<td>Structured questionnaire to see if the developer’s explanation for the riskiest containers agreed with the metrics.</td>
</tr>
<tr>
<td>RQ2: Does splitting an architectural design into smaller container diagrams help practitioners find error-inducing flaws and identify change impacts?</td>
<td>Convergent mixed methods</td>
<td>Statistical analysis of practitioner survey to determine participant attitudes about whether the risk model is applicable to different risk management stages and development approaches, practitioner preference for the model or textual risk descriptions, and whether the model is missing concepts or contains redundant concepts.</td>
<td>Thematic analysis of open question of participant attitudes about completing the design review activity.</td>
</tr>
<tr>
<td>RQ3: What information is necessary in architecture risk models to describe outputs of architecture analysis techniques?</td>
<td>Convergent mixed methods</td>
<td></td>
<td>Thematic analysis of practitioner survey to determine participant attitudes about whether the risk model is applicable to different risk management stages and development approaches, and whether the model is missing concepts or contains redundant concepts.</td>
</tr>
</tbody>
</table>
3.1.2 Approach to answering RQ1

Central to the risk container proposition is the theory that risks such as error-proneness and change-propagation can be isolated into design subsets (the risk containers). This theory is tested by calculating container level metrics to test hypotheses H1 (if error-proneness and change propagation risks are isolated into Design Rule, Resource and Use Case Containers, the risk inducing architectural features, e.g. components or classes, should exist in relatively few risk containers); H2 (if error-proneness and change propagation risks are isolated into Design Rule, Resource and Use Case Containers, there should be a strong correlation between container level risk indicators and container level project outcomes); and H3 (if error-proneness and change propagation risks are isolated into Design Rule, Resource and Use Case Containers, the set of files that must be changed to fix errors or implement changes should fit neatly into containers), as well as answer RQ1.

To examine the relationship between these two variables (container level design metrics and risk), a non-experimental quantitative approach that combined causal comparative and correlation design was selected. A causal comparative approach was needed to test the relative performance of the different risk container types, and a correlation design was needed to test the strength of association between the container design metrics calculated for each container type and error-proneness/change propagation. As explained in section 1.3, risk isolating is defined as the architecture elements that cause a risk being isolated within the bounds of a risk container and as such element isolation metrics were used to calculate the degree of container overlap (H1) and complement the risk predicting design metrics (H2), in order to determine the most risk isolating container type.

The causal comparative correlation testing required sources of comparable design and outcome data. Outcome data for the maintainability risks selected for testing, error-proneness and change propagation, is collected from source code repositories such as Subversion and Git, and issue tracking systems such as Jira. In such repositories the reference to upfront designs is often through class names found in issue reports and the change sets committed to source code repositories to resolve those issues. It was, therefore, decided to base the research on class architectures. The abstract term “element” referred to in section 1.3 is concerned with classes in Chapters 3-7. Projects from multiple sources that offered the possibility of discussing the findings with the developers were sought to generate further insights. Ideally all projects would have been selected from industry. However, this proved problematic and therefore two industry projects were augmented with two additional academic projects. Section 3.2 further expands
upon the details of, and rationale behind, the specific project selection criteria as well as explaining how the selected projects meet those criteria. Section 3.3 explains the data collected from each project in order to answer RQ1. Sections 5.1 and 6.1 present the metrics used in the causal comparative testing.

Spearman’s rank correlation coefficient (Spearman, 1904) was used to test the strength of the association between the container level metrics calculated from designs and the outcome metrics calculated from implementation source code in order to determine whether the different container types are risk predicting. The classification scheme shown in Table 3-2 is used to classify and make sense of the correlations between design metrics and implementation error-proneness and co-change (StatsTutor, 2021). Strong and significant correlations between the design metrics and implementation error-proneness and co-change would provide evidence for risk containers being able to isolate real project risks (M1) after being formed from design artefacts used in practice (M2). Throughout the dissertation ρ (rho) is used to indicate the strength and α (alpha) is used to indicate the significance level of the Spearman’s rank correlation results.

<table>
<thead>
<tr>
<th>Spearman’s Rank</th>
<th>Correlation Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00-.19</td>
<td>Very Weak</td>
</tr>
<tr>
<td>.20-.39</td>
<td>Weak</td>
</tr>
<tr>
<td>.40-.59</td>
<td>Moderate</td>
</tr>
<tr>
<td>.60-.79</td>
<td>Strong</td>
</tr>
<tr>
<td>.80-1.0</td>
<td>Very Strong</td>
</tr>
</tbody>
</table>

A controlled study needs a control against which to compare the performance of the risk containers against. As explained in section 1.6 it has been decided to test Design Rule, Use Case, and Resource Containers. To reason about their performance for isolating error-proneness and change propagation risks at the design stage, the causal comparative correlation testing was repeated for control containers. The same number of control containers for each container type (Design Rule, Use Case, and Resource) are randomly populated with the same classes allocated to the test containers. The controls help understand whether the clustering algorithms used to populate the containers from the design are isolating risks by comparison to random population.
Once the quantitative data collected from the causal comparative correlation testing had revealed the most risk isolating type of risk container, a structured questionnaire was used to garner feedback from developers to test hypothesis H4 (if risks are isolated into containers, there should be a correspondence between developer perception of containers that were error-prone or associated with excessive change propagation, and container level risk indicators). The structured questionnaire only considered the most change propagation isolating container type. This limitation was due to the scarcity of access to developer time. The questionnaire required developers to rank the extracted risk containers by their perceived level of risk before the metric rankings had been revealed to them; and then provide an explanation about why they think the metrics had predicted which risk containers to have the most risk. This was done to corroborate the quantitative evidence obtained from the causal comparative correlation testing by confirming that the developer’s explanations for why the metrics had predicted the riskiest containers was rooted in coupling problems.

The approach of first collecting quantitative data from the causal comparative correlation testing, before building on that quantitative data with qualitative research, is an example of explanatory sequential mixed methods research (Creswell and Creswell, 2018). The evidence gathered from the causal comparative testing enabled an assessment to be made about whether any of the risk container types satisfy criteria M1 (can be used to isolate real project risks) and M2 (be formed from design artefacts used in practice) of the meaningful to developers criteria defined in section 1.3.

3.1.3 Approach to answering RQ2

Having used causal comparative testing to assess whether any of the risk container types satisfy meaningful criteria M1 and M2, an on-line experiment was used to test hypothesis H5 (smaller Design Rule, Resource, and Use Case Containers diagrams should help practitioners find error-inducing design flaws and identify change impacts more easily than larger architecture diagrams) and answer RQ2 in order to assess whether any of the three risk container types satisfy the final meaningful criterion M3 (help partitioners locate find risk inducing architectural flaws more easily).

The on-line experiment is based around an architecture review scenario for a synthetic architecture because performing an architecture review is an example of a mitigation that is commonly used in practice. The experiment provides qualitative and quantitative evidence
about whether presenting an architecture as a series of container diagrams helps practitioners to locate error inducing flaws and impact changes during the review.

Participants are randomly assigned to one of four different groups (control, Design Rule Container, Resource Container, and Use Case Container) and each group must undertake the review exercise using a series of smaller container diagrams or a single larger control diagram according to their assigned group. Participants also answer an open question to capture their attitudes towards completing the exercise. Section 7.1 describes the experiment designed to answer RQ2 in more detail.

Quantitative data is obtained by recording how many error-inducing cyclic dependencies and change impacts participants can correctly locate in the synthetic design. Qualitative data obtained from participants’ comments is analysed using a thematic analysis by following the steps of Braun and Clarke’s (2006) approach: 1) data familiarisation; 2) coding important features of the data set that might be relevant to answering the research question; 3) generating initial themes that represent broader patterns of meaning from the codes initially identified in step 2, and collating data relevant to the candidate themes; 4) reviewing the themes against the data set to determine whether they tell a convincing story of the data and answer the research question; and 5) finalise and name themes to specify the story and scope behind each theme.

The approach of collecting both quantitative and qualitative data from the online experiment in order to provide a comprehensive analysis is an example of convergent mixed methods (Creswell and Creswell, 2018).

### 3.1.4 Approach to answering RQ3

To determine what kind of information is needed in an architecture risk model to support risk views, in order to test hypotheses H6 (practitioners should view an architecture risk model as being beneficial to risk management) and H7 (practitioners should not think that concepts are missing from the architecture risk model or that the risk model is redundant) and answer RQ3, an architecture risk model that can hold the results of architecture analysis techniques such as risk containers is synthesised. The synthesised risk model is then evaluated using a practitioner survey.

The details of the method used to synthesise the risk model are explained in section 8.1, and the details of the practitioner survey used to evaluate the model are explained in section 8.2. Quantitative data about practitioner experience, attitudes towards applicability of the model to
different stages of risk management (Patterson and Neailey, 2002) in different software development approaches, preference of using the model over textual risk descriptions, and attitudes about whether concepts are missing from or redundant in the model design, is collected using a series of closed questions for statistical analysis. Qualitative data is also captured about these aspects using a series of open questions for thematic analysis using Braun and Clarke’s (2006) approach and comparison with the quantitative data to look for corroboration and contradictions between the two.

Again, the approach of collecting quantitative and qualitative data from the online experiment, to provide a comprehensive analysis, is an example of convergent mixed methods (Creswell and Creswell, 2018).

### 3.2 Software Studied

It is hypothesized that the change propagation probability calculated from a design should correlate to change propagation during its implementation (co-change), and that design complexity and coupling should correlate to implementation error-proneness. Spearman’s (1904) rank correlation coefficient, which is a statistical measurement of the association between two variables, was used to test these hypotheses and answer RQ1, as explained in subsection 3.1.2. This required the following project selection criteria to ensure a fair comparison of container type performance in the Spearman’s rank correlations used to test meaningful criterion M1 (can be used to isolate real project risks): (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) correspondence between the classes and relationships documented in the design and those found in the implementation; (v) a change history of at least 100 revisions; and (vi) available defect (error) reports.

Criteria (i-iii) ensured a fair test because five is the minimum sample size needed to achieve a significance level of $\alpha=0.05$ using a two-tailed test based on the critical values of the Spearman’s rank correlation coefficient (Zar, 1984, Table B.19). If it were not possible to extract a minimum of five containers of a particular type for a given project, then that container type would be disadvantaged because it is not possible to demonstrate a significant correlation by normal standards for that project. Criterion iv ensures a fair test because if a project that did not have correspondence between the classes and relationships documented in the design were
used, it would not be possible to extract containers and calculate design metrics for variable one of the correlation testing. Criteria v and vi ensure a fair test because without change history it is not possible to observe change propagation, and without defect reports it is not possible to observe error-proneness, which are the second variable of the correlation testing. Thus, criteria i-vi ensured projects selected would support fair correlation testing between design metrics and outcome indicators (error-proneness and change propagation) for all three risk container types being evaluated (Design Rule, Resource, and Use Case).

Further selection criteria were added to test meaningful criterion M2, assess the generalizability of risk containers, and corroborate the results of the correlation testing: (vii) industry projects to support the goal of helping practitioners. (viii) projects from multiple sources to ensure that results are not skewed by specific team practices; (ix) at least one project where it was possible to interview developers to check they agreed that the risk predicted was grounded in design problems; (x) projects of different size; (xi) projects including different software functionality; and (xii) projects implemented in different programming languages.

Selection criterion vii is required to support testing meaningful criterion M2 (can be formed from design artefacts used in practice) because it ensures the project sample included projects developed by industry practitioners. Selection criteria viii, x, xi, and xii are required to determine whether risk containers are sufficiently generalizable to work with software projects which may be different in terms of development methods, size, functionality, and programming language. Selection criterion ix is needed to enable a qualitative assessment of risk containers to corroborate or dispute the results of the metric analysis that is supported by selection criteria i-vi.

The four projects included two industrial projects from a software development company. The first industry project contains Unified Modelling Language (UML, Object Management Group, 2017) 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. The API project meets criteria (i-vii).

The second industry project (Server) is the server-side modules of a data management application. Its design is a series of PL/SQL package dependency specifications that documented the dependency graph between packages and use case descriptions identifying the packages supporting each use case. PL/SQL packages are assumed to be analogous to Java classes because both represent nodes when Java and PL/SQL code is transformed into a dependency graph for
input into the 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 the extraction of the package (class) relationships needed to construct the containers without having UML available. The Server project meets criteria (i-vii).

The Lindholmen dataset (Hebig et al., 2016) was searched to find open source projects using UML. The design by Löwer et al. (2014) found in the OpenBundestagsWahl (OBTW) project on GitHub includes comprehensive upfront class and sequence diagrams and classes that encapsulate data retrieved from comma-separated data file resources, satisfying selection criteria (i-vi) and enabling population of all three container types. The project analyses German election results. OBTW was selected as the third software project and first academic project.

The fourth software (and second academic) project is Ps2tsa, a companion application for the Planet Side™ computer game (Daybreak Game Company LLC, 2018). It presents a map of the ensuing game play to help players make better tactical decisions. Ps2tsa meets the selection criteria (i-vi) and it was possible to speak with the student to confirm which classes encapsulated resources and discuss the results.

Table 3-3 summarises the key characteristics of each project including its source, number of classes in the design, number of classes in the implementation, lines of code in the implementation, and number of revisions found in the source code repository. Having correspondence between the design and implementation (criteria iv) 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 that correspondence. The number of design classes shown in Table 3-3 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 they had not been kept up to date during the maintenance phase. Despite this limitation, calculating Coverage (in Table 3-3) 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.
Table 3-3 Software Projects Studied

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UML</td>
<td>Classes</td>
</tr>
<tr>
<td>API</td>
<td>Industry</td>
<td>Yes</td>
<td>188</td>
</tr>
<tr>
<td>Server</td>
<td>Industry</td>
<td>No</td>
<td>31</td>
</tr>
<tr>
<td>Ps2tsa</td>
<td>Academia</td>
<td>Yes</td>
<td>36</td>
</tr>
<tr>
<td>OBTW</td>
<td>Academia</td>
<td>Yes</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 3-3 shows that criterion vii is satisfied because the API and Server projects were sourced from industry. Criterion viii is met because the API and Server projects were sourced from a software development company, OBTW was sourced from GitHub, and Ps2tsa was sourced from a former Open University student. Furthermore, although API and Server came from the same software development company, they were developed by different teams. It was possible to meet criterion ix by interviewing the API and Ps2tsa developers to check they agreed that the risk predicted was grounded in design problems. In Table 3-3 the column labelled ‘Classes’ presents the total number of classes and ‘KLOC’ presents the total lines of code for each project. Together, ‘Classes’ and ‘KLOC’ indicate criterion x is satisfied because the projects used vary in size considerably. Criterion xi is met because all four projects provide different functionality: the API is an Application Programming Interface; Server is the server-side business logic for a database management system; Ps2tsa is a companion application for a computer game; and OBTW is software for analysing election results. Finally, criterion xii is satisfied because API, Ps2tsa, and OBTW are implemented in Java, whereas the Server project is implemented in Oracle PL/SQL.

3.3 Data Collection

Data about each of the projects listed in Table 3-3 was collected into Comma Separated Value files. The three files per project were: (i) an elements file containing data about each class in the project; (ii) a file containing the relationships between the classes; and (iii) a file containing data about changes made to the classes during development and maintenance. These three files were used as the input to a suite of automated analysis tools developed to test hypotheses H1, H2, and H3, by calculating metrics to determine whether the different types of containers can be used to predict and isolate error-proneness and change propagation using
upfront design diagrams. The data files used are available for download from The Open University’s open research data repository using the following digital object identifier: 10.21954/ou.rd.14737941

### 3.3.1 Element Data

The element file contains information from both the design and source code of each project to provide details about all the classes belonging to the software development project. The file is organised into the columns shown in Table 3-4.

A row for each class in the project was automatically created by combining the Unix ls, grep, pipe, wc, and redirect commands in a script: ls was used to list project files; grep was used to identify the Java or PL/SQL files in that list; wc -l was used to calculate the Lines of Code (LOC) value for each class; and the output was redirected to the project’s element file.

#### Table 3-4 The Element Data File Format

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Name</td>
<td>Name of the class in the implementation (source code).</td>
</tr>
<tr>
<td>LOC</td>
<td>Number of lines of code the class is composed of.</td>
</tr>
<tr>
<td>Changes</td>
<td>Total number of times the class has been modified.</td>
</tr>
<tr>
<td>Errors</td>
<td>Total number of defects reported and fixed.</td>
</tr>
<tr>
<td>In Design</td>
<td>‘Y’ indicates the class is explicit in the design;</td>
</tr>
<tr>
<td></td>
<td>‘I’ indicates the class is implicit in the design;</td>
</tr>
<tr>
<td></td>
<td>‘N’ means the class is not in the design.</td>
</tr>
<tr>
<td>Design Name</td>
<td>Name of the class in the design.</td>
</tr>
</tbody>
</table>

The number of changes for each class was calculated by determining the total number of times the class had been committed (updated) into source code repository (Subversion/Git). In the case of the API, Server, and OBTW projects, the team’s development practices associated ticket identifiers with each commit into the Subversion/Git source code repository. In addition, tickets relating to defects were categorised as ‘defect’ for API and Server and ‘bug’ for OBTW in the project’s issue tracking system. These practices allowed the value of errors to be calculated.
by summing the number of change sets associated with each class that are also associated with a ‘defect’ or ‘bug’ ticket.

Ps2tsa did not follow these practices and an alternative approach to calculating errors was needed. For Ps2tsa, the student developer retrospectively identified the change sets associated with fixing defects by reading the Subversion commit comments and reviewing the source code changes made. This was only feasible because the project was small, and the development had only recently finished at the time of data collection, which meant the work was still fresh in the developer’s mind.

The ‘In Design’ column indicates whether a class in the implementation source code was present in the project’s upfront design documentation. This column has three options because some classes were only implicitly in the API design as will be explained in section 4.4. The Design Name column compensates for cases where the originally intended design name is different in the implementation. In the cases of the API, Server, and Ps2tsa, a discussion of the implementation mismatches with the software developers enabled identification of the correct design name. In the case of OBTW the design and implementation names corresponded.

3.3.2 Relationship Data

The relationships file was based on the design of each project and contains a list of relationships between classes that belong to the software development project. It was manually populated for each project because the API and Server designs were printed UML documents and the Ps2tsa and OBTW designs were binary image files. Table 3-5 shows the relationship file’s column structure. Joining rows in the relationship file to rows in the element file is achieved by matching values in the relationship file’s source and target columns to the design name column values found in the element file.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Generalisation sub class or an aggregation, composition, or dependency source, i.e. the source class is coupled to the target class.</td>
</tr>
<tr>
<td>Target</td>
<td>Generalisation super class or an aggregation, composition, or dependency target, i.e. the target class is used by the source class.</td>
</tr>
<tr>
<td>Type</td>
<td>A=Aggregation, C=Composition, D=Dependency, G=Generalisation or Realisation.</td>
</tr>
</tbody>
</table>
Unless otherwise specified, generalisations are assumed to be navigable from the sub-class to the super-class only, compositions are assumed to be navigable from the composite class to the composed class only, and aggregations are assumed to be navigable from the aggregation class to the aggregated class only. As such a single entry is recorded in the relationships table for generalisation, composition, and aggregation relationships unless the design explicitly specifies bidirectionality. If the design does not indicate the direction (navigability) of dependency relationships, the relationship is assumed to be bi-directional and hence two relationships are added to the relationships file, one for each direction. A single row is added to the relationships file if the navigability of dependency relationships is specified.

Only relationships explicitly shown on the design are transcribed into the relationships file. Implicit relationships are not transcribed. For example, suppose a class diagram includes abstract class A, which has an abstract method X, requiring class B as a parameter. This indicates class A is dependent upon class B and a dependency from A to B is added to the relationships file. If the designer adds a concrete sub-class of class A called class C, a dependency from C to B would be added to the relationships file only if method X is shown on class C, even though the design implies class C must provide an implementation of method X (by virtue of it inheriting an abstract method from class A), which therefore implies class C is also dependent upon class B. This sticks to the principle that only documented relationships are considered.

Dependency relationships for method parameters or return types that are Java language classes are not included, i.e. dependency relationships are only added to the relationships file if the parameter class is also a member of the software project’s design and not a Java language class. This is because it is coupling between the designed classes that is the subject of the research. In the case of the API an exception to this rule was needed for classes returning lists of other classes found in the design. That is because the design specifies collection return types as java.lang.List<Property> for example, where Property is a class in the design. In such cases, the design class having a method returning java.lang.List<Property> is assumed to be dependent upon the Property class.

3.3.3 Co-Change Data

The co-change file is based on the source code repository of each project and contains Subversion/Git changes associated with each class. Table 3-6 shows the co-change file’s column structure.
Table 3-6 The Co-Change Data File Format

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Name</td>
<td>Name of the class referenced in the change set.</td>
</tr>
<tr>
<td>Change ID</td>
<td>The Subversion/Git change set identifier.</td>
</tr>
</tbody>
</table>

To extract a given change set (set of files that have changed together, i.e. co-changed) read all rows that share Change ID in the file. The co-change file was automatically populated for each project by developing a parser to read the Subversion/Git XML log file and convert it into the format shown in Table 3-6.

### 3.3.4 Container Data

An additional file stores the results of applying the Design Rule, Resource, and Use Case Container population algorithms (described in section 4.1) to the project designs. The file is organised into the columns shown in Table 3-7. All rows sharing the same value of Container are the members of a given container.

Table 3-7 Container Data File Format

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Name</td>
<td>Name of the class allocated to a container.</td>
</tr>
<tr>
<td>Container</td>
<td>Name of the container the class is allocated to.</td>
</tr>
</tbody>
</table>

### 3.4 Chapter Summary

This chapter began by explaining how the methodology used in this dissertation is both postpositivist and pragmatic according to the definitions provided by Creswell and Creswell (2018). Having established the philosophical basis for the methodology, this chapter next described the explanatory sequential and convergent mixed method approaches used to answer the three research questions addressed in this dissertation. The chosen methodology required selection of software development projects to analyse using causal comparative correlation testing. Therefore, the chapter next described the criteria used to select projects and the details of the projects selected. Finally, the chapter explained how data was collected from the sample
of projects used to evaluate whether error-proneness and change propagation risks can be isolated into Design Rule, Resource, and Use Case Containers.
Chapter 4 - Extracting Risk Containers

This chapter describes the detailed algorithms used to extract risk containers from the software (contribution C1), the results of applying the container extraction algorithms to the sample of software projects (contribution C2), the element isolation metrics (contribution C3), and the element isolation results (contribution C4). As explained in chapter 1, this research was bounded by the number of container types tested to: (i) Design Rule Containers, which contain the classes that are based on a modularising design rule; (ii) Use Case Containers, which contain classes in the sequence diagram of a use case; (iii) and Resource Containers, which contain classes that depend on external resources such as database tables, services, and files. These container types were selected because they represent different and commonly used approaches to decomposing software systems from different architectural perspectives. Arguably all three container types are specialisms of the Concern Graphs proposed by Robillard and Murphy (2002) because they represent the architecture from the perspective of a specific concern: modularisation; use case fulfilment; and resource usage. More specifically, Design Rule Containers are based on an interpretation of Xiao et al.’s (2014) Design Rule Spaces except that Design Rule Containers are extracted from upfront designs whereas Xiao et al. extracted Design Rule Spaces from source code. Using these three container types allowed evaluation of different container types, perspectives, and diagrams to determine which ones help analysis of the selected maintainability risks (error-proneness and change propagation).

4.1 Risk Container Population

Three sets of containers were created for each project using the method explained in detail throughout this section: 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 Figure 4-2, which shows an example of how DRCs are extracted, the root of container A is class c1, the roots of container B are classes c2, c3, and c4, and the roots of container C are classes c3, c5, and c6. In Figure 4-4, which shows an example of how RCs are extracted, the root of container A is class c5 and the root of container B is class c4. In Figure 4-5, a use case sequence diagram, the 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 Figure 4-1 and Figure 4-3, 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 it was confirmed with the developers, because the source code was not available, 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.

```java
class Relationship {
    source : Class -- Generalisation sub-class or Aggregation/Composition/Generalisation source
    target : Class -- Generalisation super-class or Aggregation/Composition/Generalisation target

    getTarget() { return self.target }
    getType() { return self.type }
}

class DesignGraph {
    edges : Set
    addEdge(dependency : Relationship) { self.edges.add(dependency) }
    getEdges() { 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 }
    addMember : (Class) { self.members.add(member) }
    addAll(members : Set) { self.members.addAll(members) }
    getMembers() { 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

    populatedRDC(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, "Aggregation"));
            drc.addAll(graph.getDirectDependencies(root, "Composition"));
            drc.addAll(graph.getDirectDependencies(root, "Dependency"));
        }

        return drc
    }

    Figure 4-1 Design Rule Container Population Algorithm

DRCs were constructed using the algorithm represented as object-oriented pseudo code in Figure 4-1. As explained by Xiao, Cai, and Kazman (2014), design rule classes are the key
interfaces that separate an architecture into independent modules. They stem from Baldwin and Clark’s (2000) design rules. The method supports three types of design rules: abstractions, compositions and patterns. Figure 4-1 presents the population rules for DRCs (the interpretation of DRSpaces used) as an object-oriented pseudo code algorithm. Once DRCs have been initially populated with their roots based on patterns, abstractions or compositions (line 47 of the Figure 4-1 algorithm), they are expanded with direct aggregations and dependencies (lines 51-53).

For an abstraction DRC the roots parameter must be a full inheritance tree, i.e. starting from the most abstract super-class and including all sub-classes. In Figure 4-2, 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 to container B by line 52 because it is a dependency of c3; c6 is not added because it is not coupled to the abstract class or its sub-classes.

The relationships between classes c3, c5, and c6 lead to the creation of Container C in Figure 4-2 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.

<table>
<thead>
<tr>
<th>DR Container</th>
<th>Design Rule Type</th>
<th>Roots</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Pattern</td>
<td>c1</td>
<td>c1, c2</td>
</tr>
<tr>
<td>B</td>
<td>Abstraction</td>
<td>c2, c3, c4</td>
<td>c2, c3, c4, c5</td>
</tr>
<tr>
<td>C</td>
<td>Composition</td>
<td>c3, c5, c6</td>
<td>c3, c5, c6</td>
</tr>
</tbody>
</table>

Figure 4-2 Design Rule Container Example
In Figure 4-2, the designer chose to base class c1 on the singleton design pattern and container A is an example of a pattern design rule. 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.

In summary, the type of design rule being processed (abstraction, composition, or pattern) dictates the roots that must be supplied to the algorithm, and that are added to the DRC by line 47 of the algorithm shown in Figure 4-1. However, the same expansion rules (lines 51-53 of the algorithm) are used to expand all design rule containers irrespective of the type of design rule being processed to create the DRC.

In all cases (abstraction, composition, and pattern), the DRCs produced from the algorithm must satisfy two additional rules: (i) they must contain more than one class; and (ii) they cannot be a subset of another container. Rule (i) is derived from Xiao, Cai, and Kazman’s (2014) original definition of Design Rule Spaces (DRSpace) which explains each DRSpace is a graph whose vertices are the related classes the DRSpace is composed of, and the edges are the relationships between those related classes. As DRSpaces are based on one or more relationships between classes, DRSpaces must contain at least two classes to be based on at least one relationship as per Xiao, Cai, and Kazman’s definition. Rule (ii) was introduced to account for a scenario observed in both the API and OBTW projects whereby the classes making up a composite structure were also all in an inheritance relationship with an abstract class. Without applying rule (ii) this would synthesize two almost identical DRCs, the first containing the abstract class and all of the composition classes, and the second containing just the composition classes. Furthermore, in one API case the abstraction class was also a member of the composition which meant that two identical DRCs would be produced. Having two highly similar DRCs is unnecessary and unhelpful because the same risk would be isolated twice. Therefore rule (ii) was introduced to prevent a risk being isolated into two similar containers. For example, if class c6 were removed from Figure 4-2, the remaining members of container C (c3 and c5) would become a subset of container B, and as such container C would no longer be required due to rule (ii).
Figure 4-3 Resource Container Population Algorithm

Figure 4-3 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 Figure 4-4, 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 encapsulating class (c5) by line 71 of the algorithm. Thus, N will be the maximum number of recursive aggregations, compositions, or 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 Figure 4-5 shows that classes c1, c2, c3, c4, and c5 are all used to fulfil Use Case 1. All classes referenced by the use case 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.

![Use Case Container Example](image)

**Figure 4-5 Use Case Container Example**

### 4.2 Control Container Population

The test Design Rule, Use Case, and Resource Containers are based on structural dependencies. These three types of test containers were selected to determine whether clustering classes based on structural dependencies within modules, use cases, and resource usage helps to isolate risks into the containers created. Therefore, controls that are not based on structural dependencies are required, to determine whether any results indicating that risks can be isolated into the test containers are due to the structural dependencies upon which the test containers are based.

Packages were considered for use as the control, but not all of the project designs were organized in packages. Therefore, control containers for each container type were created by randomly allocating the classes of the test containers to the same number of control containers. 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 error-proneness and change propagation than random population. If the performance of the control containers is similar or better than their equivalent test container type, the results would suggest that clustering classes based on structural dependencies within modules, use cases, and resource usage does not help to isolate error-proneness and change propagation risks into containers. The test and control containers were stored in a file formatted as described in section 3.3.4 for each project.

4.3 Automating Container Analysis

The goal was to automatically populate and analyse risk containers from the design, to avoid human error threatening the evaluation, and to demonstrate that practitioners could easily apply the approach. An Automated Container Analysis Tool (ACAT) was developed in the Java programming language version 1.8 to analyse the performance of risk containers. ACAT is designed to calculate and correlate the risk container level design metrics with implementation outcome metrics described in later sections. ACAT has been designed to be easily extended for other types of risk containers and other risks. Figure 4-6 illustrates the overall ACAT processing model as an activity diagram.

![Figure 4-6 Automated Container Analysis Tools Activity Diagram](image)

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 a relationships
file as described in section 3.3.2. 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 as described in section 3.3.3. Once the relationships and change files have been created, the elements file is created in accordance with the file format specified in section 3.3.1. Next, the process populates the risk containers by applying the algorithms described in section 4.1 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 Figure 4-1 (lines 51-53) and Figure 4-3 (line 71). The containers created are stored in a file for each project that is formatted as per section 3.3.4. Once the container, elements, change, and relationship files have been populated, the tools automatically calculate the metrics described in later sections that can be used to predict and isolate error-proneness and change propagation using upfront design diagrams. The metrics calculated are output in an HTML report.

The ACAT supports two modes of operation: 1.) analyse design; and 2.) analyse design and compare to implementation. Mode (1) is for analysing an upfront design at the design stage to predict which containers will have the most error-proneness (Chapter 5) and change propagation (Chapter 6) using metrics calculated from the design. Mode (1) automates the Select Container Types and Risk Indicators, Extract Risk Containers, and Calculate Container Risk Indicator activities shown in Figure 1-1 of the introduction. As such Mode (1) automates the main proposition of this thesis by predicting which containers will isolate the greatest risk. Mode (2) is used to correlate the design metrics to implementation metrics. The main purpose of mode (2) is to support the experiments required by the method used in this dissertation, i.e. to corroborate whether predictions based on the design are realised in the implementation and test hypothesis H2 (if error-proneness and change propagation risks are isolated into Design Rule, Resource and Use Case Containers, there should be a strong correlation between container level risk indicators and container level project outcomes) in support of answering RQ1: how effective are Design Rule, Resource and Use Case Containers at predicting and isolating the risks of implementation error-proneness and implementation change propagation? Mode (2) could also be used during the implementation and maintenance phases to validate predictions before
acting upon risk mitigations. However, during the implementation phase and beyond, it would make sense to rebuild the design graph based on source code to take account of any drift between upfront design and implementation. The latter use of Mode (2) is outside the scope of this dissertation which is concerned with the risk analysis at the design stage as opposed to the implementation stage. The activities coloured in grey in Figure 4-6 support Mode (2) only, those not coloured in grey support both modes of operation.

The industry designs were printed documents and the UML diagrams for OBTW and ps2tsa were binary image files. These forced 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 Figure 4-2) and Resource Containers (e.g., classes c4 and c5 in Figure 4-4) 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 as shown in Table 4-1.

Table 4-1 Method Automation

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Container Sub Type</th>
<th>Stage of Figure 4-6 Automated Container Analysis Tools Activity Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Acquire Data</td>
</tr>
<tr>
<td>DRC</td>
<td>Abstraction</td>
<td>Manual</td>
</tr>
<tr>
<td>DRC</td>
<td>Composition</td>
<td>Manual</td>
</tr>
<tr>
<td>DRC</td>
<td>Pattern</td>
<td>Manual</td>
</tr>
<tr>
<td>RC</td>
<td>N/A</td>
<td>Manual</td>
</tr>
<tr>
<td>UCC</td>
<td>N/A</td>
<td>Manual</td>
</tr>
</tbody>
</table>

Table 4-1 indicates that most parts of risk container analysis were automated, which suggests that should a machine-readable UML model be available, the method can be fully automated because transformation of UML relationship data to the relationships file format described in section 3.3.2 is straightforward. 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, or UML recovered from source code. This means that although the dissertation 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.

A minor limitation of the tools is that class names are assumed to be unique within a software project. Although this assumption is safe for the projects studied, it prevents ACAT analysis of projects with duplicate class names distinguished by package. This is a limitation of the code implementation and support for duplicate class names could be achieved with additional coding effort should the ACAT be further developed for commercial use.

4.4 Implementation Coverage

Implementations can diverge from upfront designs over time when: (i) classes are renamed during implementation; (ii) classes that are not in the design are added during implementation; (iii) classes that are proposed in the design are not implemented; and (iv) dependencies between classes that are not described in the design are added or removed during implementation. It is therefore important to ensure that there is correspondence between the upfront designs and the implementations in the four projects studied. Otherwise, risk containers and their metrics populated from the design would not relate to the classes in the implementation and the outcome metrics. Therefore, correspondence is essential for a predictive effect between risk container metrics calculated from the design and implementation outcome metrics that indicate error-proneness and high change propagation probability. There are two ways that design relevance can be assessed; firstly by checking the percentage of the implementation that could be mapped to risk containers based on the design, and secondly by observing stronger and more significant correlations between design metrics for the test containers which are based on the design rather than the control containers which are based on random assignment. The latter is discussed in subsequent sections, but the results of the former implementation coverage is calculated as follows.

\[
SP(r) = \frac{\Sigma_{c \in \text{cont}(c)} 1}{|c|}
\]

Equation 4-1 Size Percent (SP)
Size Percent (SP) is the percentage of the project’s implementation classes that are allocated to a container and is calculated using Equation 4-1 where: \( r \) is the risk container; \( M \) is the set of all the member classes allocated to container \( r \); \( C \) is the set of all the classes in the implementation of a particular software system; and \( \text{cont}(c) \) is the number of all containers that class \( c \) has been allocated to. In cases where a class is allocated to more than one container, 1 is divided by the number of containers when calculating SP to avoid double counting. Once SP has been calculated for each container, the Implementation Coverage % is calculated by summing SP for all containers of a given type per software project. Table 4-2 summarises the number created and Implementation Coverage % of each container type per project. The percentage coverage of the implementation by the risk containers was highest for Design Rule Containers in all projects. Risk container coverage is important because it indicates how much of the implementation can be assessed for risks using the method presented.

**Table 4-2 Number and Coverage of Containers**

<table>
<thead>
<tr>
<th>Name</th>
<th>DRCs</th>
<th>RCs</th>
<th>UCCs</th>
<th>DRCs</th>
<th>RCs</th>
<th>UCCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>15</td>
<td>23</td>
<td>35</td>
<td>59.78</td>
<td>45.62</td>
<td>16.18</td>
</tr>
<tr>
<td>Server</td>
<td>9</td>
<td>14</td>
<td>68</td>
<td>12.50</td>
<td>9.91</td>
<td>6.47</td>
</tr>
<tr>
<td>Ps2tsa</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>79.07</td>
<td>34.88</td>
<td>32.56</td>
</tr>
<tr>
<td>OBTW</td>
<td>9</td>
<td>8</td>
<td>12</td>
<td>46.97</td>
<td>43.94</td>
<td>25.76</td>
</tr>
</tbody>
</table>

The figure of 59.78% for implementation coverage by the API DRCs requires further explanation because Table 3-3 shows that 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 as such both the interface and its implementation were assigned to the appropriate risk container, even though only the interface is referenced in the design diagram. This is the reason why API implementation coverage is higher than the 42% that might be expected by comparison to the proportion of implementation classes that were in the design shown in Table 3-3 for API Design Rule and Resource Containers. The 1:1 implementations that are not shown on the API class diagrams...
were marked with ‘In Design=I’ in the element file created for the API project to denote that they are implicitly in the design.

It should also be noted that with respect to treating 1:1 interface class pairs as a single class, the implementation is not added to the DRC when the interface has been added as a result of relationship expansion (lines 51-53 of Figure 4-1). In such a case only the interface is added when processing lines 51-53 because the abstraction/pattern/composition class is assumed to be dependent on the interface of the 1:1 pair only. That is because the end of such relationships reach the DRC boundary.

4.5 Element Isolation Metrics (Container Overlap)

This section presents the method and metrics used to determine the degree of overlap (element isolation) for the different container types tested (contribution C3), and the results of applying that method to the four software projects studied (contribution C4).

\[
CPC = \frac{1}{|C|} \sum_{c \in C} \sum_{r \in R} \begin{cases} 1 & \text{if } c \in r \\ 0 & \text{if } c \notin r \end{cases}
\]

Equation 4-2 Containers per Class (CPC)

Containers Per Class (CPC) is the mean number of containers to which each class is allocated and is calculated using Equation 4-2 where: C is the set of all the classes in a particular software system; and R is the set of all risk containers of a given type for a particular software system. CPC indicates the average amount of class sharing between containers. For example, the CPC for Figure 4-7 is 1.5, because classes c1, c4, and c6 are in one container and classes c2, c3, and c5 are in two containers. The min, max, lower, and upper quartile are also calculated to check the distribution and ensure that the mean values are not misleading.
Internal Coupling (IC) is the mean, over all containers, of the percentage of dependencies between two classes inside the same container and is calculated using Equation 4-3 where: $R$ is the set of all risk containers of a given type for a particular software system; $M$ is the set of all the member classes allocated to a specific risk container; $\text{depi}(c)$ is the number of outgoing dependencies of class $c$ where the dependency is also a container member; and $\text{dep}(c)$ is the number of outgoing dependencies of class $c$. A container’s internal coupling is the ratio between the dependencies that are within the container and the total number of dependencies in which the container’s classes are involved. A dependency only counts if the dependent class is within the container.

$$IC = \frac{1}{|R|} \sum_{c \in R} \frac{\sum_{c \in M} \text{depi}(c)}{\sum_{c \in M} \text{dep}(c)}$$

Equation 4-3 Internal Coupling (IC)

For example, in Figure 4-7, class $c_2$ does not depend on classes $c_3$ or $c_4$ (it’s the other way around), so the internal coupling of container A is 100%: the one internal dependency is also the total dependencies of $c_1$ and $c_2$. 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 Figure 4-7 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).
Figure 4-8 presents a box plot showing 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 for RCs (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.

![Figure 4-8 Containers per Class (x=mean marker).](image)

Figure 4-9 shows that mean levels of internal coupling (IC) are greatest for DRCs in three projects (API, Server, and ps2tsa). DRCs have the second highest IC mean for OBTW, where RCs have a 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. DRCs therefore isolate the most coupling in all projects and it is based on the relationships stemming from the design rules and not random assignment.
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 mean internal coupling in three projects, but the highest median and overall distribution 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.

4.6 Validity

This section reviews the approach taken to extract risk containers for construct validity threats using Messick’s (1987) classification of consequential, content, substantive, structural, external, and generalizability threats.

4.6.1 Consequential

Design Rule Containers aim to extend the work of Xiao, Cai, and Kazman (2014) and Wong et al. (2009). One threat to validity is that the Design Rule Hierarchy (DRH) algorithm or the DRSpaces they are based on could have been misinterpreted. The consistency of the change propagation results discussed in Chapter 6 with Wong et al.’s underlying hypothesis suggests that DRH has been interpreted correctly.
Manual identification of pattern design rules is prone to adhoc human error. The consequence 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, error-proneness and change propagation risks 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 has been 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’ (see section 9.2).

As Use Case Containers were manually populated, classes could have been omitted or added to a particular container by mistake. It is argued 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.

The programs used to analyse the projects have been comprehensively unit tested, with the Java JUnit testing framework. Whilst some bugs will undoubtedly still exist, reasonable precautions to find and fix bugs prior to analysing the results have been taken. The aspect of this threat relating to the automated extraction of risk containers from the design, was mitigated by inspecting the design relationships to identify what risk containers should be extracted, and checking that the contents of the container data files were consistent with the expectation.

Using the Unix wc -l command to calculate lines of code is crude because the result will include comment lines as well as code lines. Furthermore, using wc -l per file assumes each file contains only one Java class or one PL/SQL package. Inspection confirmed the significant majority of API, Ps2tsa, and OBTW .java files contained only one Java class and that all Server .pkb files contained just one PL/SQL package. Overall, the threats posed by this method of calculating lines of code are tolerable because the same method was used for all four projects.
4.6.2 Generalizability

It was 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 presented in Chapters 4, 5, and 6 further supports that view.

Not all software development projects will have the upfront designs the method requires. The emergence of agile methods may magnify this threat. However, Nord, Ozkaya, and Kruchten (2014) explain that architecture is needed in large scale agile projects to avoid excessive redesign later. The method is therefore more likely to be applicable to larger scale software development where the architecture risk assessment has produced upfront designs. Nord, Ozkaya, and Kruchten’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, the 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 4.3.

4.7 Chapter Summary

This chapter began by explaining the algorithms used to extract the Design Rule, Resource, and Use Case Containers (contribution C1, Table 1-1) from the designs of two industry (contribution C2, Table 1-1) and two open source software projects. The containers extracted using the algorithms are the test containers. The chapter then explained how additional containers based on random assignment of classes to the same number of containers act as a control to determine whether the characteristics of the test containers are due to design properties (coupling, cohesion, and complexity) of the container members. Automation was required to make this research rigorous and support the goal of helping practitioners. Therefore, this chapter then described the degree to which container extraction and metric calculation was automated in the Automated Container Analysis Tools. Having explained the container
extraction algorithms and how extraction and metric calculation was automated, this chapter described the results of the container extraction. Firstly, in terms of how much of the implementation is covered by the different risk container types populated from the design, and secondly the degree of overlap between containers of each type in terms of containers per class and proportion of coupling that is internal to the containers (contribution C3, Table 1-1). Finally, the chapter discussed the validity threats to the container extraction approach used.
Chapter 5 - Predicting Error-Proneness

This chapter presents the metrics used to determine whether risk containers can be used to isolate and predict error-proneness (contribution C5) and the results of applying those metrics to the four software projects studied (contribution C6). Together, chapters four and five provide an answer to RQ1 for the risk of implementation error-proneness. As explained in section 3.1.2, nonexperimental causal comparative correlation testing (Creswell and Creswell, 2018) was used to determine which container types are the most error-proneness predicting. This is achieved by calculating design metrics (variable one) for each risk container and checking if a correlation exists to implementation error-proneness indicators for each container (variable two). Thus, to support the correlation testing, design metrics that are known to be predictive of error-proneness, and metrics known to be indicative of implementation error-proneness, need to be calculated for risk containers. Section 5.1 explains: (i) which standard design metrics are known to be predictive of error-proneness, and how they can be calculated for risk containers, to derive variable one used in the correlation test; (ii) that four different design metrics were chosen for variable one to test whether outgoing coupling, incoming coupling, cohesion, or complexity metrics provide the most effective error-proneness predictor for the different risk container types; and (iii) the indicator of implementation error-proneness used and how it is calculated for risk containers to derive variable two using in the correlation test.

Section 5.2 presents how error-proneness is distributed over the different risk container types tested. This is important because if error-proneness were evenly distributed across the extracted containers it would not be possible to use design metrics to predict which containers have a higher risk of being error-prone than others. Section 5.3 presents the results of using a risk container design metric based on outgoing coupling to predict error-proneness, section 5.4 presents the results of using a risk container design metric based on incoming coupling to predict error-proneness, section 5.5 presents the results of using a risk container design metric based on cohesion to predict error-proneness, and section 5.6 presents the results of using a risk container design metric because on complexity to predict error-proneness. Having presented the detailed error-proneness prediction results, the chapter discusses validity threats in section 5.7, before summarising the error-proneness results and drawing conclusions in section 5.8.

5.1 Error-proneness Metrics

It is hypothesised that containers encapsulating less cohesive, more complex, and tightly coupled structures will be more difficult to develop and maintain, thus increasing the risk of
coding errors that lead to software defects. Standard coupling, cohesion, and complexity metrics calculated at the container level are compared to container error-proneness to test this hypothesis. The container level metrics are based on Chidamber and Kemerer’s (1994) Coupling Between Objects (CBO), Mubarak, Counsell, and Hierons’s (2010) version of Henry and Kafura’s (1981) Fan In (FI), Mitchell and Mancoridis’ (2006) Cluster Factor (CF), and McCabe’s (1976) Cyclomatic-Complexity (MCC).

CBO and FI have been selected to investigate the relationship between container coupling and error-proneness because previous research has demonstrated both CBO (Basili, Briand, and Melo, 1996; Briand et al., 1998, 2000; Yu, Systa and Muller, 2002; Gyimothy, Ferenc, and Siket, 2005) and FI (Yu, Systa and Muller, 2002; Gyimothy, Ferenc, and Siket, 2005) to be predictive of class error-proneness.

It was not possible to use Chidamber and Kemerer’s (1994) Lack of Cohesion in Methods (LCOM) metric to investigate cohesion because knowledge required to calculate it, about which methods access class instance variables, is not available in any of the project designs. CF was selected as alternative to LCOM for cohesion because it can be calculated solely on the basis of the structural graph between classes and has been widely cited (Wu, Hassan, and Holt, 2005; Mitchell and Mancoridis, 2006; Praditwong, Harman, and Yao, 2011; Garcia, Ivkovic, and Medvidovic, 2013). However, it should be noted that Praditwong, Harman, and Yao (2011) recommend caution with respect to using CF as a predictor or error-proneness based on other studies that have shown LCOM to be a predictor of error-proneness (Briand et al., 2000; Gyimothy, Ferenc, and Siket, 2005). Previous studies have shown MCC is predictive of error-proneness (Munson and Khoshgoftaar 1992, Coleman et al., 1994).

McCabe proposed MCC as a metric for calculating the complexity of control structures (sequence, if then else, while and until). Risk containers which are based on structural relationships do not have control structures. However, McCabe’s cyclomatic complexity is an application of the cyclomatic number which indicates the relative complexity of strongly connected directed graphs (Berge, 1973). This means MCC can be used to calculate the relative complexity of risk containers once they have been transformed into strongly connected directed graphs.

Chidamber and Kemerer (1994) defined CBO for a class as “a count of the number of other classes to which it is coupled” (p. 486). Container CBO is calculated using Equation 5-1 where: $r$ is the risk container; $M$ is the set of all the member classes allocated to container $r$; $\text{dep}(c)$ is the
number of outgoing dependencies of class $c$; and $\text{cont}(c)$ is the number of containers that class $c$ has been allocated to. Division by $\text{cont}(c)$ is necessary because the relationship causing the coupling is assumed to support the functionality of all containers the class is a member of equally.

$$\text{CBO}(r) = \sum_{c \in N} \frac{\text{dep}(c)}{\text{cont}(c)}$$

Equation 5-1 Coupling Between Objects (CBO)

Using the classes in Figure 5-1 as an example: class $c_1$ has one dependency (CBO=1), class $c_2$ has no dependencies (CBO=0); class $c_3$ has two dependencies (CBO=2); class $c_4$ has one dependency (CBO=1); class $c_5$ has one dependency (CBO=1); and class $c_6$ has no dependencies (CBO=0). Summing the CBO of each class allocated to the container results in the container’s CBO. However, because classes can be allocated to more than one container, the CBO of each class is divided by the number of containers to which that class has been allocated when summed to give the container’s CBO as per Equation 5-1. Thus, the CBO of container B is calculated by summing the dependencies of its member classes ($c_2=0/2=0 + c_3=2/2=1 + c_4=1/1=1 + c_5=1/2=0.5$) to give 2.5. Container A’s CBO is ($c_1=1/1=1 + c_2=0/2=0$) equal to 1 and for container C the CBO is ($c_3=2/2=1 + c_5=1/2=0.5 + c_6=0/1=0$) equal to 1.5.
Container level FI is the sum of all incoming coupling to the container’s member classes and is calculated using Equation 5-2 where: \( r \) is the risk container; \( M \) is the set of all the member classes allocated to container \( r \); \( f(c) \) is the number of incoming dependencies upon class \( c \); and \( \text{cont}(c) \) is the number of containers that class \( c \) has been allocated to. Division by \( \text{cont}(c) \) is necessary because the relationship causing the coupling is assumed to support the functionality of all containers the class is a member of equally. For example, in Figure 5-1: no classes depend on class \( c_1 \), three classes depend on class \( c_2 \); no classes depend on class \( c_3 \); no classes depend on class \( c_4 \); one class depends on class \( c_5 \); and one class depends on class \( c_6 \). The FI of container \( A \) is \((c_1=0/1 + c_2=3/2)\) equal to 1.5, FI of container \( B \) is \((c_2=3/2 + c_3=0/2 + c_4=0/1 + c_5=1/2)\) equal to 2, and the FI of container \( C \) is \((c_3=0/2 + c_5=1/2 + c_6=1/2)\) equal to 1.

\[
FI(r) = \sum_{c \in M} \frac{f(c)}{\text{cont}(c)}
\]

Equation 5-2 Fan In (FI)

Mitchell and Mancoridis’s (2006) CF is calculated using Equation 5-3 where: \( r \) is the risk container; \( M \) is the set of all the member classes allocated to container \( r \); \( \text{depi}(c) \) is the number of outgoing dependencies of class \( c \) where the dependency is also a container member; and \( \text{depe}(c) \) is the number of outgoing dependencies of class \( c \) where the dependency is not also a container member. First, the number of dependencies between container members is summed, i.e. the coupling that is internal to the container, to give Internal Dependencies (ID). Then, ID is divided by itself plus half the number of dependencies from container members to classes that are not also container members, i.e. the External Dependencies (ED). For example, in Figure 5-1 there are three dependencies inside container \( B \): one between class \( c_3 \) and class \( c_2 \), one between class \( c_4 \) and class \( c_2 \), and one between class \( c_3 \) and class \( c_5 \). This gives an ID value of three. The classes in container \( B \) (\( c_2, c_3, c_4, \) and \( c_5 \)) are dependent upon only one external class, which is \( c_6 \), owing to the dependency from \( c_5 \) to \( c_6 \). That means the CF value for container \( B \) is 0.86 as shown in Equation 5-4:

\[
CF = \frac{ID}{ID + 0.5 \times ED} = \frac{3}{3 + 0.5 \times 1} = \frac{3}{3.5} = 0.86
\]

Equation 5-4 Cluster Factor Example Calculation
The high ratio between ID and ED is demonstrating that the classes in container B are highly cohesive, because as a collective, they require few external dependencies to fulfil their responsibilities. Unlike when calculating CBO and FI the internal and external dependencies are not divided by the number of containers of which the class is a member. That is because double counting does not apply given that CF is a relative measure of how cohesive a container is, whereas CBO and FI are absolute measures of how much coupling relates to the classes in the container. In its original form, CF is unsuitable for correlating to error-proneness because a higher CF value represents greater cohesion and should, therefore, be associated with less error-proneness. In order to test whether incohesive containers are more likely to be error-prone, Lack of Cluster Factor (LCF) is calculated by subtracting CF from 1 as per Equation 5-5.

\[ LCF (r) = 1 - CF (r) \]

**Equation 5-5 Lack of Cluster Factor (LCF)**

**MCC = e − n + 2p**

**Equation 5-6 McCabe Cyclomatic Complexity (MCC)**

McCabe defined Cyclomatic Complexity (MCC) as shown Equation 5-6 where: \( e \) is the number of edges; \( n \) is the number of vertices; and \( p \) is the number of connected components for a graph structure. There are five prerequisites for calculating MCC: (i) the dependency graph is directed; (ii) the graph must have a unique entry node; (iii) the graph must have a unique exit node; (iv) all nodes must be reachable from the entry node; and (v) the exit must be reachable from all nodes.

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

A=Aggregation, C=Composition, D=Dependency, G=Generalisation

**Figure 5-2 Design Structure Matrix of an Abstract Class and Sub-classes**
Risk containers do not necessarily satisfy all five criteria. For example, consider Figure 5-2 (an example of an abstraction design rule) which contains an abstract list and its concrete list sub-classes. Each concrete list class has the responsibility for providing list functionality for a specific database table. Other code may depend upon the specific concrete list classes rather than the abstract class. Representing A_4 as a graph would not satisfy the prerequisites owing to having multiple entry points (i.e., the concrete sub classes). Consequently, the container graphs must be transformed into strongly connected directed graphs before MCC can be calculated.

![Directed Graphs for Design Rule Container Examples](image)

Figure 5-3 shows the results of transforming the example Design Rule Containers shown in Figure 5-1 into directed graphs. Note: the graph edges are drawn in the direction of information flow (target to source) because if, for example, the state of class c2 changes in container A, then class c2 will be impacted. However, the MCC calculation produces the same result if the edge direction is reversed (source to target), which means that direction is unimportant so long as it is consistently applied, i.e. edges can be drawn in the direction of information flow or dependency between classes so long as all edges are consistent. The directed graphs that result for containers A and C are both compliant with all five criteria and are strongly connected. For example, container A is (i) directed (c2->c1), (ii) has a unique entry node (c2), (iii) has a unique exit node (c1), (iv) all nodes (c1) are reachable from the entry node (c2), and (v) the exit (c1) is reachable from all nodes (c2). The same is true for container C but not for container B. This is because container B has two entry nodes (c2, c5), two exit nodes (c3, c4), c2 cannot reach c5 and c5 cannot reach c4. Figure 5-4 illustrates the approach taken to ensure a risk container is always associated with a strongly connected graph.
The first step to transforming a container graph into a strongly connected graph is to introduce a hypothetical node $S$ to represent the rest of the external system. The second step is to connect $S$ to each entry point. The third step is to connect each exit point back to $S$. Note how the result of this transformation shown in Figure 5-4 results in container B’s graph becoming strongly connected because it has a unique entry and exit node ($S$), all other nodes can be reached from $S$, and all other nodes can reach $S$. The same transformation is applied to all risk containers, irrespective of whether their initial directed graphs were already strongly connected.

Having arrived at a strongly connected graph for each container, the formula shown in Equation 5-6 is used to calculate MCC. The strongly connected graph for container B shown in Figure 5-4 has seven edges and five nodes including the hypothetical node $S$. The number of components required by the formula is always one for risk containers because they do not contain nested sub-routines which is what $p$ represents according to McCabe (1976): “$p \neq 1$ can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines as shown above” (p. 314).

Table 5-1 Container level Cyclomatic Complexity Calculations for Figure 4-2

<table>
<thead>
<tr>
<th>Container</th>
<th>Members</th>
<th>$e$</th>
<th>$n$</th>
<th>$2p$</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>c1, c2</td>
<td>3</td>
<td>-</td>
<td>3</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>c2, c3, c4, c5</td>
<td>7</td>
<td>-</td>
<td>5</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c3, c5, c6</td>
<td>4</td>
<td>-</td>
<td>4</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 5-1 shows the results of calculating MCC for the example containers shown in Figure 5-1 using the transformed strongly connected graphs shown in Figure 5-4.
According to El Eman et al. (2001) class size in terms of lines of code can confound coupling metrics such as CBO when using them to predict error-proneness. El Eman et al. therefore recommend that future studies control for the effect of size when using CBO to predict error-proneness. However, lines of code cannot be determined from upfront design. Therefore, Design Classes (DC) which is equal to the number of classes in the container that are represented in the design is used to investigate if there is a similar confounding effect between container size and container level coupling metrics. In most cases DC is equal to the number of classes allocated to container \( r \). However, in the case of the API, DC counts the 1:1 interface and implementation pairs as 1 rather than 2 to ensure that metrics are normalized by container size correctly. DC is used in the calculation of Coupling Density Design (CDD, Equation 5-7) and Fan In Density Design (FIDD, Equation 5-8) which normalize CBO and FI by the number of classes in the container that are in the design (DC).

\[
CDD(r) = \frac{CBO(r)}{DC(r)}
\]

Equation 5-7 Coupling Density Design (CDD)

\[
FIDD(r) = \frac{FI(r)}{DC(r)}
\]

Equation 5-8 Fan In Density Design (FIDD)

Cluster Factor is not normalised by container size because the ratio between the number of internal and external dependencies is independent of the number of classes in a container. The formula for applying MCC to risk containers already takes the number of classes in the container into account (edges – nodes = internal dependencies – classes) and therefore MCC is not normalised by container size either. The decision to normalise CBO and FI, but not CF and MCC, by container size reflects the fact CBO and FI are absolute counts of coupling, whereas CF is a relative measure of cohesion and MCC is a relative measure of complexity.

The error-proneness of a risk container is calculated using bug spaces which were invented by Xiao, Cai, and Kazman (2014). A bug space is the subset of all nodes (classes) that contain at least \( N \) bugs. The error-proneness of a risk container can be determined by calculating the percentage of the bug space that the risk container’s classes occupy. The 75\(^{th}\) percentile of bugs per thousand lines of code (KLOC) defines the bug space threshold. This metric has been termed Bug Space Percent (BSP) and is calculated using Equation 5-9 where: \( r \) is the risk container; \( B \) is the set of all the classes in a particular software system that are in the upper quartile of bugs per thousand lines of code; \( M \) is the set of all the member classes allocated to container \( r \); \( cont(c) \) is
the number of containers that class \( c \) has been allocated to. The most error-prone risk container is the one with the highest occupancy of the bug space. When calculating BSP for risk containers, the class BSP is divided by the number of containers the class is a member of. Such division was also performed by Xiao, Cai, and Kazman (2014) and is necessary to ensure BSP is not double counted if a class is allocated to multiple risk containers.

\[
BSP(r) = \frac{1}{|B|} \sum_{c \in B \cap c \in E} \frac{1}{\text{cont}(c)}
\]

**Equation 5-9 Bug Space Percent (BSP)**

In order to have a fair correlation test for CDD and FIDD, which are the CBO and FI design metrics normalised by container size respectively, BSP is also normalised by container size to derive Bug Space Percent Density (BSD) using Equation 5-10 where DC is again the number of container \( r \) members that are in the design.

\[
BSD(r) = \frac{BSP(r)}{DC(r)}
\]

**Equation 5-10 Bug Space Percent Density (BSD)**

Having calculated the design metrics (CBO, FI, CF, MCC, CDD, FIDD) and implementation error-proneness (BSP, BSD) values for each container, Spearman’s (1904) rank correlation coefficient is used to test the hypothesis that tight coupling, lack of cohesion, and complexity should correlate to implementation error-proneness because developers are more likely to make mistakes when developing tightly coupled, incohesive, and complex programs than loosely coupled, cohesive and simpler programs.

For a type of risk container to isolate error-proneness it should exhibit: (i) lower Containers Per Class and a higher percentage of Internal Coupling (IC) to be considered element isolating; and (ii) a strong and significant correlation between the coupling (CBO, FI), cohesion (CF) and complexity (MCC) metrics calculated from the design and error-proneness to be considered risk predicting.

The control containers should not exhibit these qualities because they are based on random assignment as opposed to the coupling, lack of cohesion, and complexity that is hypothesised to cause error-proneness.

\[
CBOC(c) = \text{dep}(c)
\]

**Equation 5-11 Coupling Between Objects Classes (CBOC)**
\[ FIC(c) = f_i(c) \]

Equation 5-12 Fan In Classes (FIC)

\[ TBFC(c) = err(c) \]

Equation 5-13 Total Bugs Fixed per Class (TBFC)

The equivalent class level metrics are also calculated to establish a baseline with which to compare the predictive power of the container level design metrics. For the error-proneness baseline, Coupling Between Objects Classes (CBOC, Equation 5-11) and Fan In Classes (FIC, Equation 5-12) are correlated to Total Bugs Fixed per Class (TBFC, Equation 5-13) where: \( dep(c) \) is the number of outgoing dependencies of class \( c \); \( fi(c) \) is the number of incoming dependencies upon class \( c \); and \( err(c) \) is the number of times class \( c \) was changed to fix an error/defect/bug/fault. TBFC is used instead of Bug Space Percent occupancy because all error-prone classes would have equal occupancy of the bug space. Whilst this means the error-proneness baselines are not directly comparable to the container error-proneness predictions, they do establish the strength of the association between coupling and error-proneness in the base data set and can therefore be used as a baseline with which to compare the approach taken for risk containers. It was not possible to calculate a baseline for MCC because the internal structures of the classes were not documented in the designs. It was not possible to calculate a baseline for CF because it represents the cohesion of module and not a class.

5.2 Bug Space Coverage Results

It is important to consider how much of the bug space is covered by risk containers and the distribution of Bug Space Percentage (BSP) across the containers. The former is important because if containers created from the design don’t include the most error-prone classes, the goal of isolating error-proneness risks into containers is not met. The latter is important because if all risk containers typically covered the same proportion of the bug space, it would amount to all containers being equally error-prone. If the risk container population algorithms produced containers that were equally error-prone, they would not support the goal of identifying which parts of the architecture are riskier than others in terms of implementation error-proneness.
Table 5-2 Number and Coverage of Containers

<table>
<thead>
<tr>
<th>Name</th>
<th>DRCs</th>
<th>RCs</th>
<th>UCCs</th>
<th>Implementation Coverage (Size Percent)</th>
<th>Bug Space Coverage (Bug Space Percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>15</td>
<td>23</td>
<td>35</td>
<td>59.78 45.62 16.18</td>
<td>70.91 51.82 17.27</td>
</tr>
<tr>
<td>Server</td>
<td>9</td>
<td>14</td>
<td>68</td>
<td>12.50 9.91 6.47</td>
<td>13.79 10.36 5.20</td>
</tr>
<tr>
<td>Ps2tsa</td>
<td>11</td>
<td>7</td>
<td>10</td>
<td>79.07 34.88 32.56</td>
<td>81.81 54.54 45.45</td>
</tr>
<tr>
<td>OBTW</td>
<td>9</td>
<td>8</td>
<td>12</td>
<td>46.97 43.94 25.76</td>
<td>50.01 38.91 33.36</td>
</tr>
</tbody>
</table>

The implementation and bug space coverage results shown in Table 5-2 indicate that in three projects, the total UCC (API, Ps2tsa, and OBTW) and RC (API, Server, and Ps2tsa) bug space coverage is disproportionately high by comparison to implementation coverage, whereas DRC bug space coverage was disproportionately larger than implementation coverage in all four projects. However, bug space coverage is only marginally greater (<=2% greater) than implementation coverage for the API UCCs, Server DRCs, and Server RCs project. Having a greater bug space percent than size percent demonstrates that a disproportionately high amount of the error-proneness risk was isolated into some containers, which in turn demonstrates that the parts of the implementations of each project that were covered in the upfront design resulted in more bugs during development than the parts of the implementation that were not described in the upfront design.

The observation that in many cases the collection of risk containers isolates more than its fair share of the risk, leads naturally to the question of how that excess bug space coverage is distributed between individual containers. To answer that question Excess Bug Space Occupancy is used, and is defined as the percentage of containers that have a greater bug space coverage than implementation coverage. Table 5-3 presents Excess Bug Space Occupancy calculations per container type for each project.
Table 5-3 Percentage of Containers with Excess Bug Space Occupancy

<table>
<thead>
<tr>
<th>Container Type</th>
<th>API</th>
<th>Server</th>
<th>Ps2tsa</th>
<th>OBTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRC</td>
<td>53%</td>
<td>22%</td>
<td>27%</td>
<td>56%</td>
</tr>
<tr>
<td>RC</td>
<td>74%</td>
<td>36%</td>
<td>43%</td>
<td>13%</td>
</tr>
<tr>
<td>UCC</td>
<td>31%</td>
<td>25%</td>
<td>50%</td>
<td>67%</td>
</tr>
</tbody>
</table>

It can be observed in Table 5-3 that the excess bug space coverage is attributed to less than half of the containers in seven out of twelve cases. Less than 56% of DRCs contain the excess bug space in all four projects, which allied to the observation that DRCs covered only some of the implementation in all four projects (59.78%, 12.50%, 79.07%, 46.97%), is consistent with Xiao, Cai, and Kazman’s (2014) observation that most error-proneness is found within a few DRSpaces.

Figure 5-5 Distribution of Bug Space Percentage (x=mean marker)

Figure 5-5 shows the distribution of Bug Space Percentage (BSP) that the different container types are based on per project. Figure 5-5 shows relatively wide distributions around the mean for all risk container types, in all projects, except for the API and Server UCCs. These results indicate that the DRC, RC, and UCC population algorithms will result in containers that are likely to contain different proportions of the overall error-proneness risk. This, allied to the observation that some containers isolate a disproportionately large amount of the bug space.
(Table 5-3), suggests that all three types of risk container can be used to separate areas of relatively high error-proneness from areas of relatively low error-proneness.

5.3 Coupling Between Objects Results

Previous research has shown high levels of Coupling Between Objects (CBO) is predictive of error-prone classes (Basili, Briand, and Melo, 1996; Briand et al., 1998, 2000; Yu, Systa, and Muller, 2002; Subramanyam and Krishnan, 2003; and Gyimothy, Ferenc, and Siket, 2005). Therefore, it is hypothesised that containers encapsulating classes with high-levels of coupling should isolate error-proneness. Figure 5-6 shows the distribution of CBO values per container type for each project and the distribution of CBO per container type over all projects. The overall CBO mean of DRCs is 11.9, median of DRCs is 6.17, mean of RCs is 8.85, median of RCs is 6.86, mean of UCCs is 1.42, and median of UCCs is 0.56. Therefore, in the four projects studied, DRC and RC classes contain more than ten times as much outgoing coupling than UCC classes when considering the median.

![Figure 5-6 Distribution of Container Coupling Between Objects (x=mean marker)](image-url)
Table 5-4 CBO and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th></th>
<th>RC</th>
<th></th>
<th>UCC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>α</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.64</td>
<td>0.020</td>
<td>0.85</td>
<td>0.001</td>
<td>0.75</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.07</td>
<td>&gt;0.500</td>
<td>0.25</td>
<td>0.500</td>
<td>0.37</td>
<td>0.050</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>0.88</td>
<td>0.005</td>
<td>0.19</td>
<td>&gt;0.500</td>
<td>0.63</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.08</td>
<td>&gt;0.500</td>
<td>0.05</td>
<td>&gt;0.500</td>
<td>0.51</td>
<td>0.001</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.62</td>
<td>0.050</td>
<td>0.99</td>
<td>0.005</td>
<td>0.65</td>
<td>0.050</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.11</td>
<td>&gt;0.500</td>
<td>0.04</td>
<td>&gt;0.500</td>
<td>-0.16</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>0.51</td>
<td>0.200</td>
<td>0.98</td>
<td>0.001</td>
<td>0.42</td>
<td>0.200</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.43</td>
<td>0.500</td>
<td>0.08</td>
<td>&gt;0.500</td>
<td>0.14</td>
<td>&gt;0.500</td>
</tr>
</tbody>
</table>

Table 5-4 shows the results of performing a Spearman’s (1904) rank correlation between container CBO calculated from the design and implementation error-proneness. In all projects and for all container types, the correlation observed for the test containers is much stronger than the correlation observed for the control containers. In ten out of twelve cases the CBO correlations observed for the control containers are insignificant (α>0.05). This is to be expected because the control containers are based on random assignment, whereas the test containers cluster classes based on modularising design rules, resource usage, and use case fulfilment.

The CBO correlations for the test DRCs lie in the range between ρ=0.51 and ρ=0.88. According to Table 3-2, these values are classified as moderate (ρ=0.51), strong (ρ=0.62, ρ=0.64), and very strong (ρ=0.88) correlations between container level CBO calculated from the design and implementation error-proneness. The DRC correlations are significant by normal standards (α<=0.05) for all projects except OBTW. CBO correlation with error-proneness for the test RCs lies in the range between ρ=0.19 and ρ=0.99. These results represent very weak (ρ=0.19) and very strong correlations (ρ=0.85, ρ=0.98, ρ=0.99). The RC correlations are significant (α<=0.05) for all projects except Server. The UCC correlations between CBO and error-proneness are between ρ=0.42 and ρ=0.75 which are moderate (ρ=0.42) and strong (ρ=0.63, ρ=0.65, ρ=0.75), and are significant (α<=0.05) for all projects except OBTW.
The predicted Coupling Between Objects Classes (CBOC) and the Total Bugs Fixed per Class (TBFC) was computed using Equation 5-11 and Equation 5-13 respectively, to act as a baseline with which to compare the predictive power of the container CBO. The baseline results are shown in Table 5-5. The correlation between class CBOC and TBFC lies in the range between $\rho=0.28$ and $\rho=0.44$, but is only significant to the $\alpha=0.05$ level in two projects (API and Ps2tsa). Values in this range are classified as weak ($\rho=0.28$, $\rho=0.35$, $\rho=0.39$) to moderate ($\rho=0.44$). Comparison of Table 5-4 with Table 5-5 reveals that in eleven out of twelve cases, the correlation between test container CBO and error-proneness is greater than the equivalent correlation between CBOC and TBFC. The test container correlations are significant ($\alpha<=0.05$) in nine of the eleven cases where the test container correlation is greater than the equivalent class correlation. This suggests container-level CBO is a more effective predictor of error-proneness than CBOC which is class-level CBO.

### Table 5-5 CBOC and TBFC Baseline Metrics

<table>
<thead>
<tr>
<th>Name</th>
<th>CBOC-TBFC</th>
<th>$\rho$</th>
<th>$\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td></td>
<td>0.44</td>
<td>0.001</td>
</tr>
<tr>
<td>Server</td>
<td></td>
<td>0.35</td>
<td>0.100</td>
</tr>
<tr>
<td>ps2tsa</td>
<td></td>
<td>0.39</td>
<td>0.020</td>
</tr>
<tr>
<td>OBTW</td>
<td></td>
<td>0.28</td>
<td>0.200</td>
</tr>
</tbody>
</table>

Table 5-6 shows the correlations between Coupling Density Design (CDD), which is CBO normalised by the number of design classes in the container, and error-proneness. Unlike Lines of Code (LOC), the number of design classes in the container is calculable from the design. The correlations between CDD and error-proneness are lower than those observed for CBO for all container types and are only significant ($\alpha<=0.05$) in the Ps2tsa UCC and RC, and Server UCC cases. These results suggest that container size (design classes) confounds container CBO when used to predict error-proneness and that error-proneness predictions based on container CBO are potentially overestimated unless container size is considered.
Table 5-6 CDD and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th></th>
<th>RC</th>
<th></th>
<th>UCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.05</td>
<td>&gt;0.500</td>
<td>0.03</td>
<td>&gt;0.500</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.05</td>
<td>&gt;0.500</td>
<td>-0.02</td>
<td>&gt;0.500</td>
<td>0.21</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>0.55</td>
<td>0.200</td>
<td>0.15</td>
<td>&gt;0.500</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.14</td>
<td>&gt;0.500</td>
<td>0.02</td>
<td>&gt;0.500</td>
<td>0.57</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.20</td>
<td>&gt;0.500</td>
<td>0.88</td>
<td>0.050</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.03</td>
<td>&gt;0.500</td>
<td>-0.12</td>
<td>&gt;0.100</td>
<td>-0.10</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>0.18</td>
<td>&gt;0.500</td>
<td>0.51</td>
<td>0.500</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.16</td>
<td>&gt;0.500</td>
<td>-0.16</td>
<td>&gt;0.500</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Overall, the container CBO results suggest that: (i) design CBO is a reliable (moderate to very strong) predictor of implementation error-proneness for all container types (all cases except Server RCs); (ii) in three of four cases RCs have the strongest correlations between design CBO and error-proneness; (iii) container level CBO is more predictive of error-proneness than class level CBO; and (iv) the CDD results demonstrate that container level CBO correlation to error-proneness is confounded by container size and that CBO will overestimate error-proneness if DC is not taken into account.

5.4 Fan In Results

Figure 5-7 shows the distribution of Fan In (FI) values per container type for each project and the distribution of FI per container type over all projects. The overall FI mean of DRCs is 12.61, median of DRCs is 6.88, mean of RCs is 6.92, median of RCs is 3.95, mean of UCCs is 1.95, and median of UCCs is 0.06. Therefore, in the four projects studied, DRC classes contain more incoming coupling than RC classes, which in turn contain more incoming coupling than UCC classes. The observation that DRCs have more incoming coupling than UCCs is consistent with expectation because a purpose of modularisation is to serve many use cases with common modules, and DRCs represent common modules. Therefore, DRCs should typically have more incoming coupling than UCCs as observed in the four projects studied.
Figure 5-7 Distribution of Container Fan In (x=mean marker)

Table 5-7 FI and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th>RC</th>
<th>UCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>A</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.47</td>
<td>0.100</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.11</td>
<td>&gt;0.500</td>
<td>-0.03</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>0.91</td>
<td>0.005</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.56</td>
<td>0.200</td>
<td>0.52</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.79</td>
<td>0.010</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.11</td>
<td>&gt;0.500</td>
<td>0.37</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>0.42</td>
<td>0.500</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.25</td>
<td>&gt;0.500</td>
<td>-0.13</td>
</tr>
</tbody>
</table>

In all cases except Server RCs and UCCs, the correlation between FI and error-proneness in Table 5-7 is stronger for the test containers than it is for the control containers. In the cases where the test container correlation is stronger than the control, the correlations are weak ($\rho=0.27$), moderate ($\rho=0.42, \rho=0.47$), strong ($\rho=0.63, \rho=0.66, \rho=0.77, \rho=0.79$), and very strong ($\rho=0.91, \rho=0.94, \rho=0.98$). Except for API DRCs ($\rho=0.47$), OBTW DRCs ($\rho=0.42$), and OBTW UCCs
($\rho=0.27$), the correlations for test containers that outperform their corresponding control containers are significant ($\alpha<=0.05$). What can also be observed is the resemblance between Table 5-4 and Table 5-7, which indicates that the performance of FI (mean DRC FI=0.65, RC FI=0.65, and UCC FI=0.69) as a predictor of error-proneness is similar to that of CBO (mean DRC CBO=0.66, RC CBO=0.75, and UCC CBO=0.61).

The Fan In Classes (FIC) and the Total Bugs Fixed per Class (TBFC) was computed using Equation 5-12 and Equation 5-13 respectively, to act as a baseline with which to compare the predictive power of the container level FI. The baseline results are shown in Table 5-8. The Spearman’s (1904) rank results range between $\rho=-0.01$ and $\rho=-0.10$ suggesting there is no correlation between FIC and TBFC in the projects studied. These results suggest CBOC (Table 5-5) calculated from the design, is a better indicator of class implementation error-proneness than FIC (Table 5-8) calculated from the design, in the sample of projects used.

### Table 5-8 FIC and TBFC Baseline Metrics

<table>
<thead>
<tr>
<th>Name</th>
<th>FIC-TBFC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\rho$</td>
</tr>
<tr>
<td>API</td>
<td>-0.01</td>
</tr>
<tr>
<td>Server</td>
<td>0.09</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>-0.10</td>
</tr>
<tr>
<td>OBTW</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 5-9 shows the correlations between Fan In Density Design (FIDD), which is FI normalised by the number of design classes in the container, and error-proneness. The correlations between FIDD and error-proneness are lower than those observed for FI for all container types. That suggests container size (design classes) confounds FI when container FI is used to predict error-proneness, and that the risk of predicted error-proneness based on container FI will be overestimated unless container size is taken into account. However, this observation cannot be relied upon because the test container results for FIDD are only significant ($\alpha<=0.05$) in two of twelve cases for Server and Ps2tsa UCCs.
Table 5-9 FIDD and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th></th>
<th>RC</th>
<th></th>
<th>UCC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>α</td>
<td>α</td>
<td>α</td>
<td>α</td>
<td>α</td>
<td>α</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.05</td>
<td>&gt;0.500</td>
<td>0.07</td>
<td>&gt;0.500</td>
<td>-0.14</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.18</td>
<td>&gt;0.500</td>
<td>-0.07</td>
<td>&gt;0.500</td>
<td>-0.03</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>-0.07</td>
<td>&gt;0.100</td>
<td>-0.05</td>
<td>&gt;0.500</td>
<td>0.92</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.42</td>
<td>0.500</td>
<td>0.40</td>
<td>0.200</td>
<td>0.88</td>
<td>0.001</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.51</td>
<td>0.200</td>
<td>0.57</td>
<td>0.500</td>
<td>0.81</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.02</td>
<td>&gt;0.500</td>
<td>0.26</td>
<td>&gt;0.500</td>
<td>0.32</td>
<td>0.500</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>-0.15</td>
<td>&gt;0.500</td>
<td>0.51</td>
<td>0.500</td>
<td>0.31</td>
<td>0.500</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.04</td>
<td>&gt;0.500</td>
<td>-0.22</td>
<td>&gt;0.500</td>
<td>0.00</td>
<td>&gt;0.500</td>
</tr>
</tbody>
</table>

Overall, the container FI results suggest that: (i) design FI is also a predictor (moderate to very strong) of implementation error-proneness for all cases except Server RCs and OBTW UCCs; (ii) in two of four projects RCs and UCCs each have the strongest correlations between design FI and error-proneness; (iii) container level FI is more predictive of error-proneness than class level FI; and (iv) the FIDD results demonstrate that container level FI correlation to error-proneness is confounded by container size.

5.5 Cohesion Factor Results

Cohesion is a desirable property of software modules because it promotes understandability, reuse, and maintainability (Marcus, Poshhyvanyk, and Ferenc, 2008). Previous research has shown that lack of cohesion is predictive of error-proneness (Yu, Systa, and Muller, 2002; Gyimothy, Ferenc, and Siket, 2005). It is therefore hypothesised that more cohesive containers should be less error-prone and as such correlation between the Lack of Cluster Factor (LCF) and error-proneness of containers is expected.

Figure 5-8 shows the distribution of Cluster Factor (CF) values per container type per project as well as the distribution of CF per container type over all projects. DRCs are the most cohesive type of container in all four projects studied because the overall CF mean of DRCs is 0.75, median of DRCs is 0.84, mean of RCs is 0.52, median of RCs is 0.63, mean of UCCs is 0.22, and
median of UCCs is 0. DRCs are the most cohesive container type in all four projects, and RCs are more cohesive than UCCs.

Table 5-10 contains the correlation results between LCF and error-proneness. It can be seen in Table 5-10 that all of the LCF and error-proneness correlations except for Server UCC are insignificant (α>0.05). Furthermore, in four cases for DRCs, three cases for RCs, and two cases for UCCs, the correlation is negative which indicates LCF is associated with the absence of error-proneness. These negative correlations are inconsistent with the hypothesis that a lack of cohesion is more likely to result in error-proneness. The insignificance of the LCF results allied to the negative correlations observed suggests LCF is not a good predictor of error-proneness based on the four projects studied.
Table 5-10 LCF and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th></th>
<th>RC</th>
<th></th>
<th>UCC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
<td>α</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>-0.14</td>
<td>&gt;0.500</td>
<td>-0.64</td>
<td>&gt;0.500</td>
<td>0.02</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.01</td>
<td>&gt;0.500</td>
<td>0.03</td>
<td>&gt;0.500</td>
<td>0.22</td>
<td>0.500</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>-0.17</td>
<td>&gt;0.500</td>
<td>0.22</td>
<td>0.500</td>
<td>0.70</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.14</td>
<td>&gt;0.500</td>
<td>-0.10</td>
<td>&gt;0.500</td>
<td>0.61</td>
<td>0.001</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>-0.29</td>
<td>&gt;0.500</td>
<td>-0.16</td>
<td>&gt;0.500</td>
<td>-0.57</td>
<td>0.500</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.17</td>
<td>&gt;0.500</td>
<td>0.27</td>
<td>&gt;0.500</td>
<td>-0.36</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>-0.71</td>
<td>&gt;0.500</td>
<td>-0.45</td>
<td>&gt;0.500</td>
<td>-0.34</td>
<td>&gt;0.500</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>-0.08</td>
<td>&gt;0.500</td>
<td>0.22</td>
<td>&gt;0.500</td>
<td>0.04</td>
<td>&gt;0.500</td>
</tr>
</tbody>
</table>

Taking all of these observations into account suggests: (i) LCF is not a good predictor of error-proneness for any of the container types; (ii) DRCs are generally highly cohesive; (iii) UCCs are generally least cohesive; and (iv) the cohesiveness of RCs sits in between that of DRCs and UCCs. However, lack of cohesion is just one factor that can influence error-proneness and the CBO and FI results demonstrate container coupling is another factor. The next section considers the container complexity results to see if complexity is also a factor.

5.6 McCabe's Cyclomatic Complexity Results

As explained in Chapter 1, greater complexity increases the potential for programming errors. It is therefore hypothesised that containers with higher McCabe (1976) Cyclomatic Complexity (MCC) values should be more error-prone. Figure 5-9 shows the distribution of MCC values per container type for each project as well as the distribution of MCC per container type over all projects. The overall MCC mean of DRCs is 12.47, median of DRCs is 5.5, mean of RCs is 38.98, median of RCs is 40, mean of UCCs is 3.32, and median of UCCs is 2. The lower median of two for UCCs is expected because many Server use cases are fulfilled by just one package which gives them an MCC of two using the method applied. These results suggest that RCs are typically the most complex type of container, followed by DRC containers as typically the next most complex type, with UCCs being typically the least complex container type overall.
Table 5-11 MCC and Error-proneness Correlation

<table>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th>RC</th>
<th>UCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ρ</td>
<td>α</td>
<td>ρ</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.66</td>
<td>0.010</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.07</td>
<td>&gt;0.500</td>
<td>0.23</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>0.93</td>
<td>0.002</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.29</td>
<td>0.500</td>
<td>0.23</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.43</td>
<td>0.200</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.25</td>
<td>0.500</td>
<td>0.37</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>0.44</td>
<td>0.500</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.38</td>
<td>0.500</td>
<td>-0.08</td>
</tr>
</tbody>
</table>

Table 5-11 contains the correlation results between MCC and error-proneness. Table 5-11 reveals that: (i) DRC correlations were moderate (ρ=0.43, ρ=0.44), strong (ρ=0.66), and very strong (ρ=0.93), but only significant (α<=0.05) in the API and Server projects; (ii) RC correlations were very weak (ρ=0.16), strong (ρ=0.64), and very strong (ρ=0.83, ρ=0.98), and significant (α<=0.05) in all cases except the Server project; (iii) UCC correlations are moderate (ρ=0.43), strong (ρ=0.60, ρ=0.72), and very strong (ρ=1.00), and significant in all projects except OBTW;
and (iv) in all cases except Server RCs the test container correlations are stronger than the control container correlations.

Taking all these observations into account suggests: (i) RCs are typically the most complex container type; (ii) MCC is a good predictor of error-proneness for all container types; (iii) its predictive performance is marginally weakest for DRCs; and (iv) its predictive performance is strongest for RCs. These results are consistent with the hypothesis that greater complexity isolated within a risk container should result in greater error-proneness.

5.7 Validity

This section reviews the approach taken to determine whether risk containers can be used to predict error-proneness for construct validity threats using Messick's (1987) classification of consequential, content, substantive, structural, external, and generalizability threats.

5.7.1 Consequential

The number of times classes changed to fix bugs was determined by referring to the project’s defect tracking system and source code repository for the API, Server, and OBTW projects. For the Ps2tsa project, the developer had to retrospectively read the commit message and examine the changes to determine whether they were for bug fixing or feature development. Manual categorisation could introduce error into error-proneness results for the Ps2tsa project. This threat is acceptable due to the short amount of time that had passed between the developer writing the application and identifying the bugs, as well as the consistency of the Ps2tsa error-proneness results to the other projects.

5.7.2 Content

As explained in section 5.1 the bug space for each project was determined by identifying the classes in the upper quartile of bug fixes per class. Determining how many bugs have been fixed per class is subject to error because the difference between a change and bug fix can be subjective in the absence of a controlling specification that describes what the intended behaviour of the class is supposed to be. In the case of the industry projects (API and Server) this threat was mitigated because all software observations were sentenced by an observation review board whose responsibility included determining which observations were bugs and which were changes. In the case of the Ps2tsa project this threat was mitigated because each
change set was reviewed with the developer to identify those they considered to be bugs. In the case of the OBTW project it is clear from the categorisation of observations in GitHub that the team made efforts to separate bugs from changes as well.

5.7.3 External

The consistency of the Coupling Between Objects, Fan In, and McCabe Cyclomatic Complexity correlations to error-proneness all suggest that the error-proneness evaluation does have convergent, discriminant, and predictive qualities. That is expected because prior research (Munson and Khoshgoftaar 1992; Coleman et al., 1994; Basili, Briand, and Melo, 1996; Briand et al., 1998, 2000; Yu, Systa and Muller, 2002; Subramanyam and Krishnan 2003; Gyimothy, Ferenc, and Siket, 2005) has shown these standard software engineering metrics to be predictive of error-proneness. However, the Cohesion Factor error-proneness results were not convergent with the Coupling Between Objects, Fan In, and McCabe Cyclomatic Complexity results which is unexpected because cohesion is also known to be a predictor error-proneness (Briand et al. 2000; Gyimothy, Ferenc, and Siket, 2005). This lack of convergence represents an external validity threat to the Cohesion Factor results. However, this threat is insignificant with respect to the goal of this thesis because the results show that practitioners can use Coupling Between Objects, Fan In, and McCabe Cyclomatic Complexity to identify which risk containers isolate the most error-proneness instead.

5.7.4 Structural

Section 5.7.3 explains how the Cohesion Factor results could represent a threat to external validity because they do not converge with the coupling and complexity results as expected. This observation is also a threat to structural validity because cohesion, coupling, and complexity are different dimensions of the error-proneness test and although the Cohesion Factor does correlate with the construct of interest (i.e., Design Rule Containers are highly cohesive as expected) lack of cohesion did not correlate with error-proneness test scores as expected.

5.8 Chapter Summary

This chapter began by explaining in section 5.1, how metrics were calculated to represent design properties (incoming coupling, outgoing coupling, cohesion, and complexity) and implementation error-proneness to perform correlating testing using Spearman’s (1904) rank
correlation coefficient between those design metrics and implementation error-proneness for the risk containers extracted (contribution C5, Table 1-1).

Section 5.2 explained how the bug space was distributed over the different types of containers to determine whether some containers isolate more error-proneness than others. It was observed that error-proneness is not uniformly distributed and that some risk containers isolate more error-proneness than others. This observation is important because even distribution of error-proneness amongst containers would defeat the purpose of using containers to isolate areas of greater risk, whereas the uneven error-proneness distribution observed means that if a correlation between design metrics and error-proneness is also observed the metrics can be used to predict areas of greater error-proneness risk from designs.

Sections 5.3 to 5.6 used, respectively, Coupling Between Objects (CBO, outgoing coupling), Fan In (FI, incoming coupling), Lack of Cluster Factor (LCF, cohesion), and McCabe Cyclomatic Complexity (MCC, complexity) as error-proneness indicators. Correlations between CBO, FI, and MCC and error-proneness were observed for all three risk container types, which means that allied to the observation that error-proneness is unevenly distributed, CBO, FI, and MCC could be used as design time error-proneness predictors of differing strength.

With three very strong correlations and one very weak correlation, it was observed that CBO is most predictive for RCs (mean $\rho=0.80$). By comparison one moderate, two strong, and one very strong correlation were observed for DRCs (mean $\rho=0.66$), and one moderate and three strong correlations were observed for UCCs (mean $\rho=0.61$). When using Fan In the predictive power was found to be similar for all container types (DRC mean $\rho=0.65$; RC mean $\rho=0.65$; DRC mean $\rho=0.69$). Likewise, there was not much difference in predictive power when using Cyclomatic Complexity (DRC mean $\rho=0.62$; RC mean $\rho=0.65$; DRC mean $\rho=0.69$).

LCF calculated from the design is not a good predictor of implementation error-proneness. However, the container cohesion results (Figure 5-8) do suggest that DRCs are highly cohesive and more cohesive than RCs, which in turn are typically more cohesive than UCCs which are the least cohesive container type. The cohesion values shown in Figure 5-8 are reflective of the IC results (section 4.5), i.e. DRCs are the most cohesive container type because they have a higher proportion of internal coupling than RCs and UCCs. These findings support Wong et al.’s (2009) original goal for the Design Rule Hierarchy algorithm upon which DRCs are based, which was to separate areas of architecture into independent parts to increase developer parallelism.
The similarity of the CBO, FI, and MCC correlations is an example of multicollinearity. As explained by Munson and Khoshgoftaar (1992), multicollinearity is when multiple metrics all predict the same outcome. Munson and Khoshgoftaar observed that multiple different complexity measures were associated with error-proneness. As the multicollinearity observed is between outgoing coupling (CBO), incoming coupling (FI), and complexity (MCC) there is no way to know whether it is complexity, coupling, or both that results in the error-proneness. The multicollinearity between different error-proneness predictors (CBO, FI, and MCC) is expected because if risk containers have more coupling, they are also likely to be more complex, owing to the underlying graph structure of the container having more nodes and edges.

Crucially, it was observed that the correlations are much weaker in the control containers based on random assignment, which confirms that more highly coupled and complex designs do increase the risk of error-proneness. However, it was also observed that CBO and FI are confounded by container size, and LCF is not a good predictor of error-proneness for any of the container types. Overall, these observations suggest that Cyclomatic Complexity is the most reliable risk container metric for predicting error-proneness of those tested because it is not confounded by container size and its correlation to error-proneness is at least moderate in all cases except Server RCs, i.e. MCC correlation strength is moderate or greater in eleven of twelve container/project scenarios tested.

A supplemental observation is that the correlations observed between container CBO/Fl and error-proneness were stronger than the CBO/Fl correlations for individual classes. This supplemental observation provides some evidence to suggest that performing a risk container-based assessment of a design, i.e. using risk containers to isolate the areas of greatest risk, will be more effective than assessing the design from the perspective of individual classes.

Finally, the chapter provided a review of validity threats to the error-proneness correlation results. The evaluation of each container type’s ability to predict and isolate implementation error-proneness discussed in this chapter provides contribution C6 (Table 1-1).
Chapter 6 - Predicting Change Propagation

This chapter presents the metrics used to determine whether risk containers can be used to isolate and predict change propagation (contribution C7) and the results of applying those metrics to the four software projects studied (contribution C8). Altogether, Chapters 4 and 6 provide an answer to RQ1 for the risk of implementation change propagation. As explained in subsection 3.1.2, nonexperimental causal comparative correlation testing (Creswell and Creswell, 2018) was used to determine which container types are the most predictive of change propagation. This is achieved by calculating change propagation probability from the design (variable one) for each risk container, and checking if a correlation exists to implementation co-change for each container (variable two). Section 6.1 explains how change propagation probability and implementation co-change are calculated for risk containers. Inspection of the data sets used revealed an opportunity to collect additional evidence to confirm whether risk containers isolate change propagation. Firstly, by testing the hypothesis that on average co-change should be greater between classes that share containers than between those that do not share containers, and secondly by testing that sets of changes made to fix bugs and change the software fit neatly into risk containers. Section 6.1 also explains how the metrics were calculated to provide these additional pieces of confirmatory evidence. Section 6.2 presents the correlation results between change propagation calculated from the design and implementation co-change. Section 6.3 presents the results of checking whether co-change is greater between classes that share containers as expected. Section 6.4 presents the results of checking whether sets of changes do fit neatly into containers. Section 6.5 presents feedback from developers to determine if they agree with, and can account for, the correlation results presented in section 6.2. Having presented the detailed change propagation results, the chapter discusses validity threats in section 6.6, before summarising the change propagation results and drawing conclusions in section 6.7.

6.1 Change Propagation Metrics

Fregnan et al. (2019) proposed a coupling relation taxonomy in which coupling initially branches into structural, dynamic, semantic, and logical coupling. Due to the ripple effect (Lindvall, Tvedt, and Costa, 2003; Bass, Clements, and Kazman, 2012), it is hypothesised 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.
Therefore, the metrics used to predict co-change are based on structural coupling and not the other types of coupling identified by Fregnan et al. Dynamic coupling metrics were not used because the messages that are sent (as opposed to those that can be sent), between entities, depends on software usage patterns which are not known at design time. Semantic coupling was discounted because it would have increased experimental complexity and validity threats due to synonyms. Logical coupling was not used because it is based on change patterns, which cannot be known at design time. Thus, the metrics used to predict change propagation were based on structural coupling because it was considered the most reliable source of coupling information available in the software designs.

It is hypothesized that change propagation probability calculated from a design should correlate to change propagation during its implementation (co-change). To test this hypothesis, Spearman’s (1904) rank correlation coefficient is used, which is a statistical measurement of the association between two variables. As explained in subsection 2.2.2, Abdelmoez et al.’s formula requires the set of specific public attributes and function parameters of component B that are used by component A to be known, whereas Clarkson, Simons, and Eckert’s formula needs only the structural dependencies between components to be known. Abdelmoez et al.’s (2005) could not be used because the server project designs do not include Object Sequence Diagrams (OSDs) indicating usage of public attributes and function parameters of class (PL/SQL package) A by class (PL/SQL package) B, and some OSDs in the other projects do not indicate which methods are used in class interactions. However, all projects did have class diagrams that can provide the dependency component data. Therefore, Clarkson, Simons, and Eckert’s (2004) formula is used in favour of Abdelmoez et al.’s (2005) owning to the fidelity of the designs available for the Server, OBTW, and Ps2tsa projects.

The Clarkson, Simons, and Eckert formula (2004) shown in Equation 6-1 is applied 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 \( \rho(b, u) \) 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. Although Clarkson, Simons, and Eckert did not explicitly define the product index and limits in their equation, the product limit is the number of possible dependency paths between a and b.
Clarkson, Simons, and Eckert derived $\rho$ from historic change data. The non-zero implementation co-change, over all pairs of classes in the four software projects, is 0.46. Using a value derived from historic change data is inappropriate because the goal of this dissertation is to predict risks from upfront design and therefore a substitute for $\rho$ that is not based on historic change data must be provided. The value 0.5 was used as a hypothetical co-change probability between different classes because 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 Figure 6-1. 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, Simons, and Eckert change propagation probability would be calculated as shown in Equation 6-2:

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

Equation 6-2 Example Change Propagation Probability Calculation for Figure 6-1

If a direct relationship between classes c3 and c6 were added, the change propagation probability would change to be as shown in Equation 6-3:

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

Equation 6-3 Example Change Propagation Probability for Figure 6-1 (Impact of adding a direct Relationship)
The Change Propagation of the Design (CPD) for a container is calculated using Equation 6-4 and is the sum of the Clarkson, Simons, and Eckert change propagation probability over all pairs of distinct classes within the container where: \( r \) is the risk container; \( M \) is the set of all the member classes allocated to container \( r \); and \( clk(c1, c2) \) is the Clarkson, Simons, and Eckert change propagation probability between class \( c1 \) and class \( c2 \).

\[
CPD(r) = \sum_{c1 \in M, c2 \in M, c1 \neq c2} clk(c1, c2)
\]

Equation 6-4 Change Propagation Design (CPD)

The CPD calculations for the classes in Figure 6-1 are shown in Table 6-1, as well as fictitious data for observed co-change probability. Note that \( \rho(a,a) = 1 \) and that the Clarkson, Simons, and Eckert 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.

### Table 6-1 Co-change Probability for Figure 6-1

<table>
<thead>
<tr>
<th>Change Propagation Design (CPD)</th>
<th>Change Propagation Implementation (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>c1</td>
</tr>
<tr>
<td>c1</td>
<td>1.00</td>
</tr>
<tr>
<td>c2</td>
<td>0.00</td>
</tr>
<tr>
<td>c3</td>
<td>0.00</td>
</tr>
<tr>
<td>c4</td>
<td>0.00</td>
</tr>
<tr>
<td>c5</td>
<td>0.00</td>
</tr>
<tr>
<td>c6</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The Change Propagation of the Implementation (CPI) of a container is calculated using Equation 6-5 and is the sum of the actual co-change probabilities between each pair of distinct classes in the container where: \( r \) is the risk container; \( M \) is the set of all the member classes allocated to container \( r \); \( cochg(c1, c2) \) is the number of change sets involving both class \( c1 \) and
class c2; and $chg(c1)$ is the number of change sets involving class c1. 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 another class, to compute the observed co-change probability. For example, assuming the fictitious observed probabilities of Table 6-1, the CPI for the same container B is 2.93: the sum of the co-change probabilities between the classes are as follows: class c2 and the other class members of B (c3, c4, c5) is 0.24+0.24+0.18=0.66; class c3 and the other class members of B is 0.36+0.27+0.18=0.81; class c4 and the other class members of B is 0.40+0.30+0.10=0.8; and class c5 and the other class members of B is 0.33+0.22+0.11=0.66, for a total CPD value of 0.66+0.81+0.80+0.66 = 2.93.

$$CPI(r) = \sum_{c1, c2; c1 \in c2} \frac{cochg(c1,c2)}{chg(c1)}$$

Equation 6-5 Change Propagation Implementation (CPI)

Once the design and implementation change propagation values are calculated for each container, Spearman’s rank correlation coefficient is used to test the hypothesis: change propagation probability calculated from the design should correlate to observed co-change in the implementation. The equivalent class level metrics are also calculated to establish a baseline with which to compare the predictive power of the container level design metrics. For the change propagation baseline, Change Propagation Design Class (CPDC, Equation 6-6) is correlated to Change Propagation Implementation Class (CPIC, Equation 6-7). Again where: $clk(c1, c2)$ is the Clarkson, Simons, and Eckert change propagation probability between class c1 and class c2; $cochg(c1, c2)$ is the number of change sets involving both class c1 and class c2; and $chg(c1)$ is the number of change sets involving class c1.

$$CPDC(c1, c2) = clk(c1, c2)$$

Equation 6-6 Change Propagation Design Class (CPDC)

$$CPIC(c1, c2) = \frac{cochg(c1,c2)}{chg(c1)}$$

Equation 6-7 Change Propagation Implementation Class (CPIC)

If containers isolate the risk of change propagation, it would be expected that a class should be more likely to be changed with a class it shares a container with, than with one with which it does not share a container. To test this hypothesis, Containers in Common Co-change Probability (CCCP) and No Containers in Common Co-change Probability (NCCCP) are calculated.
It is to be expected that CCCP should be greater than NCCCP if a container type isolates the risk of change propagation.

CCCP is the mean co-change probability between classes that share one or more containers and is calculated using Equation 6-8 where: C is the set of all the classes in a particular software system; cic(c1) is the set of other classes that class c1 shares containers with (containers in common); cochg(c1, c2) is the number of change sets involving both class c1 and class c2; and chg(c1) is the number of change sets involving class c1.

\[
CCCP = \frac{\sum_{c1 \in C} \sum_{c2 \in cic(c1)} \frac{cochg(c1, c2)}{chg(c1)}}{\sum_{c1 \in C} |cic(c1)|}
\]

Equation 6-8 Containers in Common Co-change Probability (CCCP)

To demonstrate how CCCP is calculated, consider the example implementation co-change data shown in Table 6-1. Firstly, the sum of co-change probabilities for each class and the other classes with which it shares one or more containers is calculated. Using class c2 shown in Figure 6-1 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 6-1, gives 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 the mean co-change probability between classes that do not share any containers and is calculated in the same way as CCCP, but only probabilities between classes that do not share any containers are included. NCCCP is calculated using Equation 6-9 where: C is the set of all the classes in a particular software system; ncic(c1) is the set of other classes that class c1 does not share containers with (no containers in common); cochg(c1, c2) is the number of change sets involving both class c1 and class c2; and chg(c1) is the number of change sets involving class c1. The NCCCP for the example data in Table 6-1 and the containers shown in Figure 6-1 is 0.117. Mean and upper quartile values over each class are reported. The upper
quantile is used to confirm the mean. Only classes documented in the design are included in NCCCP and CCCP calculations.

\[
NCCCP = \frac{\sum_{c_1 \in c} \sum_{c_2 \in ncc(c_1, c_2)} \text{cochp}(c_1, c_2)}{\sum_{c \in c} |ncc(c)|}
\]

Equation 6-9 No Containers in Common Co-change Probability (NCCCP)

The 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, Isolated Change Sets (ICS) and Neatly Isolated Change Sets (NICS) were calculated. Container types with greater CCCP than NCCCP ought to also have greater ICS and NCC than other container types where the difference between CCCP and NCCCP is less. ICS is the percentage of Git/Subversion change sets that are a subset of one or more risk containers and is calculated using Equation 6-10 where: \(S\) is the set containing all the Subversion/Git change sets; and \(R\) is the set of all risk containers of a given type for a particular software system.

\[
ICS = \frac{|\{s \in S | \exists r \in R : s \subseteq r\}|}{|S|}
\]

Equation 6-10 Isolated Change Sets (ICS)

NICS is the percentage of Git/Subversion change sets that are a subset of exactly one risk container and is calculated using Equation 6-11 where \(S\) and \(R\) are the same as in ICS.

\[
NICS = \frac{|\{s \in S | \exists r \in R : s \subseteq r \text{ s.t. } |\{r \in R : s \subseteq r\}| = 1\}|}{|S|}
\]

Equation 6-11 Neatly isolated Change Sets (NICS)

In terms of the metrics defined, change propagation isolating container types should exhibit: (i) lower containers per class and a higher percentage of internal coupling to be considered element isolating; and (ii) a strong and significant correlation between internal change propagation probability calculated from the design and internal implementation co-change, a greater mean co-change between classes that share containers than between classes that do not share containers, and a higher percentage of change sets isolated into containers.

The control containers should not exhibit these qualities because they are based on random assignment as opposed to the coupling that is hypothesised to cause change propagation.
6.2 Change Propagation Correlation Results

Table 6-2 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 very strong and significant correlation ($\alpha \leq 0.010$) for all test container types when the classification scheme shown in Table 3-2 is applied. All control containers 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>
<thead>
<tr>
<th>Project</th>
<th>Containers</th>
<th>DRC</th>
<th>UCC</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\rho$</td>
<td>$\alpha$</td>
<td>$\rho$</td>
</tr>
<tr>
<td>API</td>
<td>Test</td>
<td>0.95</td>
<td>0.001</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.48</td>
<td>0.100</td>
<td>0.68</td>
</tr>
<tr>
<td>Server</td>
<td>Test</td>
<td>0.98</td>
<td>0.001</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.12</td>
<td>&gt;0.500</td>
<td>0.69</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>Test</td>
<td>0.98</td>
<td>0.001</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.21</td>
<td>&gt;0.500</td>
<td>0.39</td>
</tr>
<tr>
<td>OBTW</td>
<td>Test</td>
<td>0.95</td>
<td>0.001</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>0.53</td>
<td>0.200</td>
<td>0.68</td>
</tr>
</tbody>
</table>

The predicted change probability (using Clarkson, Simons, and Eckert’s formula) and the observed co-change (using the repository’s log) for each class was computed to act as a baseline with which to compare the predictive power of the container level change propagation design metrics. The baseline formulae for Change Propagation Design Class (CPDC) and Change
Propagation Implementation Class (CPIC) are shown in Equation 6-6 and Equation 6-7, respectively, and the baseline results are shown in Table 6-3.

Table 6-3 Class level CPDC and CPIC Baseline Metrics

<table>
<thead>
<tr>
<th>Name</th>
<th>CPDC-CPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ρ</td>
</tr>
<tr>
<td>API</td>
<td>0.39</td>
</tr>
<tr>
<td>Server</td>
<td>0.42</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>0.41</td>
</tr>
<tr>
<td>OBTW</td>
<td>-0.12</td>
</tr>
</tbody>
</table>

The correlation between CPDC and CPIC are consistent in three of the four projects (API, Server, and ps2tsa) which exhibit weak (p=0.37, p=0.39) to moderate (p=0.4) correlations that are significant (α<=0.05). A very weak negative correlation between CPDC and CPIC was observed in the OBTW project, which means there is no observable predictive effect between CPDC and CPIC in the OBTW project at the class level. Comparing Table 6-2 with Table 6-3 reveals that the class 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.

6.3 Containers in Common Co-change Results

If a risk container type is really isolating change propagation, then it is reasonable to expect that a class should be more likely to change with a class it shares a container with, than with one with which it does not share a container. Thus, in this section we check whether classes are more likely to change with other classes they share containers with to provide additional evidence that change propagation is being restricted within the extracted risk containers.

Table 6-4 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 the 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.

Table 6-4 Containers in Common Co-Change

<table>
<thead>
<tr>
<th>Project</th>
<th>Container Type</th>
<th>CCCP Mean</th>
<th>Q75</th>
<th>NCCCP Mean</th>
<th>Q75</th>
<th>Z Test Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>DRC</td>
<td>0.44</td>
<td>0.67</td>
<td>0.31</td>
<td>0.50</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>UCC</td>
<td>0.37</td>
<td>0.50</td>
<td>0.33</td>
<td>0.50</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td>RC</td>
<td>0.34</td>
<td>0.50</td>
<td>0.32</td>
<td>0.50</td>
<td>0.001</td>
</tr>
<tr>
<td>Server</td>
<td>DRC</td>
<td>0.52</td>
<td>0.67</td>
<td>0.39</td>
<td>0.67</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>UCC</td>
<td>0.32</td>
<td>0.44</td>
<td>0.42</td>
<td>0.67</td>
<td>0.100</td>
</tr>
<tr>
<td></td>
<td>RC</td>
<td>0.39</td>
<td>0.50</td>
<td>0.42</td>
<td>0.67</td>
<td>&gt;0.100</td>
</tr>
<tr>
<td>ps2tsa</td>
<td>DRC</td>
<td>0.57</td>
<td>0.92</td>
<td>0.25</td>
<td>0.40</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>UCC</td>
<td>0.51</td>
<td>0.77</td>
<td>0.29</td>
<td>0.50</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>RC</td>
<td>0.53</td>
<td>0.71</td>
<td>0.30</td>
<td>0.50</td>
<td>0.001</td>
</tr>
<tr>
<td>OBTW</td>
<td>DRC</td>
<td>0.33</td>
<td>0.43</td>
<td>0.30</td>
<td>0.43</td>
<td>0.050</td>
</tr>
<tr>
<td></td>
<td>UCC</td>
<td>0.38</td>
<td>0.57</td>
<td>0.30</td>
<td>0.43</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>RC</td>
<td>0.31</td>
<td>0.43</td>
<td>0.32</td>
<td>0.44</td>
<td>&gt;0.100</td>
</tr>
</tbody>
</table>

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, Tvedt, and Costa, 2003; Bass, Clements, and Kazman, 2012).

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

It is hypothesized (H3) in section 1.8 that if error-proneness and change propagation risks are isolated into Design Rule, Resource, and Use Case Containers, the set of files that must be changed to fix errors or implement changes should fit neatly into containers. Therefore, if change sets fit inside the risk containers of a given type, it would provide supporting evidence that the risk container type is helpful for isolating change propagation, to corroborate the correlation results in section 6.2.

Figure 6-2 shows Isolated Change Sets (ICS) values for the test DRCs are greater than those for the other container types in OBTW, but approximately equal highest in API, Server, and ps2tsa. In all projects, the test DRC ICS values observed 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 UCC ICS values, allied to the observation that in one project NCCCP was greater than CCCP for UCCs, casts doubt over whether they are really isolating change propagation, in support of Hassan and Holt (2004). 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 (α=0.001). In all projects the test ICS values observed are greater than their corresponding controls, albeit for all container types in the Server project, as well as OBTW RCs, the difference is very small.
Figure 6-3 Percentage of Neatly Isolated Change Sets

Figure 6-3 shows the Neatly isolated Change Sets (NICS), i.e. the change percentage of change sets that fit inside only one container, is greater for the test DRCs than the other container types in the API, Server, and OBTW projects. A z-test confirmed that the difference between the DRC and RC/UCC NICS means was significant in the API, Server and OBTW projects. In the Ps2tsa project test DRC and RCs have similar NICS values which are greater than UCC NICS. This suggests that DRCs are more effective than the other container types at neatly isolating change sets. However, it can also be observed in Figure 6-3 that in the Server project, the control DRC NICS is greater than the test DRC NICS, and in the OBTW project the test DRC NICS and control DRC NICS values are approximately the same. These results suggest that in the API and Ps2tsa projects, the DRC algorithm is more effective than the RC and UCC algorithms as well as random assignment. In the case of the Server and OBTW projects, the DRC algorithm fails to outperform random assignment for NICS.

When considered together, the ICS and NICS values suggest that DRCs are the most effective type of design risk container for isolating implementation co-change because: (i) test DRC ICS values are highest or approximately equal highest in all four projects; (ii) test DRC ICS values are greater than control DRC ICS values in all four projects; (iii) test DRC NICS values are highest in all four projects; (iv) test DRC NICS values are greater than control DRC NICS values in two of the four projects; and (v) the test DRC NICS values are much greater than test RC and UCC NICS values in three of four projects.
When taken into consideration with the class isolation results, which show that DRCs are more class and coupling isolating than RCs and UCCs, the change propagation results suggest DRCs are the most change propagation isolating container type. That is because the results show that DRCs have: 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 (i-v) are due to design dependencies.

6.5 Developer Feedback Results

Having identified DRCs as the most effective type of risk container for isolating change propagation, feedback from developers about DRCs was sought for two reasons: (i) to determine if developers recognize and can reason about risk containers, i.e. whether they were meaningful entities to them; and (ii) to corroborate that the containers predicted to contain the most risk did contain coupling problems that would account for the high level of change propagation (hypothesis H4). Developers of the API and Ps2tsa projects were available and a structured questionnaire was used to interview the available developers in order to collect their feedback.

Owing to the time taken to conduct interviews and the scarcity of the developers’ time it was decided to interview the developers only after the change propagation metric analysis had been completed and the leading container type had been identified. This represents a research limitation because feedback was only captured for Design Rule Containers. It is justifiable because the risk container proposition states that containers should be risk isolating and meaningful to developers which means if the metric analysis shows they are not risk isolating it doesn’t matter if they are meaningful to developers.

Prior to questioning, the concept of a risk container was explained to the developers using container diagrams such as those in Figure 4-2. During the interview the developers were allowed to refer to an un-ranked container summary as per the example shown in Table 6-5, as well as a full list of classes for each container.
Initially the developer was asked to name the containers most likely to contain classes that will have to be frequently changed together to add functionality or fix bugs (picking their top five). Only once they had nominated their top five, were the top five containers ranked by the metrics revealed to the developer. The developer was then asked to explain why they thought the classes ranked in the top five by the metrics would have to be frequently changed together to implement changes or fix bugs.

Table 6-6 shows that of the top five containers predicted to isolate the most change propagation by the CPD metric, the API developer agreed on four of them (albeit with a different ranking).

Both the API developer and CPD ranked A_4 as the highest container. The developer suggested A_4 has 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 API 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 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_10 and A_1 each having classes related to a different set of tables, the developer provided the same justification for both DRCs. 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 DRC 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”. DRCs were more effective at isolating the risk associated with these change scenarios than RCs, despite RCs being based on tables. That is because these scenarios impacted many RCs, as each one is based on a different table, whereas the risk was isolated into a single DRC.

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 DRCs 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 CPD metric captures such relationships, hence the agreement with the developer’s assessment.

The exercise was repeated with the student who developed the Ps2tsa project. Their top five DRCs 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 DRC ranked in 10th place, the developer explained they had nominated it because that DRC 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 correspondence between the metric rankings and the developer rankings for API and Ps2tsa projects provides some evidence to support hypotheses H2 and H4. That is because there was 80% agreement for the API and 60% agreement for Ps2tsa. Furthermore, the developer’s ability to recognise DRC contents and reason about the causes of coupling problems associated with them, suggests they are based on modularizing design rules as per Wong et al.’s (2009) Design Rule Hierarchy algorithm, i.e. the developer was able to comprehend them as decoupled ‘technical’ modules. This not only corroborates the risk isolation for change propagation results it also provides additional evidence that the DRCs are meaningful (useful) to developers and could be used to better manage mitigations (i.e. design reviews). Furthermore, analysing an upfront design using DRCs requires only class diagrams to be available, which according to Petre (2013) are the most common type of UML diagram used by the practitioners surveyed.

6.6 Validity

This section reviews the approach taken to determine whether risk containers can be used to predict change propagation for construct validity threats using Messick’s (1987) classification of consequential, content, substantive, structural, external, and generalizability threats.

6.6.1 Content

The change propagation results are based on the assumption that all classes committed together in a Subversion/Git change set required modification to fix a bug or implement a change, i.e. each Subversion/Git change set represents a discrete software change. Whilst committing changes discretely is best practice for working with these tools, it is possible that the software developers did not always follow best practice. Developers not following the best practice by committing several discrete changes together or committing the individual files that make up a discrete change separately, represents a content validity threat because the change
propagation results reported could contain error. This threat is negligible because: (i) the developers of the API and Server industry projects worked in accordance with a software development policy requiring them to commit changes discretely; (ii) in the case of the API, Server, and OBTW projects, the team’s development practices associated ticket (change) identifiers with each commit into the Subversion/Git source code repository; (iii) the change log comments indicate for all projects indicate that most commits represent discrete changes; and (iv) the Ps2tsa developer confirmed that although it is likely they did make some compound commits, they had followed the best practice in the main. Given that the minimum number of change sets was 105 in the OBTW project and much higher in the other projects (API=2179, Server=2202, and Ps2tsa=1040) it is reasonable to suggest that any error will affect a small proportion of the data set and therefore shouldn’t affect the conclusions presented.

6.6.2 Generalizability

In order to apply Clarkson, Simons, and Eckert’s (2004) method of calculating change propagation probability, a constant value of 0.5 was used in place of observed data. Whilst the correlation results in Table 6-2 suggest this was a reasonable value to use, whether it could be further optimized is unknown.

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, a Java parser to extract “design-like” relationships from source code was developed. The parser extracts relationships including: (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 (Softonic International S.A., 2020) which is a large Java project according to Zhang and Tan. The ACAT was then used to automatically populate 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 (1904) rank correlation between CPD and CPI is strong (p=0.93) and significant (α=0.001). These results are consistent with those for the other projects.

Whilst this is not a fully comparable test, because pattern and composition Design Rule Containers were not created, it provides early confidence that the method is 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.

6.6.3 Substantive

Integral to the proposition of making predictions about the change propagation and error-proneness qualities of a software implementation based upon its design is the assumption that the implementation is based upon the design. Section 4.6.1 explains the threat to consequential validity due to error in the manual transcription of design relationships. In addition to that threat, and as introduced in section 3.2, it is essential to have correspondence between the classes and relationships documented in the design, and those found in the implementation. Otherwise, a substantive threat exists to the theoretic foundation of predicting implementation error-proneness and change propagation based on design metrics. The coverage statistics shown in Table 4-2 provide some mitigation against this threat because they demonstrate that the design and implementation corresponded to some extent. However, it is also important to consider whether the relationships between those classes in the design exist in the implementation.

The strength and significance of the correlations between change propagation calculated from the design relationships and implementation co-change ($\rho \geq 0.85$ and $\alpha \leq 0.010$ for all projects and all container types) suggests that the design relationships do exist in the implementation. To further mitigate the threat of non-correspondence between design and implementation relationships, a Java program based on a Java parser (JavaParser, 2019) was developed to determine whether each design relationship listed in the relationship file (section 3.2.2) exists in the source code. The program revealed that Ps2tsa and OBTW source code included 95% and 63% of the design relationships respectively. The 37% of OBTW design relationships not found in the implementation related to classes evenly distributed across the risk containers. Therefore, these missing OBTW design relationships should not be unduly influencing the error-proneness and change propagation correlations. To confirm that is the case, the analysis tools were re-run without the missing design relationships. The results of the re-run show that the strength and significance of the error-proneness and change propagation correlations remain approximately the same which confirms the missing OBTW design relationships are not influencing the results. It was not possible to run the program over the API
and Server source code because their source code was not available. However, it is known that their implementations were rigorously based on the design and that the API’s initial class structure was created by generating source code from the UML design.

As acknowledged in section 4.6.2, designs that correspond to software implementations may not always be available which is why the automated container analysis tools described in section 4.3 were designed to be agnostic of how the class dependency graph is obtained. Having a process that is relationship source agnostic means the class dependency graph could, in theory, be reverse engineered from source code which means practitioners could perform a container-based risk assessment using source code rather than using upfront designs which are the subject of this dissertation. Indeed, the ArgoUML results presented in section 6.6.2 provide some evidence that the process will work when the class dependency graph is reverse engineered from source code.

6.7 Chapter Summary and Discussion

This chapter began by explaining in section 6.1, how metrics were calculated to predict container change propagation from the design, and actual container co-change from the implementation (contributions C7, Table 1-1), to perform correlating testing using Spearman’s (1904) rank correlation coefficient to determine whether change propagation probability calculated from the design is predictive of implementation co-change. Section 6.1 also explained how to calculate average co-change between classes that share containers and those that do not share containers, and the proportion of developer change sets that fit neatly into the risk containers extracted. The latter two calculations were anticipated to provide additional evidence to confirm whether risk containers do isolate change propagation.

Section 6.2 presented the results of correlating container change propagation calculated from the design to container co-change calculated from the implementation. It was found that all three risk container types are very strongly co-change predicting when Clarkson, Simons, and Eckert’s (2004) change propagation metric is used as the design time predictor. These strong experimental results justify choosing metrics based on structural coupling as co-change predictors instead of metrics based on the other types of coupling identified by Fregnan et al. (2019): dynamic, semantic, and logical coupling.

It was found that the correlations are much weaker in the control containers based on random assignment, which confirms that greater design coupling between container members
does increase the risk of implementation change propagation within containers. This observation is consistent with the error-proneness results which also found that control containers were less predictive than the test containers. It was also observed that when class level change propagation is correlated to class level co-change it is weaker than the equivalent correlation between container level change propagation and container level co-change. This observation is also consistent with the error-proneness results, and provides more evidence that performing a risk container-based assessment of a design, i.e. using risk containers to isolate the areas of greatest risk, will be more effective than assessing the design from the perspective of individual classes.

To corroborate the correlation results, section 6.2 then described the results of comparing the probability of classes changing together if they share one or more containers, to the probability of classes changing together if they do not share containers. In all four projects used in the evaluation, co-change is more likely between classes that share a Design Rule Container than between classes that do not share containers. The same is only true for Use Case Containers in three projects, and for Resource Containers in two projects. DRCs also have the highest probability of classes changing with other classes with which they share containers in three of four projects. This suggests change propagation risk between classes is being isolated into the test containers in support of the correlation test results and furthermore that it is most confirmatory for Design Rule Containers.

To further corroborate the correlation results, section 6.3 calculated the proportion of change sets isolated into containers because, assuming developers commit work into Subversion/Git per task, as is considered good practice, and as per the working practices of the projects used, the proportion of change sets that fit into risk containers based on design relationships is expected to be greater than those that fit into the control containers which are based on random assignment. The results show that the most or approximately equal most developer change sets fitted into specific Design Rule Containers in all four projects.

Altogether, these observations suggest that Design Rule Containers are the most effective of the three risk container types tested for predicting and isolating the risk of change propagation. That is because Design Rule Containers exhibited: (i) very strong and significant correlations between change propagation calculated from the design and implementation co-change; (ii) a higher probability of classes that share containers changing together than those that do not share containers in all four projects; (iii) the highest probability of classes that share containers changing together in three of four projects; and (iv) that the most or approximately equal most
developer change sets fitted into specific Design Rule Containers in all four projects. Design Rule Containers are, therefore, considered the most effective of the three risk container types tested because observation (ii) was true in fewer projects for Resource and Use Case Containers, and observation (iii) was not true for Resource and Use Case Containers.

Having identified Design Rule Containers as the leading container type, section 6.5 presented feedback from developers that indicates 80% (API) and 60% (Ps2tsa) agreement between the top five Design Rule Containers that the developers expected to isolate the most change propagation, and the top five Design Rule Containers predicted to contain the most change propagation by the metrics in section 6.2. In all cases the developers recognised Design Rule Containers as decoupled ‘technical’ modules and were able to explain why coupling isolated into the top five containers would result in excessive change propagation. This not only corroborates the change propagation metric results, but it also provides additional evidence that the Design Rule Containers are meaningful (useful) to developers and could be used to better manage mitigations (i.e., design reviews).

Finally, in section 6.6, the chapter presented the validity threats to the method used to determine whether risk containers are change propagation predicting. The evaluation of each container type’s ability to predict and isolate implementation change propagation discussed in this chapter provides contribution C8 (Table 1-1).
Chapter 7 - Container Usefulness Experiment

This chapter presents the method and results of an online experiment designed to determine whether smaller risk container diagrams help practitioners locate error inducing design flaws and impact changes more easily (contribution C9) in order to answer RQ2.

7.1 Experiment Design

The experiment is based around UML designs for a fictitious pet food e-commerce software system. The fictitious UML design has been divided into an equal number of smaller Design Rule Containers (DRCs), Use Case Containers (UCCs), and Resource Containers (RCs) using the container population rules described in section 4.1. Each of these three sets of containers represents a different experimental test group. A ‘no containers’ control group is used in addition to the three risk container type groups. Participants were recruited through the researcher’s network of industrial and academic colleagues, as well as callforparticipants.com. The research statement for the online experiment is shown in Appendix A.

The fictitious design is composed of 48 classes. Control group participants must reason about a single class diagram showing all 48 classes (Figure B-9-1 in Appendix B). For each risk container group (DRC, UCC, and RC) participants must reason about a set of nine smaller diagrams (one per container). On average each container diagram contains 8.33, 8.00 and 7.66 classes for DRCs, UCC, and RCs, respectively. This is to test the hypothesis that smaller diagrams, such as Design Rule, Resource, and Use Case Containers diagrams, should help practitioners find error-inducing design flaws and identify change impacts more easily, than a larger architecture diagram.

The experiment has three parts. In part one participants are asked the following questions about their practitioner and UML experience:

1. How many years of experience do you have in commercial software development (including architecting, designing, programming, testing)?
2. How many years of experience do you have working with Unified Modelling Language (UML)?
3. How many different projects that you have been involved with have used UML class diagrams?
4. How many different projects that you have been involved with have used UML sequence diagrams?

These questions are designed to provide context to each participant’s response. Part two of the experiment provides participants with an overview of the fictitious system, introduces the architecture review scenario, and explains how to take part using the instruction form shown in Figure 7-1.

---

EverSoft Inc. are a fictitious software development company that specialise in the development of bespoke web applications for their clients. They have been contracted to design and develop an e-commerce web site for a pet food company. The pet food company offers products for hamsters, rabbits, cats, and dogs. Their product range includes three main types of food: dry, wet and mixed. Not all types of food are available for all types of pet. The pet food company requires a new e-commerce website that enables their customers to perform the following goals:

1. A customer should be able to manage their account
2. A customer should be able to place orders for pet food
3. A customer should be able to enquire about how much food they should be feeding their pet based on the current age and weight of the pet.
4. Staff should be able to update product details.

The initial design for the system has been produced by a junior architect working for EverSoft. The principal architect has tasked you with reviewing the design prior to implementation. There are two things the principal architect has asked you to cover in your review:

1. The principal architect has recently read a computer science paper that suggests cyclic dependencies are likely to result in error-prone classes during implementation. A cyclic dependency is when two classes depend upon each other and **the principal architect would like you to find any cyclic dependencies that exist in the design**. The architect has sketched examples of cyclic dependencies below for you to refer to during your review so that you understand the kinds of things that you are looking for.

In the class diagram above there are two cyclic dependencies. The first one involves ClassA and ClassB because ClassA depends on ClassB due to a composition relationship and ClassB depends on ClassA due to an aggregation relationship. The second cyclic dependency involves ClassA and ClassD because ClassA has a dependency association to ClassD and a ClassD operation depends upon ClassA as a parameter type. You should be aware that sometimes the cyclic dependencies may span multiple diagrams, i.e. they can only be identified with reference to multiple diagrams.

2. The principal architect would also like you to **determine the impact of two changes** the pet food company have requested since the initial design was produced. The requested changes require you to:
   1. Determine all of the classes that would have to be changed if the return type of AbstractPetFoodCalculator were changed from int to float.
   2. Determine all of the classes that would be affected if class Logger were to be removed from the design.

The principal architect has provided you with all of the UML diagrams you will need to complete your review and a review form with which you can submit your review comments.

Please browse through the series of diagrams before attempting to answer all review questions on the last page. You will be able to navigate back and forth between the diagrams and the answer form until you are ready to submit your answers. You may find it helpful to print out these instructions to avoid having to navigate back to them during your review.

**Figure 7-1 Usefulness Experiment Instructions**
Participants play the role of peer reviewer to identify any cyclic dependencies and the impact of two change scenarios upon the design. Participants are provided with a reminder of what are cyclic dependencies, and how to identify them in class and sequence diagrams.

In part three, participants browse UML diagram(s) representing each risk container in their randomly assigned group (or control). They must identify the pair of classes involved in ‘planted’ cyclic dependencies and the classes impacted by two change scenarios. Participants are shown class diagrams for the Control, DRC, and RC groups. UCC group participants are presented with sequence diagrams. This reflects the types of diagrams used to create such risk containers as shown in section 4.1. Examples of the container diagrams shown to participants are illustrated in Figure B-9-2, Figure B-9-3, and Figure B-9-4 in Appendix B.

Cyclic dependencies are used as an example of an error-proneness inducing design flaw (Mo et al., 2015) to test whether presenting the design as a collection of risk containers helps or hinders practitioners to identify such potential error inducing flaws. If the results show that participants identify more of the planted cyclic dependencies in one of the risk container groups than the others (or control) it would suggest that presenting the design with (or without) smaller containers is helping the participants to locate the flaws (isolated risks). This observation would support the view that smaller container diagrams satisfy part M3 of the meaningful definition provided in section 1.3. The time taken to complete the experiment is also recorded to see if it varies between the different experimental groups. Two different timings were recorded for each participant: (i) the duration between first accessing and exiting the experimental platform, and (ii) the duration that each participant declared it took them to complete the experiment.

Wolfe and Horowitz (2017) explain that attention towards a search field of objects of interest is not random and is modulated by five factors. They assert that this is necessary because limits on visual processing make it impossible to recognize everything at once. These adaptations imply that the size and complexity of the search scene will influence how quickly an individual can locate an object of interest contained within it. It is hypothesised that it should be easier for practitioners to locate flaws that are contained within the smaller diagrams, such as single container diagrams, than the larger control diagram that shows the whole design.

The corollary is that it ought to be harder for practitioners to locate cyclic dependencies that are split across two smaller container diagrams due to them representing a larger (and split) scene. Therefore, practitioners are asked to identify three single-container cyclic dependencies for each of the four groups. Participants assigned to the DR and UC Container groups are also
asked to locate three cyclic dependencies that span multiple diagrams. It is impossible to ask
that for Resource Containers. That is because they are populated with classes that are
recursively dependent on the resource encapsulation class, i.e. if class A depends on class B and
class B depends on the resource encapsulating class C, all three classes would be members of
the resource container created from class C. This means that all classes with a direct or indirect
dependency on class C will be shown in the same Resource Container diagram. It is also
impossible to ask the control group about multiple-diagram dependencies because all classes
are shown on a single diagram. This difference in the number of review questions must be
accounted for when comparing the time taken to complete the experiment between the
different experimental groups. As such time per question is compared between participants
instead of time to complete the questionnaire.

Participants are also asked to assess the impact of potential changes. This is because if risk
collectors were to be adopted by practitioners to isolate risks, possible mitigations would
include redesign. If containers are found to aid practitioners with design refactoring, as well as
during the initial risk identification stage, it would provide further evidence that risk containers
are useful to practitioners. Participants are asked to identify the classes that would have to be
modified for one change that is isolated within a single container and a second change that
spans multiple container diagrams for all three risk container groups. Control group participants
are asked about both changes on the single class diagram.

Participants know they have identified a cyclic dependency as soon as a pair of co-
dependent classes have been found. For change impacts, participants must continue to search
even when an impact has been found to see if there are further impacts of the change.

By asking participants in different groups the same or similar questions, the experiment tests
whether presenting a design as a set of smaller diagrams, such as container diagrams, influences
the ability of practitioners to identify error inducing flaws and change impacts. The rationale
behind the experiment is that if smaller diagrams improve the participant’s performance when
executing these tasks by comparison to those using the overall diagram, smaller risk container
diagrams would satisfy part M3 (help partitioners locate find risk inducing architectural flaws
more easily) of the definition of risk containers being meaningful to practitioners presented in
section 1.3. When viewed in combination with the results of the metric analysis (Chapters 5 and
6) which test whether risk containers satisfy parts M1 (can be used to isolate real project risks)
and M2 (be formed from design artefacts used in practice) of that same definition, it can be
established whether any of the risk container types are meaningful to practitioners, which is the overall goal of this research.

The usefulness experiment concludes with a thematic analysis of the participants responses to the question: “do you have any further comments about completing this survey”? This question was asked so that a thematic analysis of the textual responses could be performed by following Braun and Clarke’s (2006) approach explained in subsection 3.1.3.

The participant responses are available for download from The Open University’s open research data repository using the following digital object identifier: 10.21954/ou.rd.14737967

7.2 Participant Experience Results

Figure 7-2 shows that participants have a median fifteen years of industry experience and seven years of UML experience. Figure 7-2 also shows that participants have typically worked on three projects using class diagrams, and two projects using sequence diagrams.

![Figure 7-2 Participant Experience (x=mean marker)](image)

7.3 Time Taken Results

Owing to random assignment to experimental groups, there were four control, four Design Rule Container (DRC), four Use Case Container (UCC), and three Resource Container (RC) submissions. Figure 7-3 shows how long it took participants in each group to answer review questions. The ‘Containers’ values are obtained by treating the eleven DRC, UCC, and RC submissions as a single group.
The median results suggest it took participants in the container groups approximately 33% longer per review question than those in the Control group (increased from 30% longer reported in Leigh, Wermelinger, and Zisman, 2019). It took DRC participants only 15% longer. Since all types of container have approximately the same number of classes per diagram, these results suggest participants can answer review questions quicker for DRCs than they can for RCs and UCCs. Overall, the results suggest that presenting the design as a collection of container diagrams increases the time needed to complete the review.

7.4 Cyclic Dependency Detection Results

The results for the accuracy with which participants can identify cyclic dependencies are shown in Figure 7-4. Note that there are no multiple diagram cyclic dependency questions for Resource Containers, as explained in section 7.1.
The results in Figure 7-4 indicate the ability of participants to correctly identify cyclic dependencies varies widely between container types, but the results are insufficient to render a compelling comparison between the container types due to the limited number of participants. However, the combined ‘Containers’ results drawn from eleven participants suggests that the smaller container diagrams do help practitioners identify cyclic dependencies contained within a single diagram. The median scores suggest participants in the container groups were typically twice as successful at identifying cyclic dependencies within single containers (median 67% correct) than participants in the control group (median 33% correct). By comparison, the mean scores are 50% correct for the control group and 72% correct for the containers group. Both the median and mean demonstrate that the containers group performed better which is consistent with the alternative hypothesis that it should be easier for participants to find these objects of interest in the smaller container diagrams. Performing a z-test to compare the proportion of correct answers provided by the control group, to the proportion of correct answers provided by the containers group, demonstrates that although these results are in favour of the alternative hypothesis, they are not significant enough to reject the null hypothesis (z=1.43 and z>=1.96 is needed for 95% confidence).

The picture differs for multiple-container cyclic-dependencies because presenting the design as a collection of containers produces approximately the same number of correct answers as the control. This is expected because participants must memorize diagrams as they switch between them. Participants allocated to the DRC group found it easier to identify multiple-container cyclic dependencies (median 50% correct) than those allocated to the UCC group (median 33% correct) and the control group (median 33% correct) using a single large diagram.

### 7.5 Change Impact Detection Results

Figure 7-5 shows how many change impacts were correctly identified in the different experimental groups. The median values indicate that containers (median 100% correct) have no bearing on the practitioner’s ability to detect change impacts over the control group (median 100% correct). This is expected because participants must continue to search the remainder of the design even when an impact has been found to see if there are further impacts of the change. 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 the single control diagram. The lower error frequency observed for the container participants may suggest cycling through smaller diagrams helps participants identify individual impacts more easily.
Thematic Analysis Results

Participants could leave comments about completing this survey prior to exiting the experimental platform. Thirteen of the fifteen participants left comments. A total of 19 different codes applied to the qualitative comments. Analysis of the coded text using Braun and Clarke’s (2006) approach explained in section 3.1.3 revealed the five themes shown in Table 7-1. Appendix C includes the mapping between the themes identified and the participant’s original comments.

<table>
<thead>
<tr>
<th>Theme Title</th>
<th>Description</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Support Needed</td>
<td>Participants suggested the need for tool support. E.g. ability to search for classes, navigate between diagrams (click through), usages, printing capability, and a cyclic dependency locator.</td>
<td>8</td>
</tr>
<tr>
<td>Navigability</td>
<td>Participants commented to say they found navigation within and between diagrams difficult.</td>
<td>5</td>
</tr>
<tr>
<td>Design Critique</td>
<td>Participants offered a critique of the synthetic system design.</td>
<td>4</td>
</tr>
<tr>
<td>Question Ambiguity</td>
<td>A participant found a specific question ambiguous.</td>
<td>1</td>
</tr>
<tr>
<td>Diagram Too Large</td>
<td>A participant commented to say they found the control diagram too large.</td>
<td>1</td>
</tr>
</tbody>
</table>
The question ambiguity and design critique themes support the validity of the usefulness experiment results. Having only one comment relate to the question ambiguity theme suggests that most participants were able to comprehend what they needed to do. This is consistent with the quantitative results for the percentage of cyclic dependencies and change impacts correctly identified. That is because if more participants had found the questions ambiguous, they would correctly identify fewer cyclic dependencies and change impacts. The ability of four participants to critique the synthetic system design adds credibility to the suitability of the participants for taking part in the usefulness experiment. Again, this is consistent with the quantitative data which reveals that participants had a median of fifteen years of industry experience and seven years of experience working with UML. Thus, only one participant commenting to say they found a question ambiguous, but four participants being able to offer design critique, allied to the experience levels of participants, supports the validity of the usefulness experiment results.

The remaining themes suggest practitioners will need tool support for risk containers to be most effective. The participant who found the diagram too large was a member of the control group presented with the whole design in a single diagram. Commenting that the control diagram is too large is supportive of viewing a design as a set of smaller diagrams, such as risk container diagrams. Five participants drawn from across all groups commented on the navigability of the experimental platform in terms of moving back and forth between diagrams and the question forms and comments from eight participants identified the need for more tool support to complete the design review activities. When considered as a collective, the ‘diagram too large’, ‘navigability’, and ‘tool support needed’ themes all suggest that for practitioners to make the best use of risk containers, they will need tool support, i.e. they should be implemented as views in architecture modelling tools such as Enterprise Architect (Sparx, 2020) and ArgoUML (Softonic International S.A., 2020). That is because if risk containers were available as a view in such tools, the features highlighted could be provided, including: the ability to search for classes, click between diagrams, see usages of classes, print diagrams, and automatically locate cyclic dependencies. The results of the thematic analysis therefore suggest that to fulfil their purpose and be useful for identifying and managing risks, practitioners will need tool support for risk container-based analysis.

7.7 Validity

This section reviews the approach taken to determine whether smaller diagrams, such as risk container diagrams, help practitioners locate error-inducing flaws and identify change

7.7.1 Consequential

The main consequential threat to the results of the usefulness experiment is the level of participation. With only 15 participants the trends are insignificant ($z=1.43$ and $z\geq1.96$ is needed for 95% confidence) and therefore more evidence is needed to know if they are consequential. Furthermore, participants were not timed under exam conditions and so it is unknown whether the time taken was unduly influenced by disruptions. However, there is no reason to think that one group would be more disrupted than another and so this threat would be mitigated by more participants.

7.7.2 Content

The experiment tests whether smaller diagrams such as risk container diagrams, help practitioners find error-inducing design flaws and identify change impacts more easily than larger architecture diagrams. This helps to compare the performance of participants performing these tasks using smaller diagrams to other participants using a much larger diagram, and between the three container types. However, the experiment does not test whether the risk container population algorithms are better or worse than other methods of splitting a larger diagram into a set of smaller diagrams. Whilst this is not a validity threat to the research question asked, it does represent a missed opportunity. If the experiment included additional groups in which participants are presented with smaller diagrams created with alternative methods of decomposition, the experiment may have produced additional results. These results could determine whether presenting designs as a set of smaller risk containers is more helpful to practitioners than other methods of design decomposition.

7.7.3 Generalizability

The usefulness experiment was based on just one error inducing flaw (cyclic dependency) and so the results of that experiment may not translate to other error inducing flaws such as implicit cross module dependencies and unhealthy inheritance (Mo et al., 2015). Testing with only one type of error inducing flaw therefore represents a generalizability threat to the results of the usefulness experiment. However, the threat is moot for implicit cross module dependencies because by their nature implicit dependencies are not detectable from the design.
This is a limitation of using risk containers to analysing upfront designs to predict and isolate risks, i.e. the technique cannot be used to find risks associated with implicit dependencies because they cannot be in the design. This threat is valid for unhealthy inheritance trees but there is no reason to suggest that participants wouldn't be able to locate unhealthy inheritance trees as easily as cyclic dependencies in class diagrams. That is because unhealthy inheritance trees are characterised by Mo et al. (2015) as either a sub-class having a dependency relationship to a super class, or a client depending on both super- and sub-classes, which in both cases participants should be able to observe in class diagrams.

A further generalizable threat to the results of the usefulness experiment is that participants were asked to analyse a relatively small synthetic architecture. Further work is necessary to determine whether the results obtained using the relatively small synthetic architecture are generalizable to larger real architectures.

7.8 Chapter Summary

This chapter began by explaining the design for an on-line experiment to test whether participants find it easier to locate error-inducing design flaws and change impacts using a collection of risk container UML diagrams, than when using a single larger control UML diagram. The purpose of the experiment is to determine whether practitioners would find designs presented as smaller risk container diagrams helpful to locate error inducing design flaws and impact changes more easily C9 (Table 1-1). After explaining the experiment design, the chapter then presented the participants’ experience levels to establish the context of the results. The median years of industry experience was fifteen and median years of UML experience was seven. This suggests the participants are qualified to take part in the experiment. Next, the experiment results were described which suggest that analysing a design as a collection of containers leads to more accurate detection of cyclic dependencies isolated within smaller container diagrams. The trade-off for improved accuracy is time. A general improvement was not observed for cyclic dependencies spanning multiple container diagrams, but an improvement for Design Rule Containers was observed. As for identifying the impact of design changes, the results suggest splitting an architecture into smaller containers has no bearing on the practitioner’s ability to detect change impacts correctly. Overall, the results of the usefulness experiment suggest all three-risk container types are equally useful to practitioners by offering accuracy improvements in detecting potentially error-inducing cyclic-dependencies. The chapter ended with a review of the experiment’s validity threats.


**Chapter 8 - Modelling Risks**

Having established that error-proneness and change propagation risks can be isolated into containers in chapters 5 and 6, we next consider how the output of a risk container analysis should be described so that the results can be presented alongside the architectural design. Figure 1-1 shows how the output from a risk container analysis should be stored in a risk register, and that a risk model could implement a risk register to support presentation of risk views to architects based on the contents of the risk model. Figure 1-1 also proposes that the risk model is a part of the architecture description so that the risks identified are traceable to the parts of the architecture from which they arise and impact.

The literature review presented in Chapter 2 demonstrates that many varied techniques exist for identifying risks associated with architectural designs. It is, therefore, advantageous to use a generic risk model that is not only capable of describing the output of a risk container analysis, but also the output of the other techniques reviewed in Chapter 2. That is because a technique agnostic risk model will enable practitioners to combine the results of multiple different risk identification techniques, including risk containers, into a single model to enable practitioners to view all risks associated with the architecture. Section 2.3 of the literature identified existing proposals for risk models. However, these models (Fabian et al., 2010; Mayer et al., 2019) are security specific and not suitable for describing the maintainability risks which are the subject of this dissertation.

A generic risk model that can hold the results of risk container analysis and other techniques is needed to support the process presented in Figure 1-1. It is the design of that risk model that is the subject of this chapter which presents the method and results used to derive and evaluate a risk model for software architectures (contribution C10) and answer RQ3 (what information is necessary in architecture risk models to describe outputs of architecture analysis techniques?).

**8.1 Proposed Risk Model Design**

This section explains the method used to design an architecture risk model and the standard concepts that are pertinent to an architecture risk model, before describing the proposed design for an architecture risk model to support risk views.

**8.1.1 Design Approach**
The UML activity diagram presented in Figure 8-1 illustrates how the overall method used to synthesise an architecture risk model involved three activities and this subsection explains each of those three stages.

The first activity (Figure 8-1, Activity 1) iteratively designed an architecture risk model based on existing concepts taken from ISO 42010 (2011) and ISO 31000 (2018), to describe the output of software architecture techniques found in the literature review. Basing the architecture risk model on the concepts defined in these standards (ISO 42010 and ISO 31000) promotes good practice in both the systems and software architecture, and risk management domains. This design was considered complete once the risk model could be used to describe the output of the architecture analysis techniques shown in Appendix D. Not only does Appendix D include the maintainability analysis techniques described in the literature review (Chapter 2), it also includes the discarded techniques that identify risks relating to other quality attributes such as efficiency, hazards, reliability, and security. Checking that the risk model design can describe the output of architecture analysis techniques for identifying risks relating to a broad range of quality attributes ensures that the risk model design is suitably generic. Subsection 8.1.3 presents the results of completing activity 1.

![Figure 8-1 Method used to synthesise an Architecture Risk Model](image)
The second activity (Figure 8-1, Activity 2) used a survey to collect practitioner feedback about the architecture design model designed in activity 1. The survey focused on whether practitioners think the risk model is applicable to Patterson and Neailey’s (2002) five stages of risk management when working with different approaches to systems and software development (Waterfall, Agile, and Scaled Agile), whether the practitioners prefer the risk model over plain text risk descriptions, and whether the practitioners can identify any missing concepts or redundant concepts. The survey design is described in subsection 8.2, the participant’s experience is described in subsection 8.3.1, the model applicability results are presented in subsection 8.3.2, and the approach preference, pros and cons results are presented in subsection 8.3.3.

The third activity (Figure 8-1, Activity 3) refined the risk model based on an analysis of the participants’ feedback gathered in activity 2. The improvements identified and the revised model are presented in subsection 8.4.7. Whilst it is acknowledged that a practitioner survey may not be the most rigorous method for evaluating the risk model, the goal of the method presented here was to mature the design of a risk model such that RQ3 (what kind of information is needed in an architecture risk model and risk views?) could be answered. As such, the scope of this dissertation concluded once activity three had been completed because the method is considered sufficient to establish the important kinds of information needed in an architecture risk model and risk views. Even if more rigorous evaluations such as a longitudinal practitioner study are undertaken in future work, the architecture risk model concepts and relationships (i.e., the kinds of information) presented here will still be needed.

8.1.2 Standard Concepts

This section briefly enumerates and explains the key concepts in ISO 42010 and ISO 31000 that are pertinent to an architecture risk model. That is because such a model should be based upon the international standards relevant to the disciplines of software architecture and risk management to support practitioners by promoting good practice. Both ISO 42010 and 31000 include a concept of risk: it is an explicit concept in ISO 31000, whereas it is an implicit sub-type of the Concern concept in ISO 42010. Table 8-1 provides a definition of the ISO concepts that are referred to later on in the architecture risk model design.
<table>
<thead>
<tr>
<th>ISO Concept</th>
<th>ISO Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD element</td>
<td>“any construct in an architecture description.” (ISO 42010, p. 7)</td>
</tr>
<tr>
<td>Architecture</td>
<td>“fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.” (ISO 42010, p. 8)</td>
</tr>
<tr>
<td>Architecture Decision</td>
<td>“pertain to system concerns; however, there is often no simple mapping between the two. A decision can affect the architecture in several ways.” (ISO 42010, p. 7)</td>
</tr>
<tr>
<td>Architecture Description (AD)</td>
<td>“work product used to express an architecture.” (ISO 42010, p. 2)</td>
</tr>
<tr>
<td>Architecture Model</td>
<td>“uses modelling conventions appropriate to the concerns to be addressed.” (ISO 42010, p. 6)</td>
</tr>
<tr>
<td>Architecture Rationale</td>
<td>“records explanation, justification or reasoning about architecture decisions that have been made.” (ISO 42010, p. 7)</td>
</tr>
<tr>
<td>Architecture View</td>
<td>“work product expressing the architecture of a system from the perspective of specific system concerns.” (ISO 42010, p. 2)</td>
</tr>
<tr>
<td>Architecture Viewpoint</td>
<td>“work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns.” (ISO 42010, p. 2)</td>
</tr>
<tr>
<td>Concern</td>
<td>“interest in a system relevant to one or more of its stakeholders.” (ISO 42010, p. 2)</td>
</tr>
<tr>
<td>Correspondence</td>
<td>“defines a relation between AD elements.” (ISO 42010, p. 7)</td>
</tr>
<tr>
<td>Correspondence Rule</td>
<td>“enforce relations within an architecture description (or between architecture descriptions).” (ISO 42010, p. 7)</td>
</tr>
<tr>
<td>Consequence</td>
<td>“Outcome of an event affecting objectives.” (ISO 31000, p. 2)</td>
</tr>
<tr>
<td>Control</td>
<td>“Measure that maintains and/or modifies risk.” (ISO 31000, p. 2)</td>
</tr>
<tr>
<td>Event</td>
<td>“Occurrence of change of a particular set of circumstances.” (ISO 31000, p. 2)</td>
</tr>
<tr>
<td>Likelihood</td>
<td>“Chance of something happening.” (ISO 31000, p. 2)</td>
</tr>
<tr>
<td>Model Kind</td>
<td>“conventions for a type of modelling.” (ISO 42010, p. 2)</td>
</tr>
<tr>
<td>Risk</td>
<td>“effect of uncertainty on objectives.” (ISO 31000, p. 1)</td>
</tr>
<tr>
<td>Risk Management</td>
<td>“coordinated activities to direct and control an organization with regards to risk.” (ISO 31000, p. 1)</td>
</tr>
<tr>
<td>Risk Source</td>
<td>“Element which alone or in combination has the potential to give rise to risk.” (ISO 31000, p. 1)</td>
</tr>
<tr>
<td>Stakeholder</td>
<td>“individual, team, organization, or classes thereof, having an interest in a system.” (ISO 42010, p. 2) “person or organisation that can affect, be affected by, or perceive themselves to be affected by a decision or activity.” (ISO 31000, p. 1)</td>
</tr>
</tbody>
</table>
8.1.3 The Risk Model Design

This section presents an architecture risk model that is based on the existing concepts in ISO 42010 and ISO 31000 and can be used to describe the output of the architecture analysis techniques found in the literature review (Appendix D). UML class diagrams are used throughout the remainder of this chapter to present the risk model design.

The model is based on the ISO 42010 Concern entity because it represents all stakeholder concerns including “the potential risks and impacts” (ISO 42010, p. 12). It would be inappropriate to add risk specific concepts to Concern directly because doing so would make Concern risk specific and ISO 42010 (p. 20) states: “Although concerns include risks and hazards (see 5.3), the term should not be understood to be synonymous with ‘risks’ or ‘worries’, but as referring to any topic of interest”. Therefore, the model is based on a new explicit sub-type of Concern, called Risk, to accommodate ISO 31000 risk specific concepts without making Concern risk specific, as shown in Figure 8-2.

![Figure 8-2 Grouping Risks and Relating them to Concerns](image)

All the techniques shown in Appendix D demonstrate that there is a relationship between risk and quality attribute satisfaction. Some of the techniques support the identification of risks relating to any quality attribute, for example Kazman et al.’s (1994) Scenario based Architecture Analysis Method (SAAM), and Clements’s (2000) Active Reviews for Intermediate Designs (ARID). Other techniques specialise in identifying risks relating to specific quality attributes, for example Bengtsson et al.’s (2004) Architecture Level Modifiability Analysis (ALMA), and Williams and Smith’s (1998) Software Performance Engineering. Irrespective of whether a technique can identify risks associated with any or with a specific quality attribute, it was found that all risks identified by the techniques can be framed from the perspective of other stakeholder concerns such as a quality attribute. Adding a parent relationship between two different concerns, as shown in Figure 8-2, enables the risk model to represent a hierarchy between concerns enabling quality attribute concerns to be related to more specific risk concerns. For example, if the architecture analysis identifies an error-proneness risk associated with a component that is
complex to maintain, because it is difficult to understand, parent would relate the error-proneness risk to the maintainability quality attribute concern.

Unified Modelling Language (UML) components and classes are examples of AD elements because UML component and class models are model kinds and ISO 42010 (p. 7) defines AD elements as follows:

“Every stakeholder, concern, architecture viewpoint, architecture view, model kind, architecture model, architecture decision and rationale (see 4.2.7) is considered an AD element. When viewpoints and model kinds are defined and their models are populated, additional AD elements are introduced.”

Some of the techniques can identify the subset of AD elements that may be inducing the identified risk. Examples include Stevanetic and Zdun’s (2016) approach to calculate component understandability metrics from UML designs, and Shaik, Abdelmoez, and Ammar’s (2008) Software Architecture Risk Assessment (SARA) tool which calculates the relative change and error propagation probabilities of components based on the research of Goseva-Popstojanova et al. (2003) and Abdelmoez et al. (2005, 2006). Architecture elements that give rise to risk align to the risk source concept in ISO 31000. This suggests a model should enable AD elements to be related to risks in order to specify them as being risk sources in ISO 31000 terms. Therefore, a relationship called risk source connects Risk to AD Element as shown in Figure 8-3. Continuing the previous example, risk source would connect the error-proneness risk to the component that is difficult to understand in order to denote that the component that is difficult to understand as being the ISO 31000 risk source. The cardinality of the risk source relationship at the AD Element end must be zero or more because not all techniques identify specific AD element(s).

When multiple instances of a common risk are identified, some techniques can indicate the relative risk of those instances. For example, SARA uses quantitative metrics to rank components
for relative risk, whereas Folmer, van Gurp, and Bosch’s (2003) Scenario based Architecture Level Usability Assessment (SALUTA) identifies usability risks by comparing required and provided usability qualitatively for an architecture. **Indicator Value, Indicator, is for, and indicated by**, as shown in Figure 8-4 are introduced in order to enable such architecture analysis outputs to be represented in the model. **Indicator** serves to denote the type of risk indicator used, e.g. a particular software engineering metric such as Coupling Between Objects (Chidamber and Kemerer, 1994) or Cyclomatic Complexity (McCabe, 1976). **Indicator Value** represents the value of a given **Indicator** for a particular risk. The relationship cardinality at the **Indicator** end must be one to denote the **Indicator** to which the **Indicator Value** pertains. The cardinality at the **Indicator Value** end must be zero or more to one to allow for the situation when an **Indicator Value** has not yet been collected. **Indicated by** relates the **Indicator Values** back to the **Risk** they identify.

![Figure 8-4 Specifying Risk Indicators](image1)

Some **Indicators** might be reusable for different risks and concerns. For example, a coupling metric such as CBO might be applicable to different maintainability risks such as change-proneness and error-proneness. Therefore, the **indicates** relationship connects **Indicator** to **Concern** in order to denote the concerns the Indicator can be used to indicate.

![Figure 8-5 Support for Risk Containers](image2)

The **risk source** and **indicated by** relationships are of particular importance to this dissertation because a **risk source** relationship would be created for each class allocated to a particular Design Rule, Resource, or Use Case Container extracted using the algorithms described in section 4.1, and an **indicated by** relationship would be created for each design.
metric calculated to indicate the relative error-proneness/change propagation risk of the classes isolated into the container. Thus, a risk container is represented in the model by a Risk, the AD Elements related to it using risk source, and the Indicator Value(s) related to it using indicated by as illustrated by Figure 8-5.

The reason for analysing architectures to identify risks is to understand the potential risk impacts so that they can be mitigated. ISO 31000 already includes a concept called consequence to represent the possible impact of risks happening. There are two reasons why ISO 42010’s Architecture Rationale entity is insufficient to describe impacts of risks. First, risk impacts may not necessarily be a consequence of architecture decisions: they could stem from a requirement or constraint imposed upon the system described by the AD. Second, whilst an Architecture Rationale might describe how a decision aims to support such requirements and constraints, it would not describe the impact of failing to meet them. For example, Architecture Rationale might capture how a decision, e.g. the choice of a design pattern, aims to make a system maintainable, but Consequence is needed to describe the consequences if a risk analysis technique identifies that the decision has not fulfilled its aim, i.e. the consequence in ISO 31000 terms. A risk might impact specific AD elements in some cases. For example, suppose that an AD describes how components support use cases. Using SARA to identify components associated with higher levels of change propagation would reveal the affected use cases by reference to the component usages of each use case. The relationship between Consequence and the affected AD elements shown in Figure 8-6 enables the risk consequences to be described in terms of any impacted AD elements, e.g. the use cases in this example. A further relationship from Consequence to Architecture Rationale called contradicts is introduced when the rationale behind an architecture decision becomes suspect due to the potential risk identified. Appendix D enumerates a number of techniques that can identify risk consequences in terms of affected architecture elements (Kazman et al., 1994; Williams and Smith, 1998, 2002; Kazman, Klein, and Clements, 2000; Singh et al., 2001; Folmer, van Gurp, and Bosch, 2003; Goseva-Popstojanova et al., 2003; Bengtsson et al., 2004; Abdelmoez et al., 2005, 2006; Hassan, Goseva-Popstojanova, and Ammar, 2005; Liu, Dehlinger, and Lutz, 2007; Mustafiz et al., 2008; Cancila et al., 2009; Bernardi, Merseguer, and Petriu, 2011; Faniyi et al., 2011; Alberts, Woody, and Dorofee, 2014). These analysis techniques can be scenario or use case driven which means it is the scenario and use case architecture elements that are impacted if the risk becomes an issue.

In addition to consequences, ISO 31000 also has a Control concept to represent measures to maintain or modify the risk. Some architecture analysis techniques such as Alberts, Woody, and Dorofee’s (2014) Security Engineering Risk Analysis (SERA) approach include steps for developing
a control plan. The requirements elicited so far would not support developing a control plan (control in ISO 31000 terms). If the control plan requires that an architecture decision be made or revised, it would be important to record the rationale behind the decision in terms of mitigating the impact of the risk identified. Support for controls is achieved by including the ISO 31000 **Control** concept. **Controls** might only be effective for some of the many potential **Consequences** of a **Risk**. That suggests **Control** should be connected to **Consequence**. Alternatively, some **Controls** may have a reducing effect on more than one **Consequence** which suggests they could be generic. Adding a relationship from **Risk** to **Control** as shown in Figure 8-6 denotes the reusability of controls against the different potential consequences of a **Risk** instance. A relationship called **justifies** from **Control** to **Architecture Rationale** conveys the rationale behind architecture decisions that have been made to mitigate the risks identified.

![Figure 8-6 Relating Impacts and Mitigations to Risks](image)

Figure 8-7 illustrates the complete risk model and Table 8-2 provides a definition for each new entity and correspondence as well as specifying which items support each requirement and ISO 31000 risk management concept. The only ISO 31000 concepts that are not included in Table 8-2 are **Risk Management**, **Event**, and **Stakeholder**. However, as shown in Figure 8-7, the architecture risk model already inherits the **Stakeholder** concept from ISO 42010. Furthermore, the events that change the circumstances described by the architecture description are **Architecture Decisions** made by architects, and **AD Element** can be used to describe events that occur as the system described by the architecture description is operated. As such support for the **Event** concept is met due to inheriting the **Architecture Decision** and **AD Element** concepts from ISO 42010 as well. That leaves only **Risk Management** which is not needed in the model because it represents the overall process context in which the risk model might be used.
Table 8-2 Definition of new Items proposed for an Architecture Risk Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition</th>
<th>ISO 31000 Concept(s) Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk</td>
<td>Entity</td>
<td>Sub type of Concern that represents a Risk, e.g., error-proneness or security vulnerability.</td>
<td>Risk</td>
</tr>
<tr>
<td>parent</td>
<td>Correspondence</td>
<td>Allows a Risk to be related to a more general parent Concern (or a Concern to another Concern). For example, if a stakeholder were concerned about maintainability, they are implicitly concerned about the risk of error-proneness. This relationship allows finer grained risks to be related to more general concerns in a hierarchy.</td>
<td></td>
</tr>
<tr>
<td>risk source</td>
<td>Correspondence</td>
<td>Specifies the ‘risky’ AD elements giving rise to the Risk, i.e. the scope to which the architecture analysis technique could isolate the Risk.</td>
<td>Risk Source</td>
</tr>
<tr>
<td>Indicator</td>
<td>Entity</td>
<td>Indicates the relative risk level. An Indicator could be a quantitative software engineering metric such as a coupling measure or a qualitative assessment by an architect.</td>
<td></td>
</tr>
<tr>
<td>Indicator Value</td>
<td>Entity</td>
<td>The value of a particular Indicator for a particular Risk.</td>
<td></td>
</tr>
<tr>
<td>indicated by</td>
<td>Correspondence</td>
<td>Associates specific risk indicator Values to particular Risks.</td>
<td></td>
</tr>
<tr>
<td>is for</td>
<td>Correspondence</td>
<td>Identifies the Indicator type of an Indicator Value.</td>
<td></td>
</tr>
<tr>
<td>indicates</td>
<td>Correspondence</td>
<td>Identifies which Indicators indicate different Concerns.</td>
<td></td>
</tr>
<tr>
<td>Consequence</td>
<td>Entity</td>
<td>Represents a potential consequence of a Risk being left untreated. According to Alberts (2006) the Consequence should be specified in monetary terms and include a notion of likelihood to support ISO 31000.</td>
<td>Consequence, Likelihood</td>
</tr>
<tr>
<td>results in</td>
<td>Correspondence</td>
<td>Identifies the potential Consequences of a Risk.</td>
<td></td>
</tr>
<tr>
<td>affects</td>
<td>Correspondence</td>
<td>Identifies any AD elements that are affected by the risk</td>
<td></td>
</tr>
</tbody>
</table>
Additional entities have been included to allow recording the risk Analysis Technique(s) used and the Analysis Results that led to the identification of Risks associated with the AD. The Analysis Technique entity provides traceability between the method and the results of risk analysis to enable efficacy comparisons between the different techniques. Analysis Results enables different versions of the same risk to be compared before and after architectural decisions, including those made to mitigate the identified risks, to allow architects to understand whether their decisions are improving the current architecture.

The proposed model supports Patterson and Neailey’s five stages of iterative risk management through contradicts and justifies relationships (Figure 8-6): they establish a feedback loop between Architecture Rationale and Risk (via Consequence and Control) as illustrated by the activity diagram presented in Figure 8-8.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>contradicts</td>
<td>Identifies Architecture Rationale that is contradicted by the Consequence of a Risk</td>
<td>Correspondence</td>
</tr>
<tr>
<td>reduced by</td>
<td>Identifies the Mitigations that could be used to reduce consequences.</td>
<td>Correspondence</td>
</tr>
<tr>
<td>Control</td>
<td>An action that could reduce the potential Consequences of a Risk.</td>
<td>Entity</td>
</tr>
<tr>
<td>mitigates</td>
<td>Identifies the Consequences that can be mitigated by Controls.</td>
<td>Correspondence</td>
</tr>
<tr>
<td>justifies</td>
<td>Identifies Architecture Rationale that control Risks</td>
<td>Correspondence</td>
</tr>
<tr>
<td>Analysis Technique</td>
<td>Identifies the architecture analysis technique used to for a risk analysis</td>
<td>Entity</td>
</tr>
<tr>
<td>input</td>
<td>Identifies the AD upon which the analysis technique was used to perform a risk analysis</td>
<td>Correspondence</td>
</tr>
<tr>
<td>produces</td>
<td>Identifies the results of the risk analysis performed by an analysis technique</td>
<td>Correspondence</td>
</tr>
<tr>
<td>Analysis Results</td>
<td>Encapsulates the results of a risk analysis performed using an analysis technique</td>
<td>Entity</td>
</tr>
<tr>
<td>identifies</td>
<td>Denotes the risks that were identified in a set of analysis results produced by applying an analysis technique to an AD</td>
<td>Correspondence</td>
</tr>
</tbody>
</table>
8.1.4 Novel Elements

Table 8-3 presents a comparison between the proposed risk model shown in Figure 8-7 and existing risk concepts and models (ISO 31000, 2018; Fabian et al., 2010; Mayer et al. 2019). Whilst Table 8-3 shows that all models have analogues of the key elements (Risk, Control, and Consequence), there are clear differences between the existing and proposed models. It is in those differences where the novelty of the proposed model lies.

The first difference is that the proposed model supports a hierarchy of risks because the parent relationship between Concerns captures the hierarchy. That means the proposed model can describe for example, that error-proneness and change propagation are sub types of maintainability risk. Whilst it is not surprising that the Fabian et al. and Mayer et al. models do not support such a hierarchy, because they are security risk models, it seems limiting that ISO 31000 doesn’t recognise that risks could be classified especially considering the ISO 31000 principle that risk management should be structured and comprehensive.

The second difference is that the proposed risk model supports relating different indicator values to different risks which enables the significance of different risks to be compared. Although the Fabian et al. and Mayer et al. include Vulnerability which is a security specific proxy for indicator value, they do not relate such indicator values to the risk type as achieved by the indicates relationship in the proposed risk model. Again, this is not surprising because their model is security specific.
The third difference is that only the proposed risk model recognises that mitigations could be impact specific and that a risk could result in many impacts (through the mitigates relationship from mitigation to impact). This is a significant limitation of the Fabian et al. and Mayer et al. models because there is no way to know which mitigations mitigate particular impacts when there are many impacts. This could result in a specific impact being overlooked if the risk manager wrongly assumed all impacts are mitigated by the security requirements modelled.

The fourth difference is that only the proposed risk model provides traceability between the techniques used to identify risks and the risks identified. Having traceability between the analysis technique used and the risks the technique has identified offers an advantage with respect to upholding the ISO 31000 principle that risk management should include continual improvement, i.e. it enables the results produced by different analysis techniques to be compared in terms of efficacy. Such comparisons would enable managers to assess which techniques are more effective in order to improve the effectiveness of their risk management practices.

The fifth and most significant difference with respect to the overarching goal of this thesis is that the proposed risk model integrates ISO 42010 and ISO 31000 to show how architecture description and risk management concepts are related. It is this property of the proposed model that provides guidance to architects about how to model risks relating to architectures and answer RQ3: what information is necessary in architecture risk models to describe outputs of architecture analysis techniques? In summary, the proposed model supports tracing architecture elements to risks, their impacts, and the decided mitigations, as well as providing a feedback loop to either justify or contradict rationale for architecture decisions.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk</td>
<td>Risk</td>
<td>Risk</td>
<td>Risk</td>
</tr>
<tr>
<td>parent</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Indicator</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Indicator Value</td>
<td>Not supported</td>
<td>Vulnerability</td>
<td>Vulnerability</td>
</tr>
<tr>
<td>indicated by</td>
<td>Not supported</td>
<td>Risk : Security Property : potentially violated by : Vulnerability</td>
<td>Characteristic of</td>
</tr>
<tr>
<td>is for</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>indicates</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Consequence</td>
<td>Consequence</td>
<td>(Potential) Loss</td>
<td>Impact</td>
</tr>
<tr>
<td>results in</td>
<td>Implicitly supported (Impact)</td>
<td>Suffers</td>
<td>Leads to</td>
</tr>
<tr>
<td>affects</td>
<td>Not supported</td>
<td>Potentially violated by</td>
<td>Harms</td>
</tr>
<tr>
<td>contradicts</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Negates (Security criterion)</td>
</tr>
<tr>
<td>reduced by</td>
<td>Implicitly supported (Control)</td>
<td>Reduced by</td>
<td>Mitigates</td>
</tr>
<tr>
<td>Control</td>
<td>Control</td>
<td>Countermeasure</td>
<td>Risk Treatment : leads to : Security Requirement</td>
</tr>
<tr>
<td>mitigates</td>
<td>Not supported</td>
<td>Mitigated by</td>
<td>Not supported</td>
</tr>
<tr>
<td>justifies</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Decision to treat</td>
</tr>
<tr>
<td>Analysis Technique</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>input</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>produces</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Analysis Results</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>identifies</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
8.2 Risk Model Survey Design

This subsection presents the details of the practitioner survey used to evaluate the risk model design described in subsection 8.1.3.

8.2.1 Research Method

The risk model evaluation was based on the following research questions:

1) To what extent is the risk model applicable to different software development approaches (e.g. agile and waterfall) and Patterson and Neailey’s (2002) risk management stages?
2) What are the pros and cons of using the model?
3) Which approach (textual description or the proposed model) do practitioners prefer?
4) Is the model sufficient to describe risks?

To answer these research questions, a questionnaire was designed to test whether practitioners can comprehend example risks described in the proposed risk model and to gather their feedback in order to answer questions 1-4.

8.2.2 Example Risks

The first risk is an example of a maintainability risk that was described in text as follows: “Excessive change propagation - Complex concrete sub-classes have emerged from the diverse use cases the list handling classes had to support. E.g., SystemList needs ‘deleted record processing’ whereas PropertyList does not. This causes conflicts between abstract class code and concrete sub-class code. This could be considered an unhealthy inheritance tree. There are also some common complex routines that are not always abstracted so when bugs have to be fixed sometimes many List sub-classes had to be changed”. This is a real project risk taken from the API software project studied in Chapters 4-6 and the classes involved are shown in Figure 5-2. The model representation of this risk uses all concepts described in Table 8-2 except the justifies and input correspondences.

The second risk is an example of an obsolescence risk that was described in text as follows: “Low code framework Interface changes outside of the developer’s control - Oracle Data Integrator (ODI) has changed its interface specification. This will require code to be reworked if ODI has to be upgraded”. The model representation of this risk used all of the concepts except
The first risk was chosen as an example of a risk that might be identified by a maintainability metric-based analysis technique to demonstrate how the Indicator, Indicator Value, indicated by, is for, indicates, Analysis Technique, input, produces, Analysis Results, and identifies concepts would provide traceability between the risks identified, the metrics that indicate those risks, and the analysis technique used. By contrast, the second risk was chosen as an example of a risk that would be identified through obsolescence management practices. Using risks relating to different concerns (maintainability and obsolescence) that would be identified by different techniques (metric analysis and obsolescence management) was a deliberate part of experimental design to ensure participants would have to reason about the flexibility of the risk model. Appendix E presents the two example risks shown to the participants in the questionnaire.

The correspondence named input was not used in either case in order to keep the diagrams shown to the participants simple and focused on the main concepts of interest. The correspondence named justifies was not shown because the evaluation scenario presented to the participants was about describing risks and mitigations identified by architecture analysis, as opposed to monitoring of mitigations for those risks.

8.2.3 Questionnaire Design

The questionnaire has four parts as shown in Appendix E. In part one participants are asked questions about their practitioner and architecture experience. These questions are designed to provide context to each participant’s response. The participant experience questions in part 1 (questions 1-10) required a numeric answer.

In part two the participants are provided with a diagrammatic representation of the risk model and definitions for each of the concepts and relationships shown. This was provided as a reference to ensure all participants would have a common understanding of the risk model.

In part three participants are provided with two technical risks, each represented as plain text and with diagrams, following the proposed model. To ensure the participants understood the risks, two common and relatively high-level risks were chosen. The model representations of the risks were also annotated with the aspects of the textual description that related to each model entity in order to further aid comprehension. There are no questions in part 3. After
reviewing the text and diagram representations of the risks in part 3, participants answer questions in part 4 to gather their feedback.

In part four the participants first answer questions designed to collect practitioner feedback about the risk model’s applicability to the different stages of risk management in waterfall, agile, and scaled agile projects (research question 1). The model applicability questions (questions 11-17) required the participants to specify yes, no, or not sure for waterfall, agile, and scaled agile in order to denote the different software development approaches they think the risk model could be applied to, for each risk management stage. In addition, participants were invited to provide qualifying comments to explain their applicability responses. Question 17 was included to see if participants think modelling risks would be worthwhile when a project has already decided not to follow a model based architecting approach for design. In part four the participants also answer questions to garner their opinions about the risk model’s pros and cons (research question 2), whether the risk model is preferred to textual risk descriptions (research question 3), and completeness of the risk model (research question 4). Participants were requested to provide short textual responses to the remaining questions (18-22) in part 4.

8.2.4 Participants and Recruitment

The participants were drawn from different divisions of a systems and software engineering company including research and innovation, systems engineering, software development, testing, and quality assurance. This ensured that the model was reviewed from different perspectives within systems and software engineering to ensure the evaluation is not prejudiced by any particular sub-community. Considering different perspectives is essential for the model to fulfil its goal of improving risk management because for adoption it needs to support all stakeholders in the design, development, and implementation of systems and software. It also ensured that all participants had a good degree of risk management experience because risk management is a core part of the ISO 9001 certified quality system operated by the systems and software engineering company. Therefore, participants were not asked about their level of risk management experience in part one of the questionnaire because they routinely identify and manage risks when working within that quality system.

8.2.5 Data Analysis

Quantitative analysis was used to establish the overall experience profile of the participants (questions 1-10) and the applicability of the model (questions 11-17). A qualitative thematic
analysis of the textual responses to questions 11-22 was then performed by following the Braun and Clarke’s (2006) approach explained in section 3.1.3 by coding important features of the data set that might be relevant to answering the research question (to what extent is the risk model applicable to different software development approaches and risk management stages, what are the pros and cons of using the model, which approach do practitioners prefer, and is the model sufficient to describe risks?).

8.2.6 Ethical Considerations

The survey received ethical approval from the Open University’s ethics committee. Appendix F contains the participant information sheet and Appendix G contains the consent form used. The questionnaire was provided to each participant in electronic format so that they could complete it in their own time. Appendix H presents a letter from the company granting permission for the researcher to conduct the survey. The researcher is an employee of the company whose staff were surveyed.

8.3 Survey Results

8.3.1 Practitioner Experience Results

Figure 8-9 shows that collectively the 14 participants had a mean of 9 years of system engineering experience, 13 years of software development experience, 4 years of enterprise architecture experience, 8 years of solution architecture experience, 10 years of technical architecture experience, 2.5 years of SysML experience, 7.5 years of UML experience, 17 years of waterfall experience, and 4 years of agile experience. These results suggest that the sampled participants are suitable to review and comment upon the applicability of the proposed risk model to different domains of architecture practice (enterprise, solution, technical) as well as waterfall and agile approaches to systems and software development.
**8.3.2 Model Applicability Results**

Figure 8-10 shows the percentage of participants who thought the risk model would or would not (i) benefit design reviews, (ii) benefit Patterson and Neailey’s five stages of risk management, and (iii) be beneficial when an architecture model is not available for different approaches to systems and software development. Over 75% of the participants think the model will benefit design reviews, risk analysis, and risk assessment in waterfall projects. Over 64% think the model will benefit risk mitigation in waterfall projects and only 7% disagree. Approximately 55% think the model will benefit risk identification in waterfall projects with only 14% disagreeing, and approximately 50% think the model will benefit risk monitoring with only 14% disagreeing.
Figure 8-10 Applicability to Agile, Scaled Agile, and Waterfall (Questions 11-17)
Figure 8-10 indicates that the participants are less confident about the applicability of the proposed model to agile and scaled agile projects. Less than 50% of participants suggested the risk model would benefit design reviews, risk identification, risk assessment, risk mitigation, and risk monitoring.

However, more than 60% of participants thought the model will benefit risk analysis in agile and scaled agile. In addition, less than 20% of participants think the model would not be of benefit to design reviews, risk identification, risk assessment, risk mitigation and risk monitoring in agile and scaled agile projects. Figure 8-10 also indicates that the levels of uncertainty about applicability are greater for agile and scaled agile than waterfall. Altogether, more participants think the model will be applicable to agile and scaled agile than those who do not agree, but more are uncertain about the benefits than for waterfall.

For all approaches (waterfall, agile, and scaled agile) fewer participants (35%, 45%, and 29%) suggested the risk model would benefit projects that do not already have a design model (Q17). This is expected because traceability between the risk and the AD elements giving rise to it and AD elements impacted by the risk are the main hypothesized benefits of the model.

8.3.3 Approach Preference, Pros, Cons, and Completeness Results

![Bar chart showing approach preference](image)

Figure 8-11 Practitioner Approach Preference (Question 19)

Figure 8-11 indicates that only 7% of participants stated a preference for the textual risk description compared to 36% of participants preferring the risk model representation (Question 19). However, a further 36% of participants would prefer both representations to be available. Therefore, 72% of participants indicated a preference to have the risk model representation
available. A z-test confirms that the preference for the model representation over the text representation is only marginally insignificant ($z=1.84$, but $z>=1.96$ is needed for $\alpha=0.05$). However, the preference 72% of participants to have the model representation available over just having text is significant ($z=3.48$, $\alpha=0.001$).

Figure 8-12 shows that the participants considered the model to have about the right level of detail for modelling risks because 64% of participants did not consider the model to include redundant concepts, whereas 28% did. However, 64% of participants suggested items they considered important for modelling risks are missing from the model.

![Figure 8-12 Model Completeness (Questions 20 and 21)](image)

Table 8-4 shows the results of the thematic analysis. The column ‘Codes’ indicates the codes assigned to important phrases in the participants’ responses during step 2 of the Braun and Clarke (2006) method, and column ‘Theme’ indicates the final broader themes that the codes were related to in steps 3-5. The ‘Participants’ column indicates how many unique participants provided phrases relating to a code.

Overall, Table 8-4 shows that the participants were able to propose: 9 advantages and 5 disadvantages of the model representation; 10 barriers to adoption of the risk model; 9 architecture scenarios where the model is perceived to add value to architecting; 15 risk concepts that are missing from the model; and three model concepts that are potentially redundant for use in practice. Appendix I provides a mapping between the codes assigned and the participant’s original comments. The next section discusses the themes identified in more detail as well as validity threats to the research.
### Table 8-4 Results of Thematic Analysis (Questions 11-22)

<table>
<thead>
<tr>
<th>Theme</th>
<th>Codes</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Advantage</strong>, i.e., an advantage of using the proposed model by comparison to textual risk descriptions.</td>
<td>Rigour</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Traceability</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Understanding</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Risk Patterns</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Automation, Early Mitigation</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Efficiency, Report Generation</td>
<td>1</td>
</tr>
<tr>
<td><strong>Model Applicability</strong>, i.e., architecture scenarios where the model is perceived to add value to the task of architecting.</td>
<td>High Integrity, Requirements Analysis, Not just design risks</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Technical Debt, Agile adoption, Architecture Recovery, Change Impact Analysis, Decision Support, Operational Risk Analysis</td>
<td>1</td>
</tr>
<tr>
<td><strong>Barrier to adoption</strong>, i.e., a challenge that needs to be overcome to adopt the model in practice.</td>
<td>Process, Training</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Text Needed, PM integration</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Flexibility, Risk View, Tooling</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Extensibility, Needs Testing, Overwhelming</td>
<td>1</td>
</tr>
<tr>
<td><strong>Model Disadvantage</strong>, i.e., a disadvantage of using the proposed model by comparison to textual risk descriptions.</td>
<td>Effort</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Cluttering</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Constraining, Misuse, Out of date</td>
<td>1</td>
</tr>
<tr>
<td><strong>Missing Aspect</strong>, i.e., something the participant expected to be in the risk model but could not find.</td>
<td>Probability, Impact Level</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Status</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Mitigation Cost, Mitigation Effect, Risk Category</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Alt. Mitigations, Compounding Risks, Events, Impact Results, Overall Risk Score, Relative Impact, Mitigates Concern, Related Concern, Risk Conflicts</td>
<td>1</td>
</tr>
<tr>
<td><strong>Redundant</strong>, i.e., something available in the model that the participant considered to be unnecessary.</td>
<td>Analysis Technique, Analysis Results</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Architecture Rationale</td>
<td>1</td>
</tr>
</tbody>
</table>
8.4 Discussion of Participant Feedback

This section reviews the results in order to identify improvements needed to the proposed architecture risk model.

8.4.1 Model Advantages

Most participants (71%) suggested the model would improve the rigour of risk descriptions. Many participants (65%) also suggested traceability from the risk to the risk inducing and risk impacted parts of the architecture would be an advantage of using the model. Identifying rigour and traceability as model advantages in the thematic results is consistent with the quantitative results that suggest participants think the model is more applicable to waterfall software development than agile. That is because rigorous traceability between engineering artefacts is a cornerstone of the waterfall approach. Identifying rigour and traceability as model advantages is also expected because both are a general benefit of Model Based Systems Engineering (MBSE) according to Madni and Sievers (2018).

According to 57% of participants, understandability is a model advantage due to models being less ambiguous than text. However, the suggestion that risks described by the model will be more understandable needs to be balanced with the suggestion by 36% of participants who indicated that training will be necessary for the model to be adopted. In an answer to research question 3, there was a clear preference for using the risk model (alone or in combination with text) over text risk descriptions. When combined with the qualitative results the overall view is that risk models are preferred because they are more understandable, rigorous, and traceable, so long as training is provided to users of the risk model.

Four participants (29%) provided comments suggesting that the model could be beneficial with respect to risk patterns. Their comments indicated that once a risk has been modelled, the modelled risk could be used in two ways. Firstly, as a way to communicate the risk associated with a specific design to other designers. Secondly, to identify other instances of a modelled risk in the same or other models. Furthermore, two participants suggested that automation is a model advantage because searching for other instances of a modelled risk could be performed by a search algorithm. This suggestion is highly synergistic with the approach proposed by Almorsy, Grundy, and Ibrahim (2013) whereby models are searched for known vulnerability signatures. However, the quantitative results show that the percentage of participants who think the model would be applicable to risk identification is 57% for waterfall, 50% for agile, and
43% for scaled agile. Even though some participants identified how risk pattern searching could allow the model to play a role at the identification stage, some participants remain to be convinced about the applicability of the model for risk identification.

8.4.2 Model Disadvantages

The potential disadvantages of the risk model suggested by the thematic analysis are the additional effort it would take to model risks over and above the effort to describe them in text (50% of participants), as well as design models becoming cluttered by the risks modelled (36% of participants). The former is consistent with the quantitative results that show participants are less sure about the applicability of the risk model to agile developments because the risk model is a more comprehensive way of documenting software risks, and the agile manifesto advocates favouring working software over comprehensive documentation. The former therefore represents an opportunity for future research to establish whether projects that adopt the model achieve a return on investment due to improved risk management and whether the benefits outweigh any loss of agility.

8.4.3 Barriers to Adoption

The participants suggested several barriers to adoption they felt would need overcoming for the model to be used in practice. The most widely suggested barrier is that adoption would need support with process and training (36% each). This seems reasonable because architecture and software design is a specialist profession that already requires significant training and processes. Other participants suggested that specialist tooling and risk views would be required. The introduction of such tools and views, either standalone or by adding capabilities to existing MBSE tools such as Enterprise Architect (Sparx, 2020), represents an opportunity to mitigate the training and process barriers as well as the design cluttering disadvantage. The introduction of risk views is consistent with the ISO 42010 concept of having different views to express the architecture from the perspective of specific system concerns because risk is modelled as a subtype of concern. Three participants purported the need for integration with project management practices. This seems valid because, whilst Poort and van Vliet (2012) assert that architecture should be considered a risk and cost management discipline and consider architects to be risk managers, in widely used project management methodologies such as PRojects IN Controlled Environments (PRINCE2®; Bentley, 2019) and Project Management Professional (PMP; Sanghera, 2018), the project manager is responsible for risk management. So, for the risk model to be useful to practitioners, it must integrate with existing risk management approaches.
used by project managers. Three participants suggested that the textual risk descriptions would still be necessary. The intention was that each entity in the risk model would have a description attribute for storing the textual representation. Incidentally, a participant suggested that a benefit of the risk modelling approach would be to support tooling capable of producing risk reports from architecture models. Such risk reports might offer a way for architects to adopt the risk modelling approach whilst working collaboratively with project managers and their risk management methods, especially if the risk report includes the textual risk representation extracted from the description attributes.

### 8.4.4 Model Applicability

Thematic analysis revealed that participants suggested nine different situations where they felt the model would be applicable. Only three of these suggestions were proffered by multiple participants. Two participants suggested the risk model would be beneficial when working with high integrity systems (e.g., safety critical systems) and requirements analysis. The suggestion that the risk model would be more applicable to high integrity systems and requirements analysis is consistent with the quantitative results that show participants are more confident about the applicability of the risk model to waterfall projects than agile projects. That is because upfront requirements analysis is an integral part of waterfall and to date agile is less likely to be used for safety critical systems (Islam and Storer, 2020). Two participants also suggested that the model could be used to describe operational risks of the system described by the architecture as opposed to risks relating to implementing the system. This supports the view that the output of the architecture analysis techniques designed to identify system failures and system hazards found in the literature review (Appendix D) could be represented by the model.

### 8.4.5 Missing Aspects

Thematic analysis revealed that the most widely suggested items considered to be missing from the risk model are probability and impact level. Whilst these are not shown in Figure 8-7, the intention was for probability and impact level to be attributes of the Risk and Consequence entities, respectively. This feedback suggests that the definitions provided in Table 8-2 need to be more detailed or probability and impact level need to become explicit entities in the risk model design.

A suggestion made by four participants that had not been considered though, is that the model should include a representation of risk status. Including risk status would enable
progression through Patterson and Neailey’s (2002) five stages of risk management to be tracked and enable architects and project managers to understand whether, for example, any steps had already been taken to mitigate a risk. The lack of risk status revealed by the thematic analysis is consistent with the quantitative results which suggest participants think the risk model is least applicable to the monitoring stage of risk management. These observations suggest adding risk status to the model would improve support for risk monitoring.

8.4.6 Redundant Aspects

Two participants suggested that the Analysis Technique and Analysis Results are unnecessary entities for the risk model. One participant who suggested that Analysis Results was unnecessary did so because they thought the model could be simplified by associating the Analysis Technique with the Risks identified directly, as opposed to them being associated via the intermediary Analysis Results. This represents an opportunity to simplify the model.

8.4.7 Revised Risk Model

Figure 8-13 presents a revised architecture risk model based on the feedback garnered from the practitioner survey. Three new concepts have been added and indicated with a grey background: Likelihood, Level, and Status. Likelihood has been added because five participants suggested risk probability is missing from the model. It has been termed Likelihood rather than Probability for consistency with ISO 31000 which already includes the concept of risk likelihood (probability). Level has been added because five participants suggested it ought to be possible to describe the relative impact of a risk. Status as suggested by four participants has been added to improve support for risk monitoring. In all cases Likelihood, Level, and Status are in a composition “has a” relationship with the existing Risk, Control, and Consequence entities. The use of a compositional relationship denotes that Likelihood is an attribute of a Risk or Consequence, Status is an attribute of a Risk or Control, and Level is an attribute of Consequence.
As well as adding three concepts, the Analysis Results concept has been removed because, as pointed out by one of the participants, it represents an over decomposition from a risk model point of view, i.e. the risks identified by the analysis technique are the analysis results and hence an intermediary concept is unnecessary. The area enclosed within a dotted line in Figure 8-13 illustrates the parts of the architecture risk model required to describe the results of the container based software architecture risk assessment process proposed in Figure 1-1 and evaluated in this dissertation.

Having updated the risk model using the results of the practitioner survey, the risk model is ready for a more rigorous but costly longitudinal practitioner study to determine whether using the model would help to provide a return on investment, i.e. does usage of the model yield savings due to risk avoidance that outweigh the cost of using the model in practice. To perform such a study, training and tool support would be needed in order to make the proposed risk model available to practitioners. However, such further evaluation is outside the scope of this dissertation because the objective was to answer RQ3: what kind of information is necessary in architecture risk models to describe the outputs of architecture analysis techniques? The research presented has identified the kinds of information needed to provide risk views.

8.5 Validity

This section reviews the approach taken to design a generic architecture risk model for construct validity threats using Messick’s (1987) classification of consequential, content, substantive, structural, external, and generalizability threats.
8.5.1 Consequential

The limited number of participants (14) is a threat to consequential validity. This threat is mitigated to some extent by their vast experience (Figure 8-9). Such levels of industry experience suggest the participants are qualified to comment upon the applicability and suitability of the risk model for practice.

The questionnaire asks participants about their opinions regarding model applicability, pros and cons, preference, and completeness using example risks. However, it does not test their ability to develop their own risk models which is another threat to consequential validity. This threat remains to be addressed in a future work.

Recruiting practitioners from the researcher’s company of employment represents also represents a threat to consequential validity because they may be less critical of the model in order to maintain a good working relationship with the researcher. The choice of using an anonymous questionnaire was a deliberate mitigation against this threat. Recruiting participants from different departments of the company further mitigates this threat because they are less well known to the researcher.

8.5.2 Content

Whether or not the participants understood the questionnaire represents a content threat. If they misinterpreted any of the questions, their answers and the statistics based upon them (Figure 8-10, Figure 8-11, and Figure 8-12) may not be measuring the applicability and suitability of the model as the research intended. Confidence that the participants were able to understand the questionnaire results from the participants providing almost six thousand words of rich comments critiquing the model and the level of agreement between participants about model advantages, disadvantages, applicability, missing items, and redundant items.

8.5.3 External

The scope of the literature review presents an external validity threat. As ISO 42010 is a standard for “Systems and software engineering” any proposal to extend it should be generalisable to system and software architecture, but this literature review focused on software architecture and not systems architecture. Despite this, some of the analysis techniques reviewed, such as Williams and Smith (1998), Said et al. (2011), and Alberts, Woody, and Dorofee (2014) can encompass hardware aspects as well as software.
8.5.4 Generalizability

Many software projects are now choosing agile methodologies that favour working software over documentation. This does not mean that upfront should not be done, it means that it should be done when necessary. This means the proposal of using a risk model to improve support for the risk management in software engineering may be limited to certain types of projects. The results of the thematic analysis provide some indication of the types of projects to which the surveyed practitioners think the model would be most applicable (model applicability theme). When software has been developed using agile methods it has an “evolved” architecture. That evolved architecture can be recovered using methods such as reflexion modelling (Murphy, Notkin, and Sullivan, 2001). Recovery means the proposed architecture risk model could be used to support the analysis of risks in working software.

Furthermore, software architecture analysis techniques may have been missed from the review, which means the extent to which risks can be identified and described in software architectures may not have been fully determined. As a consequence, there could be other techniques whose output may not fit within the proposed risk model undermining the generalizability of the proposition. This threat to construct validity is mitigated by the number of architecture analysis techniques reviewed, the variety of risks they assess, and because many of them were taken from widely cited surveys.

As explained in section 8.2.4, all participants came from the same systems and software engineering company. This represents a further generalizability threat because the participants’ projects, processes, and working practices may not be representative of other organisations. However, many of the participants have worked for multiple organisations during their career which provides some mitigation against this threat.

8.5.5 Substantive

The assumption that a risk model will improve risk management can only really be determined from a longitudinal study using practitioners. Whilst the results are encouraging, given that more participants indicated the model would improve risk management than those who suggested it would not (for waterfall, agile, and scaled agile projects), further research is needed to see if projects using the model are more successful than those that do not adopt the model, and if the use of the risk model provides a return on investment.
8.6 Chapter Summary

This chapter began by explaining the approach taken to design a novel architecture risk model that could be used to represent the output of architecture analysis techniques, including risk containers, in order to support risk views based on the output from the container based software architecture risk assessment process proposed in Figure 1-1. The design was based on existing concepts in the international standards for architecture description (ISO 42010) and risk management (ISO 31000). An iterative approach to design was used that ended when the design could represent the output of all the architecture analysis techniques found in the literature review. Next the chapter described a practitioner survey to establish whether practitioners think the model is applicable to different software development approaches and stages of risk management, whether practitioners prefer the model to textual risk descriptions, and whether practitioners think any concepts are missing or redundant. The survey results suggest that the participants have suitable levels of experience to take part in the experiment. The key findings of the risk model evaluation are that practitioners consider the model to be more applicable to waterfall software development than agile, more practitioners consider the model to be applicable to agile software development than those who do not, and practitioners prefer the model to textual risk descriptions due to the fidelity, rigour, and traceability supported by the model. A revised model based on participant feedback about missing and redundant concepts was presented, before the chapter ended with a review of validity threats.

By comparison to existing risk models found in the literature (Fabian et al., 2010 and Mayer et al., 2019), the proposed model includes five novel features: (i) it can describe the hierarchical relationship between stakeholder concerns and specific risks that relate to those concerns; (ii) it supports indicator values relating to different risks so that the significance of different risks can be compared; (iii) it enables mitigations to be impact specific which recognises that a risk may have many impacts which need different mitigations; (iv) it provides traceability between risks identified and the techniques used to identify those risks which means the efficacy of different risk identification techniques can be compared; and finally (v) it integrates the standards for architecture description (ISO 42010) and risk management (ISO 31000) to guide architects when modelling risks relating to architectures. The proposed architecture risk model represents the final contribution of this dissertation (Table 1-1, contribution C10).
Chapter 9 - Discussion and Conclusion

This chapter discusses the overall findings and conclusion of the research, practical applications of the research, and possibilities for future research.

9.1 Risk Isolation Properties of Containers

Section 2.1 presented a systematic literature review based on widely cited surveys reveals that many techniques for analysing software architectures have been proposed owing to the relationship between architecture and maintainability risk (Dobrica and Niemelä, 2002; Babar and Gorton, 2004; Babar, Zhu, and Jeffery, 2004; Mattsson, Grahn, and Mårtensson, 2006; Bernardi, Merseguer, and Petriu, 2012; and Malhotra and Chug 2016). Many of the techniques found can not only identify risks but also attribute them to architecture subsets. This is a powerful capability because it provides architects with insight into which parts of an architecture are more likely to result in project failure if the risks identified are not managed. The ability to predict which parts of an architecture are riskier than others suggest that it ought to be possible to create an architecture risk view: an architecture view that shows where the risks are. Despite the plethora of existing techniques available, none of them consider which kinds of architecture subsets are the most risk isolating using the definition provided in section 1.3: subsets having a low degree of overlap (architectural element sharing) and being associated with risk predicting metrics. Thus, the literature does not discuss which architecture subsets are effective risk containers (as defined in section 1.3).

Knowing which types of architecture subsets contain isolated risks is even more powerful. Firstly, because the scope of the risk in terms of the elements that are inducing the risk is better understood, which allows risk impacts and mitigations such as redesign to be more precisely estimated. Secondly, because it allows the architecture models and views that provide the most accurate risk views to be derived. Knowing which models and views best support managing risks empowers architects to focus their efforts on ensuring projects are more likely to succeed by managing risk and cost as advocated by Poort and van Vliet (2012). Therefore, as presented in Chapter 1, the first research question (RQ1) is concerned with: how effective are Design Rule, Resource, and Use Case Containers at predicting and isolating the risks of implementation error-proneness and implementation change propagation? The question bounds the types of containers (Design Rule, Resource, and Use Case) and risks considered (error-proneness and change propagation) in order to have a manageable research project.
For a container type to be risk isolating, it must be both element isolating and risk predicting as defined in section 1.3. Therefore, RQ1 was split into two sub questions: (1) is the container type element isolating, i.e., to what degree do containers overlap and share elements, and how much of the coupling is between elements in the same container?; and (2) are container level design metrics predictive of the risk isolated? By answering these two sub questions it can be deduced whether a risk container type is risk isolating and answer RQ1. That is because if containers are error-proneness and change propagation risk isolating, the architectural elements should exist in few containers (hypothesis H1) and container metrics should correlate to the relative level of risk isolated into the containers (hypothesis H2).

As explained in section 4.5, the results indicate that the answer to sub question one is Design Rule Containers (DRCs) because DRCs have the least amount of class sharing in three projects and come marginally in second place in the fourth project, and DRCs have the greatest mean internal coupling in three projects, but the highest median and overall distribution of internal coupling in all four projects. The fact that DRCs typically share fewer classes and have a greater proportion of internal coupling is intuitive because individual DRCs cluster tightly coupled classes into modules, which interact with (depend on) other DRCs through relationships from only a subset of their members. These findings support Wong et al.’s (2009) original goal for the Design Rule Hierarchy algorithm upon which DRCs are based, which was to separate areas of architecture into independent parts to increase developer parallelism. Having established that DRCs are the most element isolating type of risk container (hypothesis H1, sub question one), RQ1 requires whether any of the container types are risk predicting (hypothesis H2, sub question two) to be considered next.

Hypothesis H2 requires a strong (and significant, \( \alpha \leq 0.05 \)) correlation between design metrics and error-proneness to be observed. The results presented in sections 5.3-5.6 show correlations between Coupling Between Objects (section 5.3), Fan In (section 5.4), and Cyclomatic Complexity (section 5.6) and error-proneness were observed for all three risk container types, but no container type/design metric combination had a strong and significant correlation to error-proneness in more than three of the four projects. It is concluded that DRCs are the most error-proneness isolating container type because in addition to them being the most element isolating container type (hypothesis H1, sub question one), they are as risk predicting as the other container types (hypothesis H2, sub question two).

When considering all container types, Cyclomatic Complexity is the most reliable risk container metric for predicting error-proneness of those tested because unlike Coupling
Between Objects and Fan In it is not confounded by container size and is at least moderate in twelve container type/project scenarios tested. However, the Cyclomatic Complexity correlations for DRCs were only strong and significant in two of four projects (section 5.6), whereas the predictive power of DRC Coupling Between Objects (section 5.3) was strong and significant in three for four projects. Thus, in addition to the conclusion that DRCs are the most error-proneness isolating container type, it is recommended that practitioners use both Coupling Between Objects and Cyclomatic Complexity when deciding whether to take actions (e.g. redesign) to reduce the error-proneness risk of a specific container.

Having answered RQ1 for error-proneness, it must next be answered for change propagation. The correlation results (section 6.2) between change propagation calculated from the design and change propagation calculated from the implementation (co-change) are very strong and significant for all three container types in all projects (sub question two, hypothesis H2). That suggests that change propagation calculated from the design is equally co-change risk predicting for DRCs, Resource Containers (RCs), and Use Case Containers (UCCs). Therefore, it is also concluded that DRCs are the most change propagation isolating container type because they are the most element isolating container type (sub question one, hypothesis H1) and there is a very strong and significant correlation between DRC change propagation calculated from the design and implementation co-change (sub question two, hypothesis H2).

Whilst the change propagation correlations were similar for all three container types, other observations provided additional evidence that DRCs are more change propagation isolating than RCs and UCCs. The reason for this is due to the fact that DRCs have: (i) the highest probability of classes changing with other classes with which they share containers in three of four projects (section 6.3); (iii) in all four projects used in the evaluation, co-change is more likely between classes that share a DRC than between classes that do not share DRCs, which is not true for RCs and UCCs (section 6.3); and (iii) the most change sets isolated in containers (section 6.4). The latter observation (iii) demonstrates that DRCs are more supportive of hypothesis H3 (if error-proneness and change propagation risks are isolated into Design Rule, Resource, and Use Case Containers, the set of files that must be changed to fix errors or implement changes should fit neatly into containers) than RCs and UCCs.

Furthermore, the following observations provide further evidence that risk containers are risk isolating: (i) the correlations between container level CBO and implementation error-proneness are stronger than the correlations between class level CBO and error-proneness (section 5.3); and (ii) the correlations between container level CPD and implementation CPI are...
stronger than the correlations between class level CPDC and CPIC (section 6.2). These observations suggest that clustering classes into containers based on design relationships has the effect of separating areas of relatively high risk from areas of relatively low risk.

9.2 Meaningful to Developers

Knowing which type of containers are effective is not enough because the goal is to help practitioners. Usefulness is the subject of part three of the definition of meaningful containers introduced in section 1.3: (i) isolate real project risks; (ii) be formed from design artefacts used in practice; and (iii) help partitioners locate find risk inducing architectural flaws more easily. Therefore, the second research question considered is: does splitting an architectural design into smaller container diagrams help practitioners find error-inducing flaws and identify change impacts? To answer the second question an online experiment was designed whereby practitioners had to identify error-proneness inducing design flaws and the impact of changes. The participants were randomly allocated to different groups requiring them to take part using designs presented using different kinds of smaller risk container diagrams or as a larger single diagram (control).

The results of the usefulness experiment described in Chapter 7 suggest that analyzing a design as a collection of containers leads to more accurate detection of cyclic dependencies isolated within smaller container diagrams. The trade-off for the observed accuracy improvements is that it takes longer to analyse a design presented as a collection of containers than one presented as a larger single class diagram. A general improvement was not observed for cyclic dependencies spanning multiple container diagrams, but an improvement for DRCs was observed (section 7.4). As for identifying the impact of design changes, the results suggest splitting an architecture into smaller containers has no bearing on the practitioner’s ability to detect change impacts correctly (section 7.5). However, it was observed that the error frequency was greater for the larger single class diagram control than for the participants using risk containers. Overall, the results of the usefulness experiment suggest all three risk container types are equally useful to practitioners by offering accuracy improvements in detecting potentially error-inducing cyclic-dependencies. This supports hypothesis H5 and provides some evidence to answer RQ2 because splitting an architectural design into smaller container diagrams helps practitioners find error-inducing flaws within them by comparison to one larger diagram covering the whole design.
Overall, the results of the usefulness experiment, allied to developer feedback for DRCs discussed in section 6.5, allied to DRCs needing only class diagrams, allied to them being error-proneness and change propagation isolating all supports the proposition that DRCs could be useful for practitioners who are working with upfront designs. Another implication of these observations to practitioners is that if they need to minimise error-proneness and change propagation risks, investing design and analysis effort in upfront class diagramming is more likely to pay off than investing effort in sequence diagramming.

9.3 Risk Views

Having established risk containers as an effective approach for isolating maintainability risks in architectural designs, the question of how the risk container output should be presented to architects (RQ3) was considered in chapter 8. The third research question asked: what information is necessary in architecture risk models to describe outputs of architecture analysis techniques? To answer the third question an architecture risk model based on the international standards for architecture description (ISO 42010) and risk management (ISO 31000) was synthesised from the architecture analysis techniques found in the literature review and evaluated using a practitioner survey (section 8.2). The generic model was designed to be able to describe the output of all the architecture analysis techniques shown in Appendix D including risk containers to enable practitioners to view the results of all risk analyses alongside the architectural design.

The key findings of the risk model evaluation are that practitioners consider the model to be more applicable to waterfall software development than agile (section 8.3.2), more practitioners consider the model to be applicable to agile software development than those who do not (section 8.3.2), and practitioners prefer the model to textual risk descriptions due to the fidelity, rigour, and traceability supported by the model (section 8.3.3). A revised model based on participant feedback about missing and redundant concepts is presented in Figure 8-13.

9.4 Practical Applications

The main practical applications of the research presented in this dissertation are the potential to further develop the Automated Container Analysis Tools (ACAT) presented in section 4.3 into software that could be used by practitioners to find areas of relatively high error-proneness and change propagation risks in graph-based design representations such as UML (Object Management Group, 2017) and SysML (Object Management Group, 2019) using
the risk container metrics presented in this thesis. This could be achieved by plugging the ACAT functionality into modelling tools such as ArgoUML (Softonic International S.A., 2020) and Enterprise Architect (Sparx, 2020). The availability of the ACAT within such commonly used Model-Based System Engineering (MBSE) tools would enable practitioners to identify risks as they build up their design models. The benefit of using the ACAT as models are composed is to find risky areas of the design as soon as possible which will minimise the cost of resolving any problems because errors found during design are less expensive to correct than errors found during implementation or testing (Akingbehin, 2005).

Furthermore, the risks identified by using the ACAT could be outputted into an implementation of the risk model presented in Chapter 8. Housing the risk model implementation within the same MBSE tools would enable views to be presented to architects that include traceability between the risks identified and the architectural elements and design from which they arise. Such capability would enable practitioners to compose models and analyse risks in a single MBSE tool.

Proposing a risk model relating to ISO 42010 concepts naturally leads to the question of how to integrate the model with ISO 42010. Model Kinds and Architecture Model are an extension mechanism built into ISO 42010 that could support the risk model. Model Kinds specify “modelling conventions appropriate to the concerns to be addressed” (ISO 42010, p. 6). ISO 42010 further elaborates the definition of model with the statement that “M is a model of S if M can be used to answer questions about S”, (p. 22). As Risks encapsulate the set of AD elements giving rise to the risk, they can answer the following question: “which AD elements result in a specific instance of a Risk?” Their ability to answer this question and others suggests risks are models in ISO 42010 terms.

Using the Model Kinds/Architecture Model extension mechanism to integrate the proposed risk model with ISO 42010 makes a risk model an optional part of an architecture description. In much the same way that architects using UML could opt out of creating a class model, they could also opt out of creating a risk model if it were integrated using the Model Kinds/Architecture Model extension mechanism. To promote risk analysis in architecture practice, as advocated by Poort and van Vliet (2012) in RCDA, and realise the benefits of driving down cost and time, it is suggested that ISO 42010 is extended to explicitly include the proposed risk model. Inclusion would nudge modelling languages such as SysML (Object Management Group, 2019) and UML (Object Management Group, 2017) to provide support for modelling risks to maintain compliance with the standard. That in turn would require MBSE tools such as
Enterprise Architect (Sparx, 2020), Rhapsody (IBM, 2020), and ArgoUML (Softonic International S.A., 2020) to provide features that enable architects to capture risks as an integral part of architecture composition and analysis. The availability of MBSE tool support for describing risks as part of architecture composition would address the barriers to adoption of needing risk views and tooling identified in the thematic analysis (Table 8-4).

9.5 Overall Strengths and Limitations

Perhaps the biggest limitation of the research presented in this dissertation is that it focuses on making predictions based on upfront architectural designs that are less likely to be available in agile projects. However, the results presented, especially those in chapter 7, ought to stimulate the agile software development community to re-consider how much to favour working software over comprehensive (upfront design) documentation to have maintainable software and avoid continuous refactoring (Vassallo, Palomba, and Gall, 2018). The next overall limitation is that whilst the container-based risk assessment process proposed in Figure 1-1 is intended to be generic, this dissertation only evaluated it for maintainability risks. Furthermore, the sample of projects used did not include any microservice architectures which is another limitation with respect to modern day practice (Soldani, Tamburri, and Van Den Heuvel, 2018). Finally, the low levels of participation in the experiments and surveys mean limited evidence about whether practitioners would be willing to adopt a container-based risk assessment process and an architecture risk model are available in this dissertation.

However, whilst the number of projects and level of participation are considered limitations, the involvement of some industry projects and some practitioners in all aspects of the research is considered a key strength given the goal of wanting to help practitioners mitigate risks associated with architectural designs. The breadth of research approaches and methods used (Table 3-1) is another strength that provides confidence that the container-based risk assessment proposition has been probed from multiple perspectives. Furthermore, the consistency of the results presented, especially the results relating to isolating and predicting change propagation risks, suggests the approaches and methods selected are appropriate. Finally, aligning the output of container-based risk assessments (and other architecture analysis techniques) to international standards for architecture descriptions and risk management in the risk model presented in Chapter 8 is considered a strength because it demonstrates how the proposition could be fitted into modern day practices.
9.6 Conclusion

These results provide the contributions of this dissertation first introduced in Table 1-1: the container population algorithms presented in subsection 4.1 (contribution C1); evidence that containers can be derived from designs used in practice is achieved by using two industry projects in the evaluation (contribution C2); formal definitions of container element isolation metrics, error-proneness metrics, and change propagation metrics provide further contributions (contributions C3, C5, and C7); identification of the container type that is most element isolating (contribution C4), an evaluation of each risk container type’s ability to predict and isolate implementation error-proneness (contribution C6), and an evaluation of each container type’s ability to predict and isolate implementation change propagation (contribution C8). These contributions demonstrate that Design Rule Containers are the most effective risk container type for isolating error-proneness and change propagation. This is due to them exhibiting very strong and significant correlations between design metrics and implementation change propagation, moderate or stronger correlations between design metrics and implementation error-proneness (sections 5.8 and 6.6), and having a lower degree of overlap than Resource and Use Case containers (section 4.5). These results were not repeatable using control containers created based on random assignment of Architectural Description elements to containers which suggests that it is the coupling isolated into the risk containers that is resulting in the risks. Furthermore, another contribution is provided by the results of an on-line experiment that demonstrates architecture review mitigations are more effective if designs are presented as a collection of smaller container diagrams because practitioners can more easily locate error inducing design flaws that require redesign mitigations (section 7.4): evidence about whether smaller risk containers diagrams help practitioners locate error inducing design flaws and impact changes more easily (Table 1-1, contribution C9). The final contribution of this dissertation is an architecture risk model that enables the risks identified to be presented to architects as risk views (Table 1-1, contribution C10). The model is capable of describing the output of risk container analysis and the other architecture analysis techniques found in the literature (section 8.4.7).

The implications of this thesis to practitioners are that error-proneness and change propagation risks can be predicted and isolated into Design Rule Containers enabling them to be mitigated. The low degree of overlap between Design Rule Containers is consistent with the underlying hypothesis behind Wong et al.’s (2009) Design Rule Hierarchy clustering algorithm upon which Xiao, Cai, and Kazman’s (2014) Design Rule Spaces and Design Rule Containers are
based. Like Xiao, Cai, and Kazman, the findings also reveal that most of the risk is likely to be in a few of the containers. Furthermore, the results indicate that ranking containers is more effective than ranking individual architecture elements for error-proneness and change propagation risks due to stronger correlations observed at the container level than at the individual class level. In addition, presenting an architecture as a set of smaller risk container diagrams, than larger diagrams, will enable practitioners to more easily locate error inducing design flaws. The research also provides some evidence to suggest that for maintainability risks such as error-proneness and change propagation, class diagrams are more useful than sequence diagrams. This finding can help practitioners decide where to direct architecture effort if stakeholders are concerned about maintainability risks in keeping with the premise of Risk and Cost Driven Architecture (Poort and van Vliet, 2012). Finally, the architecture model provides a means for architectures to capture traceability from the risks identified to the risk inducing architecture elements and architecture elements that will be impacted if the risk occurs.

9.7 Future Work

There are several avenues open to future work because this dissertation is limited to three risk container types and two specific maintainability risks based on the structural dependencies between classes.

Firstly, the research could be expanded by investigating additional container types, such as the others identified in Leigh, Wermelinger, and Zisman (2017) including component and scenario containers as well user story containers. The latter encapsulate the classes used to fulfil a user story in agile development. Such further work would gather evidence as to whether any other container types can outperform Design Rule Containers for isolating error-proneness and change propagation risks using structural dependencies.

Secondly, there is the question of what types of containers would work best in different types of architecture? For example, what kinds of risk containers would isolate error-proneness and change propagation in microservice architectures (Zhang et al. 2019)? Whilst not explicitly tested in this thesis, Design Rule Containers should be helpful to predict error-proneness and change propagation within the internal architecture of individual microservices, i.e. where the choice to modularise functionality into a microservice is considered a design rule. One Design Rule Container per microservice would be ranked by coupling and change propagation design metrics. However, the whole system of microservices is unlikely to be well documented, especially in an eco-system of public business-to-business microservices that has evolved
through time, and where the microservices interact through event messages as opposed to structural dependencies. As a consequence, alternative methods would be needed to reverse-engineer the event-based dependency graph between microservices. One approach would be to build a framework to observe and record the messages sent between microservices. Such an observer approach would be similar to that proposed by Klein et al. (2016) for calculating metrics about big data systems.

Thirdly, the research could also be expanded by investigating risks relating to other quality attributes, for example security or performance risks. Investigating risks relating to other quality attributes would provide evidence about the versatility and generalizability of risk container analysis beyond maintainability.

Finally, future work could investigate how to integrate risk container analysis with Model-Based System Engineering tools as well as software engineering and project management practices. Such research would identify the tooling requirements and practices needed to maximise the return on investment for adopting risk container analysis into industry projects. A specific area of interest would be how to best introduce into agile projects where the upfront designs needed to perform risk container analysis may not be available. Nord, Ozkaya, and Kruchten (2014) explain that architecture is needed in large scale agile projects to avoid excessive redesign later. Having determined that only class diagrams are needed to allow the use of DRCs for error-proneness and change propagation analysis, research could investigate the trade-off between the cost of producing such upfront designs and any loss of agility. After all, the agile manifesto does not promote doing without design completely, rather it advocates favouring working software over comprehensive documentation, and the underlying graph structures behind a UML (Object Management Group, 2017) class model needed for DRC analysis are hardly comprehensive. Specifically, future work could address the question: do the gains provided by performing a DRC analysis on a lightweight upfront dependency graph, such as reduced errors, reduced change costs, and less implementation refactoring to achieve maintainable software, provide a return on investment in agile projects? Nord, Ozkaya, and Kruchten’s zipper model and architecture stories might even provide a means to introduce DRC analysis into agile projects.

9.8 Closing Remarks

Upfront design has become less popular since the emergence of agile methodologies. Technological advances leading to cloud technology have driven a trend for architectures to shift
from monoliths to microservices, via the intermediate stepping-stones of component-based and service-oriented architectures. The agile era has also brought user experience as a vital discipline to the fore. In addition, the agile era has contributed to DevOps as a cultural and technological phenomenon to bring operations and development together with the support of automation, in order promote the rapid deployment of new features and upgrades (Kim et al., 2016). Despite these advances, we are left with a situation in which agile projects must include time contingency for wasteful refactoring to pay off technical debt (Vassallo, Palomba, and Gall, 2018).

As stories emerge from the field that agile seems brittle at scale (Nord, Ozkaya, and Kruchten, 2014), and that pain as well as gains come with microservices (Soldani, Tamburri, and Van Den Heuvel, 2018), there was even talk of a rebound to macroservices during the panel session at the International Conference on Software Architecture in 2019. Perhaps the previous software engineering eras need to be followed by an era of pragmatism. A new era characterised by software engineers conceptualising all of these methods as a toolbox where each tool has its pros and cons, rather than seeking silver bullets and following fashion. There is now a whole generation of software engineers who didn’t witness the motivations for UML and MBSE during the era before agile and are unfamiliar with such techniques. A generation that has been both supported by, and let down by, the agile manifesto. Supported with principles to follow, but let down by the absence of guidance about how to follow those principles. Despite some researchers addressing the question of how to balance agility with discipline (Boehm and Turner, 2004), there is clearly room for improvement, and models for the prioritisation of architecture effort are needed (Woods and Bashroush, 2017). Adding more guidance alongside the agile manifesto could result in a renaissance of software architecture design by enabling the pragmatic architect to better judge how much design to do as architecting evolves (Woods and Fairbanks, 2018), e.g. how much to favour working software over comprehensive documentation.

As we have seen in this dissertation, a modicum of upfront design could save a lot of refactoring to avoid error-proneness and excessive change propagation. Moving into a new era we should focus more on educating software engineers to know how and when to use the most appropriate tool from their ever-expanding toolbox to avoid them being victims of fashion caught chasing silver bullets.
References


  https://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=427321


Andrew Leigh, 202


Appendix A - Usefulness Experiment Research Statement

Research Statement

This questionnaire has been prepared to collect data for a PhD research project being undertaken with the Open University. The project focuses on the risk assessment of software architectures. This questionnaire is concerned with determining whether software development practitioners can use sub sets of architectures to identify and manage risks.

The motivation behind this research is to help software development projects be more successful in terms of budget and schedule as many fail when measured in these terms. Risk assessments and risk management are good project management practice, yet unmanaged risks are one of the most common causal factors of failed projects.

Data Protection

- Responses will be stored anonymously and securely on Bristol Online Survey (BOS) servers.
- The full anonymised data set will be shared with the research community and general public via FigShare.
- A paper containing the results of the data analysis will be published at Open Research Online http://oro.open.ac.uk allowing you to read about your contribution to the research.
- Data will be handled in accordance with the UK Data Protection Act of 1998.
- From the 25 May 2018 data will be handled in accordance with the EU General Data Protection Regulation (GDPR).
- By submitting the forms your are consenting to take part in the survey.
- If you would like to withdraw you may email andrew.leigh@open.ac.uk with the date/time of your submission. Withdrawal may not always be possible because all data is held anonymously.

The survey should take approximately 1 hour to complete and must be completed in one session.

Before continuing to the experiment please answer the following questions about your UML experience.
Appendix B - Usefulness Experiment Diagrams

Figure B-9-1 Usefulness Experiment Control Class Diagram
Figure B-9-2 Usefulness Experiment Example Design Rule

Container Class Diagram

Figure B-9-3 Usefulness Experiment Example Resource Container Class Diagram
Figure B-9-4 Usefulness Experiment Example Use Case Container Sequence Diagram
## Appendix C - Usefulness Experiment Participant Comment Coding

<table>
<thead>
<tr>
<th>Theme Title</th>
<th>Description</th>
<th>Participants</th>
<th>Original Participant Text</th>
</tr>
</thead>
</table>
| Tool Support Needed | Participants suggested the need for tool support. E.g. ability to search for classes, navigate between diagrams (click through), usages, printing capability, and a cyclic dependency locator. | 8            | “Only having the diagrams without the support found in architecture tools to navigate between diagrams and to see the relationships for entities on a per entity basis make it a slower task than I am used to.”  
“Very frustrating to go back and forth and scrolling long page without the functionality to search for a particular class.”  
“I think I would have found it easier to find cyclic dependencies across multiple diagrams if I had the facility to print them out.”  
“I'm glad tools do this for you.”  
“I don't know how to find a cyclic dependency in a cyclic diagram.”  
“I'm sure I've missed some cyclic dependencies.”  
“Instead of we write the class names, it would be easier for us to just select them through a checkbox.”  
“Hard to hold the model components in your head across the pages, if I printed it out then it would be more straightforward.” |
| Navigability        | Participants commented to say they found navigation within and between diagrams difficult. | 5            | “Only having the diagrams without the support found in architecture tools to navigate between diagrams and to see the relationships for entities on a per entity basis make it a slower task than I am used to. I ended up having multiple tabs open to do the navigation.”  
“The diagram is very difficult to navigate. Going back and forth is not very appealing and frustrating at times.”  
“I think I would have found it easier to find cyclic dependencies across multiple diagrams if I had the facility to print them out.” |
“If the models were displayed on a single page, the navigability of the questionnaire would improve.”

“It’s not that easy to do this in the browser – that might just be me. It’s hard to hold the model components in your head across the pages, if I printed it out then it would be more straightforward.”

| Design Critique | Participants offered a critique of the synthetic system design. | 4 | “WRT to the logger is is not clear if the logger is private / protected, this would effect the amount of code to change, I have taken protected since no logging methods declared on the AbstractFacade.”

“I find the Abstraction layer a bit overkill. Why not just use the direct classes, e.g. Factory, Dao, FoodCalculatroFactory, etc…”

“FacadeFactory is shown as abstract when it should be concrete. FacadeFactory’s dependency on Logger is not shown which it must be providing to AbstractFacade instances.”

“If the id in the DAO was only in the abstract and inherited, it wouldn’t need changing across all specialisations.”

| Question Ambiguity | A participant found a specific question ambiguous. | 1 | “The last question wants all ‘affected’ classes to be listed. I find the term somewhat ambiguous; does it mean those classes that would need to be changed? or does it mean all classes that share communication in some way with the Logger?”

| Diagram Too Large | A participant commented to say they found the control diagram too large. | 1 | “I can understand the pain of having a lengthy image but as a participant is very frustrating to go back and forth and scrolling long page without the functionality to search for a particular class.”
## Appendix D - Risk Model Usage per Architecture Analysis Techniques

Table D-1 Risk Model Usage per Architecture Analysis Technique

<table>
<thead>
<tr>
<th>Architecture Analysis Technique</th>
<th>Concepts and Correspondences required to model Analysis Technique Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td><strong>Authors</strong></td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>1994</td>
<td>Kazman et al.</td>
</tr>
<tr>
<td>1997</td>
<td>Lung et al.</td>
</tr>
<tr>
<td>1998</td>
<td>Williams and Smith</td>
</tr>
<tr>
<td>1999</td>
<td>Lassing, Rijsenbrij, and van Vliet</td>
</tr>
<tr>
<td>1999</td>
<td>Molter</td>
</tr>
<tr>
<td>2000</td>
<td>Clements</td>
</tr>
<tr>
<td>2000</td>
<td>Kazman, Klein, and Clements</td>
</tr>
<tr>
<td>2000</td>
<td>Petriu, Shousha, and Jalanpurkar</td>
</tr>
<tr>
<td>2000</td>
<td>Vieira, Dias, and Richardson</td>
</tr>
<tr>
<td>2001</td>
<td>Bondavalli et al.</td>
</tr>
<tr>
<td>2001</td>
<td>De Miguel et al.</td>
</tr>
<tr>
<td>2001</td>
<td>Singh et al.</td>
</tr>
<tr>
<td>2002</td>
<td>D'Ambrogio, Iazeolla, and Miranda</td>
</tr>
<tr>
<td>2002</td>
<td>Pai and Dugan</td>
</tr>
<tr>
<td>2002</td>
<td>Williams and Smith</td>
</tr>
<tr>
<td>2002</td>
<td>Yacoub and Ammar</td>
</tr>
<tr>
<td>2003</td>
<td>Folmer, van Gurp, and Bosch</td>
</tr>
<tr>
<td>2003</td>
<td>Genero et al.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>2003</td>
<td>Goseva-Popstojanova et al.</td>
</tr>
<tr>
<td>2003</td>
<td>Leangsuksun, Song, and Shen</td>
</tr>
<tr>
<td>2004</td>
<td>Addouche, Antoine, and Montmain</td>
</tr>
<tr>
<td>2004</td>
<td>Bengtsson et al.</td>
</tr>
<tr>
<td>2004</td>
<td>Cortellezza and Pompei</td>
</tr>
<tr>
<td>2004</td>
<td>Hansen, Wells, and Maier</td>
</tr>
<tr>
<td>2004</td>
<td>Zarras, Vassiliadis, and Issarny</td>
</tr>
<tr>
<td>2005</td>
<td>Cortellezza et al.</td>
</tr>
<tr>
<td>2005</td>
<td>Grassi, Mirandola, and Sabetta</td>
</tr>
<tr>
<td>2005</td>
<td>Hassan, Goseva-Popstojanova, and Ammar</td>
</tr>
<tr>
<td>2005</td>
<td>Jürjens and Wagner</td>
</tr>
<tr>
<td>2005</td>
<td>Rodrigues, Rosenblum, and Uchitel</td>
</tr>
<tr>
<td>2005</td>
<td>Thwin and Quah</td>
</tr>
<tr>
<td>2006</td>
<td>Bernardi and Merseguer</td>
</tr>
<tr>
<td>2006</td>
<td>Ober, Graf, and Ober</td>
</tr>
<tr>
<td>2006</td>
<td>van Koten and Gray</td>
</tr>
<tr>
<td>2007</td>
<td>Zoughbi, Briand, and Labiche</td>
</tr>
<tr>
<td>2007</td>
<td>Liu, Dehlinger, and Lutz</td>
</tr>
<tr>
<td>2007</td>
<td>Zoughbi, Briand, and Labiche</td>
</tr>
<tr>
<td>2007</td>
<td>Zhou and Leung</td>
</tr>
<tr>
<td>2008</td>
<td>Mustafiz et al.</td>
</tr>
<tr>
<td>2008</td>
<td>Shaik, Abdelmoez, and Ammar</td>
</tr>
<tr>
<td>2009</td>
<td>Cancilla et al.</td>
</tr>
<tr>
<td>2009</td>
<td>Wang et al.</td>
</tr>
<tr>
<td>2010</td>
<td>Siveroni, Zisman, and Spanoudakis</td>
</tr>
<tr>
<td>2011</td>
<td>Bernardi, Merseguer, and Petriu</td>
</tr>
<tr>
<td>2011</td>
<td>Faniyi et al.</td>
</tr>
<tr>
<td>2011</td>
<td>Said et al.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>2012</td>
<td>Cheung et al.</td>
</tr>
<tr>
<td>2012</td>
<td>Dubey, Rana, and Dash</td>
</tr>
<tr>
<td>2013</td>
<td>Almorsy, Grundy, and Ibrahim</td>
</tr>
<tr>
<td>2013</td>
<td>Ye, Zhu, and Wang</td>
</tr>
<tr>
<td>2014</td>
<td>Alberts, Woody, and Dorofee</td>
</tr>
<tr>
<td>2014</td>
<td>Kaur, Kaur, and Pathak</td>
</tr>
<tr>
<td>2014</td>
<td>Malhotra and Chug</td>
</tr>
<tr>
<td>2014</td>
<td>Xiao, Cai, and Kazman</td>
</tr>
<tr>
<td>2015</td>
<td>Kumar and Rath</td>
</tr>
<tr>
<td>2016</td>
<td>Stevanetic and Zdun</td>
</tr>
<tr>
<td>2017</td>
<td>Leigh, Wermelinger, and Zisman</td>
</tr>
<tr>
<td>2019</td>
<td>Li et al.</td>
</tr>
<tr>
<td>2019</td>
<td>Welfeng et al.</td>
</tr>
</tbody>
</table>
Appendix E - Risk Model Questionnaire

Part 1 – Participant Experience and Background

1. How many years of experience do you have in commercial software intensive systems engineering?

2. How many years of experience do you have in commercial software development?

3. How many years of enterprise architecture experience do you have?

4. How many years of solution architecture experience do you have?

5. How many years of technical architecture experience do you have?

6. How many years of SysML experience do you have?

7. How many years of UML experience do you have?

8. How many projects have you worked on that have involved a SysML or UML model?

9. How many years do you have working with waterfall development?

10. How many years do you have working with agile (e.g. Scrum & SAFe) development?
Part 2 – Approach Background

The research is evaluating whether risks could be described using the following model that extends ISO 42010 – Architecture Descriptions:

ISO 42010 Concept | ISO 42010 Definition
---|---
AD element | “any construct in an architecture description.” (p. 7)
Architecture | “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.” (p. 8)
Architecture Decision | “pertain to system concerns; however, there is often no simple mapping between the two. A decision can affect the architecture in several ways.” (p. 7)
Architecture Description | “work product used to express an architecture.” (p. 2)
Architecture Model | “uses modelling conventions appropriate to the concerns to be addressed.” (p. 6)
Architecture Rationale | “records explanation, justification or reasoning about architecture decisions that have been made.” (p. 7)
Architecture View | “work product expressing the architecture of a system from the perspective of specific system concerns.” (p. 2)
Architecture Viewpoint | “work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns.” (p. 2)
Concern | “interest in a system relevant to one or more of its stakeholders.” (p. 2)
Correspondence | “defines a relation between AD elements.” (p. 7)
Correspondence Rule | “enforce relations within an architecture description (or between architecture descriptions).” (p. 7)
Model Kind | “conventions for a type of modelling.” (p. 2)
Stakeholder | “individual, team, organization, or classes thereof, having an interest in a system.” (p. 2)
System-of-interest | “systems that are man-made and may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g. operator instructions), facilities, materials and naturally occurring entities.” (p. 3)
Extension Concept | Extension Definition
Risk | Sub type of Concern that represents a Risk, e.g. error-proneness or security vulnerability.
Indicator | Indicates the relative risk of a Risk. An Indicator could be a quantitative software engineering metric such as a coupling measure or a qualitative assessment by an architect.
Indicator Value | The value of a particular Indicator for a particular Risk.
Consequence | Represents a potential consequence of a Risk being left untreated.
Control | Represents an action that could be taken to reduce the potential Impact of a Risk.
Analysis Technique | Identifies the architecture analysis technique used to for a risk analysis.
Analysis Results | Encapsulates the results of a risk analysis performed using an analysis technique.
Part 3 – Approach Examples

Example 1 - Excessive Change Propagation

Text Risk Description

Title: Excessive change propagation
Details: Complex concrete sub-classes have emerged from the diverse use cases the lists had to support. E.g. SystemList needs “deleted record processing” whereas PropertyList does not. This causes conflicts between abstract class code and concrete sub-class code. This could be considered an unhealthy inheritance tree. There are also some common complex routines that are not always abstracted so when bugs have to be fixed sometimes many List sub-classes had to be changed.
Impact: Changes can be more costly and take longer than expected due to all of the changes necessary not being understood when estimating and changes are excessively expensive to implement.
Mitigations: Increase test coverage, pair programming, refactor the design
Risk Model Representation

Notes:

- Grey background elements indicate elements from the design model.
- White background elements are elements added from the proposed risk model.
Example 2 - 3rd Party Interface Changes outside of MASS control

Text Risk Description

Title: Low code framework Interface Changes outside of MASS control
Details: Oracle Data Integrator (ODI) has changed its interface specification. This will require MASS code to be reworked if ODI has to be upgraded.
Impact: Unexpected cost due to software rework to adapt ETL module code to the new ODI interfaces. Can’t take advantage of latest ODI features.
Mitigation: Don’t upgrade and accept the security risk associated with continued use of an unsupported Oracle product.

Risk Model Representation

Notes:

- Grey background elements indicate elements from the design model;
- White background elements are elements added from the proposed risk model.
## Part 4 – Risk Model Evaluation Questions

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Waterfall</th>
<th>Agile e.g. Scrum</th>
<th>Scaled Agile e.g. SAFe</th>
<th>Comments – Please include any qualifying statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.</td>
<td>Do you think the proposed risk model would help design reviews?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>Do you think the proposed risk model could help to identify risks?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>Do you think the proposed risk model could help the analysis of identified risks?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>Do you think the proposed risk model could help with the assessment of analysed risks?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>Do you think the proposed risk model could help the mitigation of assessed risks?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>Do you think the proposed risk model could help monitoring of ongoing risks?</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>Do you think the proposed risk model could be useful when a</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td>Y / N / Not Sure</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>Question</td>
<td>Answer – Please justify your answer with a brief explanation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>What do you think might be the advantages and disadvantages of modelling the risk in this way?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Which approach (textural description or the proposed risk model) do you prefer and why?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Do you think any of the entities or associations in the proposed model are unnecessary or overkill, if so which ones?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Can you think of any entities or associations that are missing from the proposed risk model?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Do you have any other feedback about the proposed risk model or its usage?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix F - Risk Model Participant Information Sheet

Human Research Ethics Committee

Research study participant information sheet - 20/07/2020

Architecture Risk Model Study
The project focuses on the risk assessment of software architectures. The questionnaire to which this participant information sheet pertains is concerned with the evaluation of an architecture risk model by practitioners. This questionnaire has been prepared to collect data for a PhD research project undertaken with the Open University by Andrew Leigh (andrew.leigh@open.ac.uk, W3444809).

The other researchers involved are Dr. Michel Wermeilinger (michel.wermeilinger@open.ac.uk) and Prof. Andrea Zisman (andrea.zisman@open.ac.uk).

Invitation:
You are being invited to take part in a research study. Before you decide whether or not to take part, it is important for you to understand why the research is being done and what it will involve. You have been invited because of your experience in Systems and Software Engineering. Please take time to read the following information carefully. Participation in this study will have no impact on you evaluation and progression at work.

General information about the research study and collected research data
This questionnaire has been prepared to collect data for a PhD research project undertaken with the Open University by Andrew Leigh. This questionnaire is concerned with the evaluation of an architecture risk model by practitioners.

Project risks are often recorded in risk registers and logs. They are typically described using text. Sometimes the risk description text is augmented with: (i) a textual impact statement; (ii) a textual mitigation plan; (iii) a financial impact estimate; and (iv) an estimate of mitigation cost. This questionnaire is collecting data to determine whether it would be beneficial to describe technical risk as a set of entities and associations alongside as part of an architectural description containing a design model (e.g., SysML or UML model).

The Architecture Risk Model Study has been reviewed by the Open University's Human Research Ethics Committee and has received a favourable opinion for the study to be undertaken.

The study will run between July and September 2020.

What will I be asked to do if I agree to take part?
To participate you will have to: a.) answer a series of questions to establish your level of experience; b.) comprehend a textual description of example risks; c.) comprehend models of the risks; d.) answer a series of questions to garner your feedback about using the proposed model. Participation is expected to take no more than 1 hour.

It is up to you to decide whether or not to take part. If you do decide to take part you will be given this information sheet to keep and be asked to sign a consent form. If you decide to take part you are still free to withdraw at any time and without giving a reason.

Participation in this study will provide you with an overall appreciation of the ISO 42010 Architecture Description concepts as well as some emerging ideas for risk model based architecting.

How will the data I provide be used?
All responses will be stored anonymously on the researcher’s personal computer. The full anonymised data set will be shared with the research community. Signed consent forms will be stored separately
from responses and for as long as the data set is kept. Data will be published in research papers and the researcher’s PhD thesis.

Your right to withdraw from the study

- You have the right to withdraw from the study at any time up to 4 weeks after returning your completed questionnaire, without having to give a reason, by emailing the research team: andrew.leigh@open.ac.uk, michel.wergelin@open.ac.uk, andrea.xisman@open.ac.uk.
- You have the right to ask for your data to be removed at any time up to 4 weeks after returning your completed questionnaire, without having to give a reason, by emailing the research team: andrew.leigh@open.ac.uk, michel.wergelin@open.ac.uk, andrea.xisman@open.ac.uk.

How do I agree to take part?
If you would like to take part please return the signed consent form to: andrew.leigh@open.ac.uk.

Thank you

Thank you for taking the time to read the information sheet.
Appendix G - Risk Model Survey Consent Form

Human Research Ethics Committee

Informed Consent for Architecture Risk Model Study

Andrew Leigh, Research Student and Computing and Communications Department

Please tick the appropriate boxes

1. Taking part in the study

I have read and understood the study information dated 20/07/2020, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction. □ □

I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time up to 4 weeks after returning your completed questionnaire, without having to give a reason. □ □

I understand that taking part in the study involves having to: a.) answer a series of questions to establish your level of architecture experience; b.) comprehend a textual description of example risks; c.) comprehend models of the risks; and d.) answer a series of questions to garner your feedback about using the proposed model. □ □

2. Use of the information in the study

I understand that information I provide will be used for publications and the researcher’s thesis.

I understand that personal information collected about me that can identify me, such as my name or where I live, will not be shared beyond the study team.

I understand that my data will be stored anonymously on the researcher’s personal computer indefinitely. I agree that my information can be anonymously quoted in research outputs. I understand that my consent form will be stored separately from my questionnaire answers on the researcher’s personal computer. I understand that the aggregated and anonymised data set will be shared with the research community.

3. Future use and reuse of the information by others

I give permission for the de-identified (anonymised) answers that I provide to be deposited in a specialist data centre after it has been anonymised, so it can be used for future research and learning.

Consent forms will be kept for as long as the research data are retained (by the researcher or an archive). The original consent forms will be digitised and stored securely (encrypted), permitting the originals to then be destroyed securely by means of shredding.

4. Signatures

________________________________________  __________________________  ________________
Name of participant [IN CAPITALS]        Signature                  Date

This Architecture Risk Model research project has been reviewed by, and received a favourable opinion, from the OU Human Research Ethics Committee - HREC reference number: 3953.

http://www.open.ac.uk/researchethics/

/  

.

234
Appendix H - Risk Model Survey Permission Letter

For the attention of:
Open University Human Research Ethics Committee
The Open University
Charlesdong Building, Level 3
Walton Hall,
Milton Keynes,
MK7 6AA

20th July 2020

Dear Sir/Madam

Re: Andrew Leigh – PhD Research Questionnaire

This letter confirms that MASS grant Andrew Leigh (MASS employee and research student at The Open University, W344/4809) permission to collect data anonymously using a questionnaire about an architecture risk model within the MASS workplace. This permission is to support the evaluation of an architecture risk model as part of his PhD and research papers.

Yours faithfully

Phil Gray
Group Head
## Appendix I - Risk Model Survey Participant Comment Coding

<table>
<thead>
<tr>
<th>Theme</th>
<th>Codes</th>
<th>Participants</th>
<th>Responses</th>
</tr>
</thead>
</table>
| **Model Advantage**, i.e.,  | Rigour| 10           | “encourage designers to think about risk”  
| an advantage of using the   |       |              | “force the system architect to model the risk and the outcome”  
| proposed model by         | Rigour|              | “forces the Ent/Soln Architect to address technical risks as they have to define and map them out”  
| comparison to textual      |       |              | “focus the engineer’s thoughts and may assist in identifying design risk areas.”  
| risk descriptions.”       |       |              | “more analysis and it is detailed.”  
|                             | Rigour|              | “uniform process across many projects”  
|                             |       |              | “ensures the process of capture, analysis and mitigation are applied.”  
|                             | Rigour|              | “uniform process across many projects”  
|                             |       |              | “forces the user to consider risks and impacts in a structured manner”  
|                             |       |              | “allows some level of rigour to be applied to defining risks and mitigations for design decisions”   |
| Traceability               | 9     |              | “Having the system modelled will allow the architect the necessary visibility of interfaces, interactions, dependencies, constraints etc to make faster risk analysis.”  
|                             |       |              | “Again, the fact that the risk is formally associated with the architecture is good.”  
<p>|                             |       |              | “I think the review process would help identify risks, I see the model more as a means of ensuring that required details are addressed.”   |</p>
<table>
<thead>
<tr>
<th>Understanding</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>“clarify where the risk is within the design”</td>
<td></td>
</tr>
<tr>
<td>“The examples given lead to a better fit (and a more complete analysis) upon existing (iteratively produced) software that is due to change. Both pieces of example software are established and require a change or refactor – thus the analysis is easier.”</td>
<td></td>
</tr>
<tr>
<td>“It clearly can be used from a traceability point of view to see the path to the identification of a specific risk. This would aid further analysis of the risk, as well as enabling better communication, and understanding around the risk.”</td>
<td></td>
</tr>
<tr>
<td>“Risk model does not help identify risks but does help identify the correct application of mitigation.”</td>
<td></td>
</tr>
<tr>
<td>“Design reviews will benefit from a clear link to architecture design and associated risk.”</td>
<td></td>
</tr>
<tr>
<td>“relating them to the decision / rationale.”</td>
<td></td>
</tr>
<tr>
<td>“it is likely that the output of the work will help to understand the mitigation options.”</td>
<td></td>
</tr>
<tr>
<td>“I think that the risk model will help to understand the risk but not necessarily identify the risks. I think that the identification of risk sits outside of the proposed risk model, however the risk model will provide the means to assess the risk and the impact to the system.”</td>
<td></td>
</tr>
<tr>
<td>“Unambiguous description of the risk. Model provides a design for a software application to manage risk aspects – way better than an Excel spreadsheet. I prefer the risk model, as I am a visual learner. The model has a “syntax” that provides more precision than a purely textual description.”</td>
<td></td>
</tr>
<tr>
<td>“Bringing a UML risk notation to design review would help explain design decision mitigations”</td>
<td></td>
</tr>
<tr>
<td>“identifies high risk areas in delivering against requirements/concerns”</td>
<td></td>
</tr>
<tr>
<td>“Clear view of risk.”</td>
<td></td>
</tr>
<tr>
<td>“The Risk Model allows the user to visualize the paths/scenarios and consider all potential events/impacts”</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Count</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Risk Patterns</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Automation</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Early Mitigation</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>1</td>
</tr>
<tr>
<td>Report Generation</td>
<td>1</td>
</tr>
<tr>
<td>Model Applicability, i.e., architecture scenarios where the model is perceived to add value to the task of architecting.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>High Integrity</strong></td>
<td>2</td>
</tr>
<tr>
<td>“I think it will be highly beneficial, especially for systems/developments with high integrity, safety, security requirements or compliances.”</td>
<td></td>
</tr>
<tr>
<td>“I think that this would potentially benefit ‘safety critical’ or ‘financial’ software systems where non-functional requirements are as important if not more important than functional requirements due to the level of analysis work required.”</td>
<td></td>
</tr>
<tr>
<td><strong>Requirements Analysis</strong></td>
<td>2</td>
</tr>
<tr>
<td>“If the ARModel was extended to include Requirements then it would be applicable when assessing Use Cases etc. within e.g. Enterprise Architect modelling of requirements.”</td>
<td></td>
</tr>
<tr>
<td>“Believe it would be of most benefit in reviewing SOR’s and deriving subsystem requirements at bid stage in large architectural programs.”</td>
<td></td>
</tr>
<tr>
<td><strong>Not just design risks</strong></td>
<td>2</td>
</tr>
<tr>
<td>“It makes sense and could be extended to cover other non-functional areas of architecture design.”</td>
<td></td>
</tr>
<tr>
<td>“Not all risks are associated with design. Modelling of risk should start from day 1 of a project.”</td>
<td></td>
</tr>
<tr>
<td><strong>Technical Debt</strong></td>
<td>1</td>
</tr>
<tr>
<td>“Is the aim really to manage technical debt?”</td>
<td></td>
</tr>
<tr>
<td><strong>Agile Adoption</strong></td>
<td>1</td>
</tr>
<tr>
<td>“Putting the how and why further into the formal class model (rather than into whiteboards, JIRA instances, Jupyter Notebooks, Slack rooms, Confluence pages etc)”</td>
<td></td>
</tr>
<tr>
<td><strong>Architecture Recovery</strong></td>
<td>1</td>
</tr>
<tr>
<td>“Using the risk model in place of design would be a good fit for a scenario where there is a “legacy black box” that has little-to-no-design that lives deeply rooted in a complex enterprise application (e.g. a banking model)”</td>
<td></td>
</tr>
<tr>
<td><strong>Change Impact Analysis</strong></td>
<td>1</td>
</tr>
<tr>
<td>“I believe this would be best used in early design or managing architectural change through additional requirements/concerns”</td>
<td></td>
</tr>
<tr>
<td><strong>Decision Support</strong></td>
<td>1</td>
</tr>
<tr>
<td>“To assist in the comparison of possible design models from a risk perspective.”</td>
<td></td>
</tr>
<tr>
<td><strong>Operational Risk</strong></td>
<td>1</td>
</tr>
<tr>
<td>“It appears that the risks are at the project level - of bringing the system of interest into being or managing it...”</td>
<td></td>
</tr>
</tbody>
</table>
| Analysis | Process | **5** | “If the Design Review and its technical risk review activity also re-assesses pre-assessed AD architecture”
“It may be more effective with regular integrative architecture and design approach.”
“If risks are assessed at the outset, then these may be reassessed as requirements(concerns)/design changes and the potential impact.”
“As a review against implementation/decision decisions and technologies perhaps”
“I am not sure if having a model to describe them makes much practical difference but could work if the team or programme adopts a design review framework which ensures the designers do consider risks and challenges them if they have not.” |
| --- | --- | --- | --- |
| Training | **5** | “If the model is understood by all participants and is repeatable then it may help the design process”
“A reviewer needs to know the syntax of the AD language when a graphical model is used.”
“Participants need some relevant experience to understand the model and the modelling language.”
“I do not see guidance on the assessment of risks other than mitigations being applied.”
“It also requires an understanding of model-based systems/software engineering.” |
| Text Needed | **3** | “Hard to say they both back each other up, for a better understanding.”
“Text based approach is also very quick to read”
“the descriptions for presentation purposes.” |
| PM Integration | **3** | “as opposed to being tucked away in a separate log and only brought out by the PM at big reviews.” |
| Flexibility | 2 | “It will require ‘buy in’ from the normal project risk holders, but technically I think this is great approach to a very important area of software/system Engineering.”
“I think it might help with ongoing reassessment of risk due to design changes. But monitoring of risk is really a project management function I don’t see this necessarily helping with that.”
“May not fit every project, may need tailoring.”
“The model should be used as reference or guide material and should not be applied as a rigid framework to work by. So, no. I think all elements have merit in appropriate contexts.” |
| Risk View | 2 | “having them as separate views would help.”
“I would propose that the “Architecture View” concept is used to partition the overall risk modelling” |
| Tooling | 2 | “Also, the AD model will need to be supported by a technical risk modelling entity/register to hold the outputs of the AR modelling and analysis activity. I don’t recall ever seeing such an entity in an Enterprise Architect model to date.”
“Definitely, especially with some software tooling.” |
| Extensibility | 1 | “I think the model appears ok, but users of the model should have scope to add and remove as necessary. It is hard to get a one size fits all model.” |
| Needs Testing | 1 | “This would have to be exercised to see if any are overkill.” |
| Overwhelming | 1 | “My worry is that if not caught early enough, the amount of identified risks may become daunting to deal with.” |
| Model Disadvantage, i.e., a disadvantage of Effort | 7 | “Due to the complexities of risk, the cost of modelling and having models”
“Can’t really see too many disadvantages other than maybe the cost involved in modelling risk?” |
| Cluttering  | 5 | “Disadvantage is that there is an obvious cost involved as the ADModel is significantly enlarged to encompass ARModelling elements,”  
“Might create a bigger overhead than it’s worth to deal with technical debt.”  
“May be over complex for smaller short-lived projects.”  
“To generate the models in sufficient detail has a higher overhead than a traditional “risk register”  
“The main issue that I see is that there is potentially a lot of upfront work required.”  
“There is a risk that the risk model could get extremely complicated in a real-life scenario making it difficult to comprehend.”  
“I think it will get overwhelming quickly”  
“However, it does clutter the design and therefore understanding of the design.”  
“Discipline required as too many “mitigations” leads to a complex diagram – or (as shown) an “impact” affects many things.”  
“The diagram approach is great for smaller tasks but, it can quickly overwhelm the reader for larger tasks with multiple related risks.” |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraining</td>
<td>1</td>
<td>“but reliance on the model could cause users not to think outside the scope of that model”</td>
</tr>
<tr>
<td>Misuse</td>
<td>1</td>
<td>“Or, if a system architect did not model all available mitigations and steered the model from the wrong risk perspective perhaps?”</td>
</tr>
<tr>
<td>Out of date</td>
<td>1</td>
<td>“With agile I believe that if the risks were identified then they would be addressed quickly by the development team so monitoring of extant risks may be nugatory.”</td>
</tr>
</tbody>
</table>
| Missing Aspect, i.e., something the participant expected to be in the risk model but could not find. | Probability | 5 | “The model does not appear to indicate the likelihood of the risk being realised.”  
“NB: there’s no mention of the Probability aspect of risk assessment though…”  
“At the risk of making it even more complex, I would like to have seen probability addressed directly.”  
“likelihood”  
“a way of expressing probability” |
| --- | --- | --- | --- |
| Impact Level | 5 | “impact score”  
“impact level”  
“Impact could specify cost, effort, delay, capability loss etc as attributes? They could also be split out further as entities?”  
“Not sure impact is accounted for enough in the model.”  
“Perhaps some kind of ‘score’ against impact and/or mitigation. This may help with decision making or justifying work to change design/implementation” |
| Status | 4 | “I think that if we had a way of categorising if the concern still remains post mitigation, or solved, then it would allow management, monitoring or later remedial action as part of the risk management process.”  
“Risk Monitoring appears to be out of scope of the ARModel (Part 2 top diagram) – there’s no link or trigger for when the AD and ARModel outputs are actively reviewed and assessed through lifecycle stage/design reviews (PDR/CDR or Integration Readiness reviews) or Sprint Retrospectives, etc.”  
“With a recorded status for each risk that is kept up to date as mitigation is applied, indicators change and so on.”  
“Not sure all the information is present for continual monitoring of risks. Another model may be better” |
<table>
<thead>
<tr>
<th>Component</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mitigation Cost</td>
<td>2</td>
</tr>
<tr>
<td>Mitigation Effect</td>
<td>2</td>
</tr>
<tr>
<td>Risk Category</td>
<td>2</td>
</tr>
<tr>
<td>Alt. Mitigations</td>
<td>1</td>
</tr>
<tr>
<td>Compounding Risks</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td>1</td>
</tr>
<tr>
<td>Impact Results</td>
<td>1</td>
</tr>
<tr>
<td>Overall Risk Score</td>
<td>1</td>
</tr>
<tr>
<td>Mitigates Concern</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Mitigation Cost**: “Perhaps cost of mitigation should be included to allow cost/benefit analysis to be calculated on a per-risk basis.”
  - “Cost – Linked to mitigation. Many mitigations impose some sort of cost.”

- **Mitigation Effect**: “There should perhaps be a weighting mechanism applied as the potential effect of mitigation and effect of the risk is not considered.”
  - “Perhaps some kind of ‘score’ against impact and/or mitigation. This may help with decision making or justifying work to change design/implementation”

- **Risk Category**: “safety / security criticality could all be covered as indicators.”
  - “Risk Groups – I’m sure risks can be grouped, into different groups. The relationship would need to be many to many.”

- **Alt. Mitigations**: “This would probably necessitate the need to model alternative mitigations.”

- **Compounding Risks**: “Risks linked to other risks – Risks can be associated with other risks. E.g. Through increased likely hood after risk another risk has occurred etc.”

- **Events**: “inclusion of Events when risks have happened would allow better analysis and mitigations.”

- **Impact Results**: “impact results’ seems undercooked.”

- **Overall Risk Score**: “overall risk score”

- **Mitigates Concern**: “I wonder if there needs to be a direct relationship to a mitigation for the concern element?”
<table>
<thead>
<tr>
<th>Related Concern</th>
<th>1</th>
<th>“I find it to be able to ‘value’ a risk without the impact relating to a stakeholder concern.”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk Conflicts</td>
<td>1</td>
<td>“Also, we have only looked at instances in isolation; like with static analysis, it is possible that there may be conflicts across specific reported risks.”</td>
</tr>
<tr>
<td><strong>Redundant</strong>, i.e., something available in the model that the participant considered to be unnecessary.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis Technique</td>
<td>2</td>
<td>“The analysis technique seems overkill.”&lt;br&gt;“The sources and justification of the risk – embodied in the Stakeholder and the Analysis Technique and Analysis Results could be out of scope if we just want to model the risks and their mitigations.”</td>
</tr>
<tr>
<td>Analysis Results</td>
<td>2</td>
<td>“The sources and justification of the risk – embodied in the Stakeholder and the Analysis Technique and Analysis Results could be out of scope if we just want to model the risks and their mitigations.”&lt;br&gt;“analysis results’ are unnecessary.”</td>
</tr>
<tr>
<td>Architecture Rationale</td>
<td>1</td>
<td>“architecture rationale would be better being an attribute of architecture decision”</td>
</tr>
</tbody>
</table>