Objects Versus Components
A Contraposition To Component-based Software For The Avionics Domain

A dissertation submitted to The University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences.

2013

By
Robert Milburn Stanistreet

School of Computer Science
# Table of Contents

**Abstract** 9  
**Declaration** 10  
**Copyright** 11  
**Acknowledgements** 12  

**Chapter 1 Introduction** 13  
1.1 Structure 15  

**Chapter 2 Overview of Historic Developments** 16  

2.1 The Arguments For Components 17  
2.1.1 Objects Were Not Sufficient 17  
2.1.2 Why Are Objects Not Enough? 18  
2.1.3 What Is A Component? 19  
2.1.4 Inheritance 20  
2.2 The Counter Argument 21  
2.2.1 GRASP Principles 21  
2.2.2 High Cohesion And Low Coupling 22  
2.2.3 Design Patterns 23  
2.2.4 Class Inheritance Versus Object Composition 24  
2.2.5 Rules For Inheritance 25  
2.2.6 Other Forms Of Good Practice 26  

**Chapter 3 Current Component-Based Software Development** 27  

3.1 Component Model 27  
3.1.1 Components 28  
3.1.2 Composition Mechanisms 29  
3.2 A Component Life Cycle 29  
3.2.1 Phases Of An Idealised Component Life Cycle 31  
3.3 X-MAN 33  
3.3.1 Component Model 33  
3.3.2 Composition Connectors 33  
3.3.3 Encapsulated Components And Connectors 36  
3.4 Further Explanations And Observations 38
CHAPTER 4  BRAKING SYSTEMS  

4.1  Braking
4.1.1  Coefficient Of Friction
4.1.2  Antilock Braking Systems

4.2  Avionics ABS
4.2.1  System Analysis
4.2.2  Control Loop Feedback
4.2.3  Tuning

CHAPTER 5  IMPLEMENTATION

5.1  Design
5.1.1  Redundant Data
5.1.2  Final Implementation Structure

5.2  Overview Of The Java Re-Implementations
5.2.1  Data Passing Mechanism
5.2.1.1  ExtendedArrayList
5.2.1.2  The Use Of The ExtendedArrayList
5.2.1.3  Criticisms
5.2.1.4  Complexity
5.2.1.5  Performance
5.2.2  GeneralFramework
5.2.2.1  A Variation Of The Composite Pattern
5.2.2.2  XMANPart
5.2.2.3  AtomicComponent
5.2.2.4  CompositionConnector
5.2.2.5  Pipe And Sequencer Connectors
5.2.2.6  CompositeComponent
5.2.2.7  The Application Of The Generic Framework
5.2.3  Utilities
5.2.3.1  ExtendedArrayList
5.2.3.2  Message
8.1.6 Tool To Support The Idealised Component Life Cycle 108
8.1.7 Wider Applicability 109
8.1.8 Broader Questions 109

APPENDICES 110
Appendix A 110
Appendix B 111
Appendix C 113
Appendix D 114
Appendix E 115
Appendix F 121
Appendix G 122

REFERENCES 123

WORD COUNT

Final word count, including footnotes and endnotes: 28,693
LIST OF FIGURES

Figure 1  The component-based software development desiderata by Broy et al. (1998) formalised into distinct development phases (K.-K. Lau & Taweel, 2011). 30

Figure 2  An example of how a component development might feed into traditional sequential software development processes (K.-K. Lau & Taweel, 2011). 30

Figure 3  The Categories other component models (1-4), but only the most idealised (Category 5) uses exogenous composition (K.-K. Lau & Wang, 2007). 31

Figure 4  An idealised life cycle, which fully meets the requirements of the design, deployment and run-time phases (K.-K. Lau & Wang, 2007). 32

Figure 5  Direct message-passing using method calls (K.-K. Lau et al., 2005, p. 2). 33

Figure 6  Indirect message-passing connectors with method calls (K.-K. Lau et al., 2005, p. 4). 34

Figure 7  The message passing of exogenous connectors (K.-K. Lau et al., 2005, p. 5). 35

Figure 8  The control and data flow resulting from the use of exogenous composition connectors (K.-K. Lau & Elizondo, 2010, p. 1166). 35

Figure 9  Various component models and how they are able to support the key characteristics of an idealised component lifecycle (K.-K. Lau & Wang, 2007). 36

Figure 10  The two encapsulated (a) atomic, (c) composite components and (b) composition connector (K.-K. Lau & Rana, 2010, p. 6). 36

Figure 11  The three basic composition connectors and their behaviours: (a) sequencer, (b) pipe and (c) selector (K.-K. Lau & Elizondo, 2010). 38

Figure 12  The Java-based version developed by Elizondo (2008 ~ 2009). 39

Figure 13  The C-based version of the X-MAN tool, showing the designer view (MBCSDRG, 2011). 40

Figure 14  Through the use of exogenous composition connectors, the X-MAN component model produces a tree structure (K.-K. Lau et al., 2005). 41

Figure 15  An abstract representation of an anti-skid braking system, including the various interacting parts (adapted from Ludovic, n.d.). 48

Figure 16  The control loop feedback of the Nalmpantis’ PID controller (adapted from Nalmpantis, 2011, p. 31). 49

Figure 17  The three subsystems that makes up the previous anti-lock braking system (adapted from Nalmpantis, 2011, p. 58). 51
Figure 18 An example of redundant data being passed from Sensors 1, 2 and 3 to Components 1 and 2 (adapted from Nalmpantis, 2011, p. 59).

Figure 19 Data is passed directly to the component as and when it is required, thus reducing data redundancy (adapted from Nalmpantis, 2011, p. 60).

Figure 20 The final C-based X-MAN implementation of an ABS system by Nalmpantis (2011, p. 69).

Figure 21 The class diagram ExtendedArrayList extending ArrayList containing OrderedPair elements.

Figure 22 An overall class diagram of the general framework that is a representation of the core parts of the X-MAN component model.

Figure 23 The XMANPart class diagram.

Figure 24 The AtomicComponent class diagram.

Figure 25 The CompositionConnector class diagram.

Figure 26 A diagram showing Sequencer and Pipe that inherits from CompositionConnector.

Figure 27 The CompositeComponent class diagram.

Figure 28 Through the use of exogenous composition connectors, the X-MAN component model produces a hierarchical tree structure.

Figure 29 The generic ExtendedArrayList class diagram and its dependencies.

Figure 30 The Message class diagram.

Figure 31 A diagram of the interface atomic component classes that inherit from the superclass AtomicComponent.

Figure 32 A diagram of the functional and simulation atomic component classes that inherit from the superclass AtomicComponent.

Figure 33 The structure of the PID composite (adapted from Nalmpantis, 2011, p. 88).

Figure 34 A diagram of the PID atomic component classes that inherit from the superclass AtomicComponent.

Figure 35 The simplified re-implementation of the original Nalmpantis (2011) ABS system.

Figure 36 A UML class diagram showing the association between ControlUnit and the Sensor and Cockpit interface classes.

Figure 37 The results of a comparison between aircraft and wheel speed Ming, Hong, and Rupeng (2011) using a more complex P-PBM-D controller (modified by Nalmpantis, 2011, p. 75).

Figure 38 The first simulation over time covering 17 second.
Figure 39 The first simulation over time covering 12 second with an increase in frequency at which the controller polls for the measurements. 85

Figure 40 The final simulation over time covering 12 second. 86

Figure 41 A ‘Category 1’ component life cycle also known as ‘Design without Repository’ (K.-K. Lau & Taweel, 2011). 97

Figure 42 The re-implementation of the final Nalmpantis (2011) structure, with more repetition of operations and a deeper, more complex tree structure. 114

Figure 43 The Director, the abstracted CompositeBuilder and the derived ConcreteCompositeBuilders. 115

Figure 44 The UML class diagram of the interface director and concrete builders that various interface composites. 117

Figure 45 The creation of the three different Interface composites using the same Director, by varying the key name and the builders used. 118

Figure 46 The UML class diagram of the director and concrete builder that creates the LowMedMax composite. 119

Figure 47 The UML class diagram of the director and concrete builder that create the Simulation composite. 119

Figure 48 The exogenous connectors component model for X-MAN (Kang, 2010, p. 27). 122
LIST OF TABLES

Table 1  An example of atomic component from the Nalmpantis (2011) C-based implementation. 43
Table 2  An example of the use of ExtendedArrayList. 59
Table 3  The invoke method of the Pipe class. 67
Table 4  An example of XML test file containing the pre-set inputs and expected outputs required for simulation (Nalmpantis, 2011, p. 73). 81
Table 5  The reproduction of the results of the original C-Based version, with the X-MAN-like and OO Java re-implementations showing the same results. 83
Table 6  Cohesion, coupling and size metrics gathered from the two implementations. 88
Table 7  An exemplar of an atomic component, called increment (Tran, 2012). 92
Table 8  The invoke method of the Sequencer class. 110
Table 9  An output of the Message class for a simple test using the X-MAN re-implementation in Java. 112
Table 10  An output of the Message class for a simple test using the object-oriented re-implementation in Java. 113
Table 11  A code example of how the interface composites are created, including: 1) aircraftSpeed, 2) wheelSpeed and 3) touchDown. 118
Table 12  The reproduction of results of the tests for the XMAN-like and OO Java re-implementations with the original Nalmpantis (2011) C-based version. 121
ABSTRACT

The project tested the argument made by component advocates that “objects are not enough” and that a switch to a component-based methodology is necessary for software engineering to reach maturity. To evaluate this, a previous component-based application that was implemented using the X-MAN component model and tool, has been re-implemented using the Java object-oriented language. The project assesses the component-centric idea that exogenous composition enables greater code reuse, but also outlines some of the implications and limitations of the rigidity of this approach.

The dissertation outlines some of the counter-arguments on behalf of objects. It describes how some of the original component claims are a result of previous misunderstandings of the object paradigm. It also establishes that due to the flexibility of objects, misuse can still occur, but that power and usefulness outweigh all of these issues in the context of the current application. There is also evidence that through the application of good patterns and principles that aids appropriate object-oriented design, more intuitive representations of the modelling of the domain and code reusability will naturally follow.
DECLARATION

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
COPYRIGHT

1) The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

2) Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

3) The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the dissertation, for example graphs and tables (“Reproductions”), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

4) Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s Guidance for the Presentation of Dissertations.
ACKNOWLEDGEMENTS

I would like to take this opportunity to express my profound appreciation to my project supervisor, Dr. John Sargeant, for his erudition, encouragement and time. His guidance and feedback have been invaluable throughout this project.

I would also like to thank my dyslexia study support tutor, Jane Samuels, for her kindness, moral support, endless patience and helping to greatly improve my writing.
INTRODUCTION

The object-oriented (OO) and component-based software paradigms are two differing underlying theories and methodologies. The object model has the core concept of dynamic objects representing any entity with a state and behaviour, which are instantiated from static classes and code reused through the inheritance mechanism. The component ideal is explicitly focused on the theory of dynamic composition, enabling greater code reuse of pre-existing software units, known as components.

The proponents of the two paradigms argue two opposing positions:

1) The argument for components originates with the properties and methodology of objects being unable to sustainably support code reuse. This results in programs that are built constantly from scratch, leading to unreliable applications that are of greater risk of failure. Only through the use of components and the idea of reusability through composition, can software engineering overcome these problems, drive down costs and finally become a fully mature discipline.

2) The object counter-argument states that objects are sufficient and have developed considerably over the last two decades. It is recognised that problems in the past did occur due to misunderstanding and misuse; however, these issues can be overcome through application of good practices, such as generally accepted principles and patterns, which support suitable and reusable object-oriented designs. Further, many of the arguments of components no longer apply because the object model already supports many of the purported advantages.

Recently the component claim has been strengthened with the conceptualization of the idealized component lifecycle. Through this concept,
and from viewing previous technologies through the perspective of the component model, component-centrists aimed to demonstrate that objects could not fundamentally achieve the same levels of reusability as components. This is because of the use of inheritance and traditional direct data passing mechanisms that give objects too much control.

A component model called X-MAN is able to meet the idealised component life cycle. It achieves this through the use of external connectors as means of composition, which decouples the invocation control from the computation. This results in highly modular structures that allow control and computation both to vary independently and is said to be reusability in all the phases of the component lifecycle.

To assess the various claims and counter claims outlined above, a series of research questions will be answered:

1) What contrasts and comparisons can be drawn between X-MAN and X-MAN simulated in Java?

2) What are the advantages and disadvantages of a X-MAN-like Java re-implementation with a plain object-oriented re-implementation?

3) How helpful was inheritance in aiding the development, given that the X-MAN component model does not support this form of code reuse?

To answer these questions, this project re-implements an anti-lock braking system (ABS) that was developed using X-MAN, but using static classes and dynamic objects instead.

Nalmpantis (2011) developed the original system in the C-based version of the X-MAN tool, which is the embodiment of both the X-MAN component model, and the idealised component life cycle.

The re-implementation is developed in the Java programming language and includes various forms, including: (1) two X-MAN-like re-implementations that imitate concepts of the X-MAN model and tool, and (2) an object-oriented re-implementation that is used for comparisons. All of the re-implementations
are used to explore the distinctions and similarities between the object-oriented and X-MAN component-based approaches.

The justifications for the re-implementations as part of this dissertation are to test the fundamental premise that objects are insufficient, to understand the idea of component reuse and composition and to demonstrate good object-oriented design.

1.1 Structure

This dissertation will seek to identity relative pros and cons between objects and components through the following work:

Chapter 2 will explore the history of the various arguments for and against both components and objects.

Chapter 3 offers definitions for concepts of current research into components, specifically detailing the X-MAN component model.

Chapter 4 provides an explanation of braking systems. Further, how the avionics anti-lock braking system (ABS) was designed and developed by a Nalmpantitis (2011) using the X-MAN model and tool. This work now forms the basis for the current project.

Chapter 5 details the three re-implementations of the anti-lock braking system in Java, including a framework used to support the representation of the two re-implementations of the ABS system. An object-oriented re-implementation was developed as a means of comparison.

Chapter 6 assesses whether the re-implementations are able to achieve the same results as the previous ABS implementation through various simulations. Metrics were gathered on the various approaches, to evaluate them in terms of object-oriented design principles.

Chapter 7 compares and contrasts the claims and counter claims, and tries to answer a set of research questions using the understanding developed during the development of this project.
OVERVIEW OF HISTORIC DEVELOPMENTS

Introduction

Modern object-oriented programming methodologies allow a system to be broken down into logical software units (Sargeant, 2012). This has introduced numerous benefits to software engineering, but the primary driver is code reuse.

However, there is a subset of the software engineering community who believe that this approach is insufficient. Instead, greater software constructs, consisting of multiple smaller units, are required to meet the task of writing ever larger and more complicated distributed systems. Among these is a branch known as the component-based software engineers, who adopt both reuse and product-centric approaches. Their objective aims to bring about the rigours of the physical engineering discipline to software development, through the component paradigm.

The component movement points to a number of reasons why its involvement is necessary; however, the principal raison d’être, and the specific motive for the current project, is that it is claimed that the object-oriented approach is inadequate.

This argument is not universally accepted, and the current project aims to investigate this claim within the context of a particular model and application. However, first it is necessary to outline the background for both arguments:

1) The supporting argument for components is that the object-centric approach is too limiting, and that the use of composition instead of inheritance offered more benefits. However, this has not brought the fundamental paradigm shift that had been expected. The movement found it necessary to define its concepts and practices, in order to distinguish them from the perceived
blurring of terminologies and ideas. These classifications lead to current component research, and ultimately to what some believe to represent the purest component-model, which through versions of a tool known as X-MAN, supports an idealised component lifecycle.

2) This report seeks to balance this position by examining some established counter-arguments. It asserts that mechanisms for code reuse, management, composition and development were, and still are, present and that software engineering is already a mature discipline. Further, through correct practice of generally applicable principles, such as good software design, most of the advantages that components purport to have, already exist in the object-oriented paradigm, or can be adapted to emulate properties that components are said to possess.

2.1 The Arguments For Components

2.1.1 Objects Were Not Sufficient

The component movement is considered to be a consequence of the software crisis, and the inability of traditional paradigms, such as procedural, functional and object-oriented, to overcome it. A paper presented by Pfister and Szyperski (1996) is cited by proponents, such as K.-K. Lau and Rana (2010) and Nalmpantis (2011), as a starting point for their constitution and justification of the movement within academia and the software industry.

The authors wished to emphasize two important aspirations:

1) The creation of a viable component software market, and the component-oriented software development needed to create and develop that software.

2) Software development maturing into a fully developed engineering field, where software, is not produced from scratch, but built and modified using readily available units. This will result in an inevitability of software componentisation and standardisation.
Object-oriented programming had not yet met the two objectives stated above. This was because of the unique characteristics that component software and component-based programming require.

2.1.2 Why Are Objects Not Enough?

Pfister and Szyperski (1996) argued that objects are too narrowly focused at the object-construct level. As a result, it is necessary to look to higher levels of abstraction to resolve the requirements of components and the issues associated with the object-centric perspective. These higher-level concepts lead to the meaningful definition of components as collections of related objects.

As a consequence of this classification, a component-oriented language and programming must meet the requisites of:

1) Stricter information hiding or encapsulation at the component or grouped objects level.

2) A more dynamic approach, specifically late-linking. Traditionally this is the process that allows the dynamic loading and linking of externally shared libraries such as resource DLLs. This corresponds to the deployment phase in the X-MAN model, which will be discussed later.

3) Increased safety properties. Szyperski (2002) explained that a safe language must meet various safety levels such as module, memory and type safety.

This would allow better support for the concept of software units and an increased reusability between components. Feasible component software markets would logically follow, allowing consumers to acquire and integrate pre-existing software functionalities as packaged and distributed components.

As mentioned, object-oriented languages did not facilitate the three requisites listed and therefore, the authors’ two aspirations. One capability that Pfister and Szyperski (1996) omitted from their requirements, but object-oriented languages do support, is inheritance. The authors challenged that inheritance
was little understood, and that there were more benefits in terms of reuse, due to a reduction in coupling, if the functionality of two components were simply composed together rather than extended.

Jell (1998) later collected the papers presented at the first Computer Users Conference (CUC’96) into a book in which he succinctly summarises the Pfister and Szyperski (1996) position:

“Today, the beginning of a shift from object-oriented programming to component-oriented programming can be observed. Component-oriented programming provides better support for whole collections of objects, for dynamic loading of classes and for safety, and allows better decoupling between components than does object-oriented programming. Frameworks follow this trend by shifting support away from traditional close applications to truly modular, open component assemblies.” (Jell, 1998).

2.1.3 What Is A Component?

Towards the end of the 20th century, the anticipated shift towards the precedence of components had not resulted in the form that had been expected. Instead, in an article called Components vs. Objects vs. Component Objects, Szyperski (1999) stated that terminologies were being confused and object advocates were attempting to rebrand objects as components.

This confusion between objects and components led Szyperski (2002) to define the necessary characteristics of a component. To do this, the concept of software units of various aspects of scale and abstraction was used. Each unit has properties that are consistent throughout its scope. An example of a software unit is an object, which is defined as a unit of instantiation. Other software fragments are based upon many categorized properties such as “separate development, separate static analysis, separate compilability, separate delivery and deployment, and so on.” (p. 568).

Within the context of this categorisation, components differ from objects in a number of ways, in which two are central:
1) A component is at least an independently deployable unit; decoupled from environments and other components.

2) A component is a unit of third-party composition that “encapsulates the implementation” and clearly states an interface and dependencies.

The key property is that components should enable composition, described as units of composition (Szyperski, 2002). The core reason for composition is reuse (K.-K. Lau & Rana, 2010) which is achieved by the independence and the execution of rules of unit. A software unit is a component, whether it is a procedure, class, module or an entire subsystem (Szyperski, 2002), as long as the unit is composable and has the properties outlined above.

2.1.4 Inheritance

As well as outlining what components are and what they are not, Szyperski (2002, pp. 109-138) went further by highlighting past issues with inheritance.

Inheritance is one of the core concepts of the object paradigm, and has two main advantages: “it is a powerful modelling tool, because it gives a concise and precise description of the world and it helps in factoring out shared specifications and implementations in applications.” (CMU, 1995).

Despite these benefits, two decades ago when the object paradigm was being applied widely, the concept of inheritance was often misused. Overuse of mechanisms such as multiple inheritance, resulted in unstable, complex and fragile designs that were not able to adapt readily with change.

However, Szyperski (2002) admitted that some “of the problems can be avoided by adopting a highly disciplined approach to inheritance”. The guidelines for best practice such as GRASP principles and Design Patterns, which will be discussed subsequently, facilitate exactly this.

Despite these admissions, Szyperski (2002) posed the question ‘Is inheritance just simply a current trend?’; while maintaining that inheritance should be avoided altogether (p. 116). He suggested using “object rather than class composition” instead.
2.2 The Counter Argument

The practice of object-oriented (OO) development has changed significantly since the original motivation. Some believe their motivation was partly due to the way the OO programs were commonly written in the past, which very commonly led to problems due to misunderstandings and malpractice.

Good practice did exist, but often it was implied, fragmented and only known about by the most experienced practitioners and academics. The result of this was systems built with objects that had little or no reusability, which was highlighted by component proponents and ultimately motivated the component-based paradigm.

However, a few key publications formalised identifiable patterns and principles of what is generally considered good OO design. Books such as *Applying UML and Patterns* by Larman (2005) and *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (1995) have resulted in acceptable practices that have become commonplace.

2.2.1 GRASP Principles

The *General Responsibility Assignment Software Patterns* (GRASP) are a set of 9 nameable principles used to aid the understanding and reasoning of good object-oriented design in a methodical, rational and explainable way (Larman, 2005, p. 277).

The GRASP principles help the assignment of general responsibilities to classes in an object design, known as responsibility-driven design (p. 273). Classes can be reorganised, restructured, obligations are assigned and classes added in order to adhere to the principles. However, good OO design is a non-trivial task to implement correctly because of the various flexibilities, variations and trade-offs to consider.

There are alternative sets of principles, such as including the mnemonic acronym SOLID, discovered by Martin (2003, pp. 95-145).
2.2.2 High Cohesion And Low Coupling

Cohesion and Coupling are the most fundamental of all the 9 GRASP principles (Sargeant, 2012) because the others are a direct consequence of their properties and effects.

The strength of how an element is connected to, has knowledge of, or relies on another element is described by Coupling (Larman, 2005, p. 299). The measurement is not simply the number of connections, but also takes into account the complexity of those connections (Sargeant, 2012). Through application of principles such as Low Coupling (the minimal strength of connections, dependencies or knowledge between objects) to OO design will lead to software that provides “lower dependency, low change impact and increased reuse” (p. 299).

The Low Coupling principle aims to minimise the dependencies between the two elements. Dependencies between class elements include attributes, class and interface inheritance, message sending and returned parameters (Larman, 2005, p. 261). As the dependencies between classes increase, they become more difficult to change, comprehend and reuse.

Cohesion is the measurement of how well an object is able to be focused, understandable, manageable and is able to be supportive of Low Coupling (Larman, 2005, p. 291). High Cohesion is achieved by assigning responsibility to a class so that the it represents a single, well-defined entity (Sargeant, 2012). A class that tries to do many dissimilar things has Low Cohesion and should be broken down into many separate classes (Larman, 2005, p. 315). Elements that have Low Cohesion are difficult to understand, reuse, maintain and are more likely to be affected by change (p. 315).

There are alternatively phrased principles that are not part of GRASP, which include:

1) Don’t Talk to Strangers (Lieberherr, Holland, & Riel, 1988), also known as the Law Of Demeter (Bock, 2000), is a principle whereby each software unit should only ‘talk’ (message pass) and have knowledge to its ‘friends’ that are...
closely related, and not to ‘anyone’ else. This reduces coupling and the impact of change.

2) Don’t Repeat Yourself (Hunt & Thomas, 1999) and Do Each Thing In Exactly One Place (Sargeant, 2012) are principles that avoid the repetition of the same or similar functionality across a system and avoids code redundancy.

3) You Aren’t Going To Need It (Jefferies, Anderson, & Hendrickson, 2000) is a principle in which code for a feature is not written unless there is a specific requirement for it. This then prevents time and money being wasted supporting that feature (Jay & Stevens, 2013).

The different re-implementations for this project will primarily be evaluated in terms of Cohesion and Coupling.

2.2.3 Design Patterns

Gamma et al. (1995), who are collectively are known as the “Gang of Four” (GoF), authored a highly influential book called Design Patterns. This book catalogues 23 patterns, or “elements of reusable object-oriented software”, that can offer enduring and well-engineered solutions to recurring problems in object-oriented software design. The designs “systematically name, explain, evaluate” and capture best practices, offering examples from real-world applications and systems, not just claims or arguments. When applied, the patterns deliver more “flexible, elegant and ultimately reusable designs” for object-oriented software (Gamma et al., 1995). Frequently conveyed in Unified Modelling Language (UML), they are also high-level instruments for communication whose names substitute for explanations between developers (Sargeant, 2012).

The designs were inspired by the identification of common designs in architecture (Alexander, Silverstein, & Ishikawa, 1977). The GoF book assumes previous comprehension of object-oriented software design and strong programming ability, specifically C++ and Smalltalk (Larman, 2005, p. 280).
Most of the Design Patterns are based on the GRASP principles, such as High Cohesion and Low Coupling, but that is not the way that it was presented at the time. Historically, the Design Patterns came before GRASP; however, the principles and practices were implicit and it assumed that the reader was also aware of them.

In terms of the re-implementation work conducted for this project, various design patterns were used to promote composition and code reuse. This included the variations of the Composite and Builder patterns that will be outlined in detail in Subsection 5.2 (p. 55).

### 2.2.4 Class Inheritance Versus Object Composition

As mentioned, the overuse and misuse of inheritance provided some of the motivation for component-based paradigm in the early years. This incentive was also acknowledged by the GoF in a discussion of the various trade-offs between composition and inheritance (Gamma et al., 1995, pp. 18-20). Similarly to Szyperski (2002), they state that their second principle of OO design is to always “favour object composition over class inheritance” (Gamma et al., 1995, p. 20).

This is due to inheritance being fixed at compile-time, and because it breaks encapsulation because of shared implementations. But conversely, composition can be used to combine elements dynamically at run-time, through well-defined interfaces. As a result, it does not break encapsulation and internal dependencies (p. 19). It also had benefits on system design resulting in fewer classes that are used to instantiate many objects. The interrelationships between those objects then define system behaviour rather than being defined in a single class (p. 19).

Regardless of these benefits, inheritance is used in 21 out of the 23 design patterns. Further, GoF acknowledge that inheritance and composition both complement each other. So as a result of use inheritance, reuse through composition is easier because newer components are able to be composed with older components (p. 20).
2.2.5 Rules For Inheritance

The GoF admitted that despite these benefits, inheritance was overused for the sole purposes of reuse and not to support composition. Further, it is surprising to then find that the GoF used *multiple inheritance* in the Adapter pattern (Gamma et al., 1995, p. 141), which is more recently considered to be an anti-pattern. What this demonstrates is that at the time of writing, good practices were yet to be determined with better hindsight and understanding.

As in the Adapter pattern, multiple inheritance is a form of inheritance that is very commonly used for the sake of reusing code. For example, the Java language only supports single inheritance (Martin, 2003, p. 308) for the purposes of simplicity. However, some controversially argue this is a flaw in Java, as variations of multiple inheritances such as *mixins* are used in other languages such as Scala (Odersky, Spoon, & Venners, 2010, p. 258). However, it is understood that multiple inheritance is not a bad practice as such, but for the fact that it is applied in order to reuse code.

In order to moderate inheritance misunderstanding and misuse, Sargeant (2012) suggested the following guidance for the sensible use of inheritance:

1) A clear ‘is-a-kind’ relationship must exist between the superclass and the subclass.

2) Each of the subclasses must be different from each other and the superclass; otherwise, there is no reason to use inheritance and a single class will suffice instead.

3) Each instantiated object must be a member of only one subclass, so there is no confusion with roles.

4) The subsequent classes must observe the GRASP principles, particularly High Cohesion.

5) Be very cautious with the application of multiple inheritance. Where it is possible, alternatives should be tried first.
2.2.6 Other Forms Of Good Practice

Design Patterns and GRASP patterns help us understand how and when to use forms of code reuse, such as inheritance and composition. Publications by the GoF and Larman have helped to make the kinds of practices used by the very best OO designers from the last 20 years ago available to all developers in the present day. As an example, the Composite design pattern was used to develop the re-implementations of the previous project. These specific examples and more are outlined in Subsection 5.2.2 (GeneralFramework, p. 62).

The effect more generally is a dramatic improvement, and many of the advantages suggested by the component proponents are also available in suitable OO designs. Therefore, some of the problems that components are said to address, arguably no longer apply. For example, code (particularly C/C++) has been written as monolithic chunks, but subsequently much has been learnt about how to write and design OO systems, resulting in software units like classes becoming smaller. Other modern practices include clean code, where code in a programming language is tidied as it is written; thus, making it more readable and understandable to others (Martin, 2009).

The separation of concerns in the many interconnected parts of a program can be managed using another programming paradigm called aspect-oriented programming. These parts are separated, but still remain grouped as connected aspects of the program such as business logic and data persistence. Supportive post-compilation technologies have allowed the interlacing of those concerns “in a manner that is transparent to the developer. These approaches maintain the illusion of separation during development work, and weave in the concern before execution.” (Larman, 2005, p. 555).
CURRENT COMPONENT-BASED SOFTWARE DEVELOPMENT

Introduction

To some extent the way people currently consider components is quite different from what was originally envisaged. In order to understand how this has happened, it is important to define terminologies currently used within the component field. Firstly, this includes an explanation of the idea of a component model and life cycles in the general sense. Secondly, these general contexts and concepts are applied specifically to a component model called X-MAN.

The next two sections are assembled from the taught materials by K.-K. Lau and Taweel (2011). It approximately covers a combination of in-house and other related research of the last decade. This specifically includes the published works, thinking and findings of the Component-based Software Development (CBSD) research group, based at the School Of Computer Science, University Of Manchester, unless otherwise cited.

In order to describe current component tools and concepts, it is necessary to define classifications and formalisations that have resulted from recent studies in the field.

3.1 Component Model

K.-K. Lau and Taweel (2007) define a software component using a component model, which has two necessary parts:

1) Component: that is the conceptually abstracted units of software and conceptual abstractions that are described by:
a. The syntax: defines how the components are exemplified, built and what form they take.

b. The semantics: the intention and design behind the components.

2) Composition: the mechanisms used to compose components, or how the components are connected and assembled together.

The next section gives general examples of component models, showing a progression of software technologies defined using necessary parts of the component model concept.

3.1.1 Components

Rather than Szyperski (1999) trying to stress the distinct separation between objects and components, a survey by K.-K. Lau and Rana (2010) seemed to accept a different view. The survey recognised that there had been various consolidations between objects and components, and an acceptance of the many resulting amalgamations. This is significant because of attempts to classify all the previous advances in the technology, including objects, from the perspective of the components paradigm. This was undertaken using the context and concept of component models.

The survey characterised four abstracted types of components (K.-K. Lau & Rana, 2010), but the two that are of particular interest are as follows:

1) A generic component concept, having two key qualities which all the component types share:
   a. It provides services or operations, usually given through an interface, to compatible software constructs such as other components, clients, objects and environments.
   b. It may also require services, which are operations needed, but not implemented by a component, but instead provided by other software constructs.

2) An encapsulated component only provides services; therefore, it has no external dependencies and only performs all the computation within its scope. There
are very few component models that support encapsulated components, one of which is X-MAN.

3.1.2 Composition Mechanisms

There are various composition mechanisms for component models that have been noted by K.-K. Lau and Rana (2010). Firstly, software composition is defined in terms of software units that are combined together into greater constructs, known as composites. Secondly, composition is said to represent any potential and meaningful interaction concerning units of software, whether they are components or composites. The composition mechanism describes the interaction between those units. Further, the software unit also defines behaviour, and through composition mechanisms, greater units of behaviour can be built.

This description incorporates in the broadest terms the most generic forms of composition mechanisms, including message passing such as direct method calling. Method calls are a conventional means of communication between objects by data passing through parameters (Mehta, Medvidovic, & Phadke, 2000). However, the mechanisms are not very automatable and improved mechanisms for composition units are required in order to support suitable systematic software construction (K.-K. Lau & Rana, 2010).

3.2 A Component Life Cycle

In order to formulate ideas, in a paper called “What characterizes a (software) component?” researchers proposed a set of desirable features. These desiderata stipulate what a component-based development process should include, in order to be more beneficial and meaningful (Broy et al., 1998). From that paper, key component development characteristics were extracted, providing the ability to build, store, assemble, copy, compose, instantiate and most importantly reuse components (shown in Figure 1, p. 30).
Figure 1 The component-based software development desiderata by Broy et al. (1998) formalised into distinct development phases (K.-K. Lau & Taweel, 2011).

Another consequence is the idea that the component life cycle should be distinct from, but also feed into traditional software development processes (shown in Figure 2). The motivation for creating a distinct component lifecycle is that existing life cycles do not promote code reuse, and that systems are often built from scratch. It differs because the components are first built and stored in a repository, ready to be assembled and executed by many differing systems.

Figure 2 An example of how a component development might feed into traditional sequential software development processes (K.-K. Lau & Taweel, 2011).
### 3.2.1 Phases Of An Idealised Component Life Cycle

From this research, other component models were surveyed by K.-K. Lau and Wang (2007) and categorized within the context of the component life cycle phases (Figure 3).

![Figure 3 The Categories other component models (1-4), but only the most idealised (Category 5) uses exogenous composition (K.-K. Lau & Wang, 2007).](image)

The result of this work was a formulation of an idealised component life cycle (Figure 4, p. 32). This embodiment had been the combination of previous component models that were studied (K.-K. Lau & Wang, 2007) as well as the desirable features from leading component conceptualists (Broy et al., 1998).

The process involved asking component experts what they thought components should aspire to, in order that the phases of an idealised life cycle could be identified.
Figure 4 An idealised life cycle, which fully meets the requirements of the design, deployment and run-time phases (K.-K. Lau & Wang, 2007).

Figure 4 shows what is accepted as being the phases of a component life cycle (K.-K. Lau, Elizondo, & Wang, 2005), which as an idealized component life cycle allows:

1) A component designer in the design phase to build components that are domain specific but not system specific, and then store those components in a repository.

2) A component designer in the design phase is then able to retrieve into the builder tool the stored components from the repository, compose them together and place them back as composites.

3) A system developer, in the deployment phase, is able to retrieve those components as subsystems and assemble them, in order to construct a complete system.

4) The assembled subsystem components are executed in the run-time phase as a system in a run-time environment.

With this understanding, K.-K. Lau and Taweel (2011) and the Manchester CBSD research team formed a new component model, known as X-MAN.
3.3 X-MAN

X-MAN is a conceptual component model that currently comprises of two known versions of the development tool, both representing the results of research conducted over the past seven years. This includes studies surveying various component models and life cycles, and inclusion of desirable features described in the previous section. However, it is an on-going research concern that is still under development. X-MAN can be described as a pure component model. Through their distinctive properties, the two tools both regulate and restrict component software development process in order to fulfil the concept of an idealised component life. A consequence of this is that the model has purportedly solved many issues concerning software development, but specifically the matter of greater component reuse (Elizondo, n.d.).

3.3.1 Component Model

This section seeks to explain the syntax, semantics and composition of the X-MAN component model.

3.3.2 Composition Connectors

K.-K. Lau and Elizondo (2010, p. 1168) define a composition connector as being “nary and used to support component composition”. There are three types of catalogued connectors that they discussed, but in order to understand the uniqueness of the third, it is necessary to explain why the first two traditional connectors are inadequate.

![Diagram of composition connectors](image)

**Figure 5** Direct message-passing using method calls (K.-K. Lau et al., 2005, p. 2).
According to K.-K. Lau et al. (2005) components are traditionally connected by message passing, either by:

1) **Direct** method calls: A form of object delegation (K.-K. Lau & Rana, 2010, p. 3) represented by implicit connectors, where objects invoke any number of methods of other objects shown in Figure 5 (p. 33).

2) **Indirect** method calls: Represented explicitly by connectors such as intermediary adapter objects or the in/out ports of architectural units. These connectors are linked with components in order to compose them (Figure 6).

Both Figure 5 (p. 33) and Figure 6 show the forms of message passing; however, as K.-K. Lau and Elizondo (2010, p. 1166) explain, direct message passing still involves too much control coupling:

> “When components are connected by direct message passing any data flow and control flow related to the ‘composition’ is mixed with the computation preformed in the individual components. Thus, there is no explicit code for connectors, since messages are ‘hard-wired’ into the components; it makes the sender and receiver components tightly coupled with one another. Similarly, there is no explicit code for the resulting ‘composition’.” K.-K. Lau and Elizondo (2010, p. 1166).

Indirect methods passing are an improvement, but are still too coupled in terms of control as components can still call the connectors internally using method calls (Figure 6).
The third type of composition connector, and a central concept in the X-MAN component model is known as exogenous connectors. As shown by Figure 7, this connector differs from the indirect connectors because the method invocation and message passing only occurs from inside the connectors and not from the components. As a result, exogenous connectors are able to solve the issue of control coupling in traditional connectors (K.-K. Lau et al., 2005, p. 91), because coupling of control is said to exist externally inside the connectors. This is due to three extrinsic properties: (1) composing components externally, otherwise known as exogenous composition; (2) communicating via peripheral data channels; and, (3) controlling the flow of the data and orchestrating the sequence of invocations between the connected components (shown in Figure 8).

The direct consequence of the use of exogenous connectors, as shown by Figure 9 (p. 36), is that the X-MAN component model is able to meet the requirements of an idealised life cycle (K.-K. Lau & Taweel, 2011).

**Figure 7** The message passing of exogenous connectors (K.-K. Lau et al., 2005, p. 5).

**Figure 8** The control and data flow resulting from the use of exogenous composition connectors (K.-K. Lau & Elizondo, 2010, p. 1166).
Figure 9 Various component models and how they are able to support the key characteristics of an idealised component lifecycle (K.-K. Lau & Wang, 2007).

3.3.3 Encapsulated Components And Connectors

The components and connectors of the X-MAN component model are unlike other models, because of the elements they are meant to encapsulate. It is generally agreed that objects encapsulate state (data) and behaviour (methods and functions) in an instance (Szyperski, 2002, p. 22). But more generally it is possible to encapsulate three elements: data (flow), control and computation (K.-K. Lau & Wang, 2007, p. 5).

Figure 10 The two encapsulated (a) atomic, (c) composite components and (b) composition connector (K.-K. Lau & Rana, 2010, p. 6).

Referring to Figure 10, the X-MAN component model has two types of encapsulated component and an abstracted composition connector (K.-K. Lau & Taweel, 2007):
1) An atomic component: the smallest unit of composition in the X-MAN model. In order to allow composition, it has two necessary features:
   a. A unit of computation (denoted U), which can be a piece of code that is self-contained such as a class, module or function.
   b. An invocation unit (denoted IU), which is a point of connection and provides access to the computation unit.

Combined, they both encapsulate only computation and data, and are said to have no external dependencies (K.-K. Lau & Taweel, 2011).

2) An abstracted composition connector, shown by Figure 11 (p. 38), defines and co-ordinates control for at least two atomic and/or composite components. Composition connectors encapsulate control or ‘invocation techniques’ (Mehta et al., 2000). They also support the transfer of control among other connectors, and according to K.-K. Lau and Elizondo (2010), there are three specific behavioural types:
   a. The pipe connector simply invokes composed components successively. However, data that is passed to the pipe connector is passed to each component that it composes, but data that is returned to the pipe is also passed between each of the composed components. The output of the first component becomes the input of the second.
   b. As with the pipe, the sequencer connector invokes connected components successively. Data that are passed to the sequencer are also passed on to these components. However, it does not explicitly pass any returned data to the next subsequent composed component.
   c. A selector connector: a result of an evaluated conditional expression that will determine which connected component will be invoked and computation calculated. This is equivalent to condition statements such as if-then or switch statements.

3) A composite component is a combination of two or more atomic or composite components, assembled together using a composition connector. It then
provides an interface, which can be further composed with additional atomic or composite components.

![Diagram](image.png)

**Figure 11** The three basic composition connectors and their behaviours: (a) sequencer, (b) pipe and (c) selector (K.-K. Lau & Elizondo, 2010).

Encapsulated components are suitable to store in a repository because they have no extrinsic dependencies and are thus supportive of an idealised life cycle. Once stored in the repository, encapsulated components are able to be built and assembled in the design and deployment phases respectively (K.-K. Lau & Taweel, 2011).

### 3.4 Further Explanations And Observations

#### 3.4.1 Versions Of X-MAN

There are two versions of the X-MAN tool that are based on the X-MAN component model, which are:

1) The *Java-based version* (shown in **Figure 12**, p. 39), which specifically uses reflection from the java.lang package to “perform structural introspection on components’ binaries and their interfaces to retrieve the information about the services that the components offer” (Elizondo, 2008 ~ 2009). It also supports a much wider range of connector types (K.-K. Lau & Elizondo, 2010), including loops.
Figure 12 The Java-based version developed by Elizondo (2008 ~ 2009).

2) The C-based version (shown in Figure 13, p. 40), which uses the Ch (C/C++ interpreter) external library, the embedded Ch toolkit and the GME (Generic Modelling Environment) framework (MBCSDRG, 2011). GME is a configurable toolkit that through meta-models allows the creation of graphical representations within specific domains (GME, 2008). Through this, the C-bases tool composes systems together by dragging and dropping deployed components and linking them together via connectors.

Nalmpantis (2011) used the C-based tool for the avionics system project and it is also used in the tutorials and taught materials of the CBSD course (K.-K. Lau & Taweel, 2011).
Figure 13 The C-based version of the X-MAN tool, showing the designer view (MBCSDRG, 2011).

It is believed that the more restricted C-based tool is more widely publicised because of its intended use for embedded systems. This might have been strengthened due to funding considerations (K.-K. Lau, 2009), in particular recognition from the CESAR (‘Cost-Efficient Methods and Processes for Safety Relevant Embedded Systems’), a European funded project (CESAR, 2009).

As discussed with Dr. John Sargeant (2012, pers. comm., 26 July), embedded systems have restrictive factors, such as only being able to run limited instruction sets and being able to compute in real-time. A critical property of a real time system is not the speed of execution, but that it is able to execute instructions predictably in a discrete amount of time. A possible explanation as to why the Java-based version might not have taken prominence could be in part due to the garbage collector of Java interfering with the consistency of computation time.

3.4.2 Looping

Although Figure 13 shows a loop connector, none of the materials describe the C-based version of X-MAN mention basic loop control flow constructs at that time. This includes the project implementations by Kang (2010), Nalmpantitis (2011) and Popoola (2011). There is uncertainty as to why the
construct was not applied using that version of the X-MAN tool, because K.-K. Lau and Elizondo (2010) had previously catalogued the structure as an adaptation connector and made use of it in the Java-based version of X-MAN. Loop structures such as while (conditional) and for (iterator) statements are required to execute repeatedly, and therefore reuse the same section of code. It is difficult to understand why such an elementary control flow structure was not used at the time of writing. Simply, looping is fundamental to programming and the absence of provision for the construct will result in a significant impact on the reuse of code. Also, it is not possible to use recursion instead because of the use of exogenous connectors.

3.4.3 Structure

The use of the exogenous composition connectors and atomic components, combined in composites, results in the forming of tree structures; therefore, systems are highly factored and modular (K.-K. Lau & Taweel, 2007, p. 5). Figure 14 shows a simple banking system and demonstrates the structure.

![Figure 14](image_url)

**Figure 14** Through the use of exogenous composition connectors, the X-MAN component model produces a tree structure (K.-K. Lau et al., 2005).

As shown in the abstract representation of a banking system, the connectors include pipes (P1, P2 and P3) and sequencers (S1, S2 and S3). Composites are represented as two bank consortia (BC1 and BC2), atomic four bank branches (B1, B2, B3 and B4) and one cash machine (ATM). What is not shown in the diagram is the internal tree structures contained inside the bank consortia.
composites. For a better representation of this, please refer to Figure 28 in Subsection 5.2.2.7 (The Application Of The Generic Framework, p. 68). As Figure 28 shows and Figure 14 (p. 41) does not, composites can only be composites as long as they have internal structures made up of atomic components and connectors. Because of their modular nature, composites can be copied and reused multiple times (K.-K. Lau & Taweel, 2007, p. 5).

Further, it is claimed that component encapsulation is retained throughout the tree structure, a unique characteristic of the model (p. 5) and something that traditional object structures do not support. However, the C-based version of the X-MAN tool only supports tree structures and does not support graph structures. Graphs are used to model naturally occurring, complex, interconnected systems such as roads, pipes or computer networks. This would appear to limit the general applicability of the tool.

3.4.4 Coupling

When studying the C-based version, there was some confusion about the apparent data dependencies contained in each atomic component.

An X-MAN atomic component is essentially a C function (Table 1, p. 43) that includes: (1) a function definition with a return type, function name and a list of typed arguments delimited by brackets and divided by commas, and (2) the body of the function included by two curly braces that has local variables and some form of computation. The atomic component has the return type void, which means it does not return a value. Instead, in order to pass data back to the caller, each component accepts a reference parameter as an argument. As is shown in Table 1 (p. 43), this references an address of a variable using the unary & operator.

As a consequence, there is seen to be a direct, extrinsic and implicit dependency, making it apparent that the atomic component is not as self-contained and decoupled, as had previously been portrayed.
Table 1 An example of atomic component from the Nalmpantis (2011) C-based implementation.

The code segment above (Table 1) is an atomic component of an antilock braking system, implemented in the C-based version of the X-MAN tool by Nalmpantis (2011). However, because of the use of the reference data type, the component does mutate a value of a variable that is outside the scope of the component. Further, due to use of references, there is effectively no semantic difference between the pipe and sequencer connectors. However, there is nothing in the literature or illustrations (Figure 10, p. 36) from the published research or taught material by K.-K. Lau and Taweel (2011) that explicitly declares this.

As mentioned previously, K.-K. Lau and Elizondo (2010, p. 3) state there is no coupling between components that communicate via co-ordination connectors such as composition connectors of the X-MAN component model; however, without some forms of coupling and dependencies, it follows that a system would simply not function. The coupling of the C-based version is contained in the implicit dependency of the reference parameter; thus, it is more correct to say that there is no direct coupling.

3.4.5 Contracts

Dr. John Sargeant recounted a discussion with Dr. Kung-Kiu Lau (2012, pers. comm., 26 July), in which he discovered that there are two separate explanations for ways to handle the data dependencies discussed in the previous subsection:

1) It is implicitly assumed that data flows through the component from left to right and, where appropriate, upwards (Figure 8, p. 35). Ultimately it does not matter what fundamental mechanism is used for passing the data.
2) Expected behaviour is specified via contracts, where a component contract “forms the contractual expectations of both parties to the component development process and final product, sometimes termed ‘design by contract’” (Councill & Heineman, 2001, pp. 28-29).

Meyer (1997) popularized ‘Design By Contract’ (DBC) programming. The practice is analogous to legal contracts, which stipulate that whatever entity is being dealt with is able to provide the services that are required, given the terms that both parties agree upon. Specifically to components, it can be thought of as a contract between the user of a component and builder of the component. DBC was formed around the central idea of formal specification and operational contracts at a low level of detail, including the use of invariants (things that do not change), pre- and post-conditions (Larman, 2005, p. 194).

The composition of components using a connector is governed by a clear set of logic-based rules, which cleanly combine both the pre- and post-conditions. The X-MAN component model is defined and restricted in such a way that the formal semantics of the individual components and their composites are well defined, which is another desirable property of safety-critical embedded systems.

It is possible through the use of Javadoc’s (a document generator API) annotations (Hirondelle, 2010). Recent versions of Java have made it possible to use process-able wide tools that could match up contracts using “@precondition” annotation that could formally specify prerequisites. There are many other such tools that allow the support of DBC in Java (Wen, 2013).
Chapter 4

**Braking Systems**

**Introduction**

In a project called *Component-based Software For The Avionics Domain*, Nalmpantis (2011) implemented an anti-lock braking system (ABS) for aircraft as a proof of concept. The project used the C-based tool and adds weight to the argument that X-MAN component life cycles, models and technologies are more able to support the idea of reuse and safety-critical embedded systems. In order to explain how he went about this, it is first important to explain the context of braking systems in general.

**4.1 Braking**

The following sections are derived from a book called *Automotive Braking Systems* by Halderman (1996).

In terms of safety, brakes are the most vital mechanism for any vehicle (p. 17). Without braking systems, it would take a long time for a vehicle at significant speed to come to a halt. The result would be an increased risk of impacts with other vehicles or structures.

Their purpose is to reduce or stop the movement of the vehicle by: (1) reducing the revolution of the tyre and wheel assemblage; and, (2) translating that rotation reduction into traction generated between the tyre and ground surfaces (p. 17). The deceleration of a braking system depends on many variables, such as the size of the structure, the mass of a vehicle, the weight transfer in the direction of travel and the down-force thrust created aerodynamically (pp. 18-19).
4.1.1 Coefficient Of Friction

Simply stated, braking systems absorb energy by the conversion of a vehicle’s velocity into the creation of heat, during the decrease in wheel turn speed (p. 17). The force that generates the heat is friction, or resistance of the “motion between two objects that are in contact” (p. 21). When braking, friction is generated due to hydraulic pressure forcing the brake pad and disc together (p. 23). But the vehicle’s speed is ultimately reduced due to interaction concerning the tyre and ground surface because of the vehicle’s weight and down-force.

The extent of friction along these contact surfaces is called the coefficient of friction, which is a factor between zero and one, and denoted $\mu$ (p. 17). The coefficient depends on ‘material, temperature and surface finish’ of the constituent parts (p. 21). Complexity is added as the surfaces vary in traction, particularly between the tyre and ground surface. Changes in traction are called split-$\mu$ conditions (p. 22), such as the transition between snow and grit. Specialist automobile braking systems, such as antilock braking systems, have a significant advantage in such braking environments.

4.1.2 Antilock Braking Systems

An antilock braking system (ABS) prevents wheel locking throughout the braking periods in low-traction environments. This is due the adaptation of the degree of braking pressure according to differences in traction in each wheel (pp. 22-24). Through the measurements of the wheel’s speed via sensors, a controller is able to gauge whether a wheel is slower than the rest, which is suggestive of a skid state. An electronic control unit is then able to vary the breaking pressure accordingly, pulsing the brakes on and off in microseconds.

Although an ABS system will not reduce velocity as efficiently on every surface, an ABS provides a greater continuous interaction for a wheel than in a skid state. This is because a locked wheel has less traction and greater tyre
degradation due to heat; therefore combined, it has less control than a revolving wheel (p. 24).

4.2 Avionics ABS

Nalmpantis (2011) developed his ABS project in the C-based version of the X-MAN tool. An aircraft uses the ABS during the landing phase of a journey.

The concepts behind the original Nalmpantis (2011) implementation is discussed next. This includes the parts of an ABS system, the PID (Proportional-Integral-Derivative) controller and the fine-tuning the variables used in the PID controller.

4.2.1 System Analysis

Nalmpantis (2011) focussed on the problem of avionics braking systems, which control the slowing of high-speed aircraft by reducing the speed of the tyres. Nalmpantis (2011) developed an ABS system in the avionics domain, which protects against occurrences of locked wheels (deep skid condition), extreme surface conditions (such as hydroplaning) and accidental early braking during landing (pp. 29-30).

Nalmpantis (2011) defines avionics as a “blend of aviation and electronics.” This combination of the digital and analogue has given many benefits to aviation, including: reduced weight, faster and accurate response and handling, and better efficiencies and maintenance. However, these benefits also have a trade-off, which mainly revolves around the complexity of managing and modelling those problems in software and hardware (p. 17).

A real world ABS is a mixture of hardware and software that, when combined, fulfil the requirements of braking in aircraft. At the centre of this system is the control unit, shown in Figure 15 (p. 48), which after landing, continually accepts measured inputs from sensors and other interacting parts, and produces outputs in response until the braking phase is complete. The sensors’ measurements include: (1) vehicle and wheel speed, (2) wheel contact
with the ground, (3) manual override and brake application, and (4) anti-skid control (Nalmpantis, 2011, pp. 27-29).

Figure 15 An abstract representation of an anti-skid braking system, including the various interacting parts (adapted from Ludovic, n.d.).

4.2.2 Control Loop Feedback

Figure 16 (p. 49) shows a part of the controller unit called the PID (Proportional-Integral-Derivative) and additional sensors that together are used finely adjust of wheel speed through braking. Sensor measurements of wheel speed are fed as input values into the loop where: (1) an error is calculated by finding the difference between actual wheel velocity and the expected reference velocity; (2) this Proportional adjustment ($K_p \times e$) value is then fed into 2 functions which calculates the Integral adjustment ($\int K_i \times e \, dt$) and the Differential adjustment ($K_d \times \frac{de}{dt}$); and, (3) the outputs are then summated (Warwick, 1989, pp. 322-323). The result is used to control a valve, which determines the amount of breaking pressure and adjusts the current
wheel speed. The output is also is then fed back into the loop as the current wheel speed.

![PID Controller Diagram](image)

**Figure 16** The control loop feedback of the Nalmpantis’ PID controller (adapted from Nalmpantis, 2011, p. 31).

Nalmpantis (2011, p. 30) describes the PID as a “generic control loop feedback mechanism”. As shown by **Figure 16**, it continually corrects the output, based upon the inputs of the value of adjusted speed of the wheel and the reference speed as a percentage of aircraft speed.

### 4.2.3 Tuning

The values of variables $K_p$, $K_i$ and $K_d$ are the result of a process called tuning. Nalmpantis (2011) describes this process as tailoring the output, according to the requirements for an application of a given domain, of each P, I and D function. The strength or weakness of the calculation of a particular function (P, I or D) is weighted by $K_p$, $K_i$ and $K_d$ and represents specific characteristics.
of braking (for example braking correction, performance and compensation), which are determined according the value of these constant variables. In the context of the original C-based implementation, the various characteristics specified in the tuning process have resulted in the values of $K_p = 0.29$, $K_i = 0.001$ and $K_d = 0.31$. These values were reused again within the present re-implementation work.
Chapter 5

IMPLEMENTATION

Introduction

Next, the Nalmpantis (2011) system design of the C-based X-MAN implementation will be briefly described. An attempt will be made to explain the thinking and design modifications as a consequence of using the X-MAN, as well as some of the implications of these decisions.

5.1 Design

Figure 17 The three subsystems that makes up the previous anti-lock braking system (adapted from Nalmpantis, 2011, p. 58).

To break down the problem further and to identify the components of the system, Nalmpantis (2011, pp. 48-50) used state charts, which describe the behaviour of a system. These are used to map the requirements to a design of an X-MAN implementation. From this process, Nalmpantis (2011) identified three subsystems: simulation, functional and interface.

Referring to Figure 17 above, which shows these subsystems and the overall ABS structure:

1) The *simulation* part of the system, in which values of interacting parts are mocked up and fed into the second subsystem and results are returned to be used to determine the final output.
2) The *functional* core part of the system that contains the PID that adjusts the degree of braking that needs to be applied.

3) The *interface* subsystem that is responsible for interaction between the system and the environment; specifically the exchange of information from sensors and the software.

**Figure 17** (p. 51) also shows the ideal formation using Pipe and Sequencer connectors that join the constituent parts of the Nalmpantis’ implementation, including: (a) the sensor components of the interface, (b) the control unit components, specifically the PID (Proportional-Integral-Derivative) of the functional components, and (c) the wheel associated with the simulation components. These components will be detailed later; however, it is necessary to describe why the final Nalmpantis (2011) implementation design differs from the one in **Figure 17** (p. 51). This is due to the particular way the Pipe connector passes data. This results in what Nalmpantis (2011) described as “redundant data” being passed to components that do not require it.

### 5.1.1 Redundant Data

**Figure 18** An example of redundant data being passed from Sensors 1, 2 and 3 to Components 1 and 2 (adapted from Nalmpantis, 2011, p. 59).
Figure 18 (p. 52) shows the data being passed between two subsystems via Sequencer and Pipe connectors. Firstly, the Sequencer invokes three sensor (Sensor1, 2 and 3) components. As it does this, the output of the Sensor1 is not passed to Sensor2, and again with Sensor2 to Sensor3. After all the outputs have been collected, they are then returned from the Sequencer back to the Pipe. The Pipe connector then invokes Component1 and Component2 consecutively. However, the Pipe connector differs from the Sequencer because it passes the output of Component1 to Component2.

Nalmpantis (2011, pp. 58-61) described this as problematic. This is because the interfaces of Component1 and Component2 have to change to support this method of data transfer. As a result, data is passed by reference to components that have no requirements or use for that data. When extrapolating this to larger systems, the interfaces of the components would need to grow linearly to reflect the linear increase in the amount of data being passed by the Pipe connector; therefore, there would be an element of ineffectiveness and inelegance in this approach.

Figure 19 Data is passed directly to the component as and when it is required, thus reducing data redundancy (adapted from Nalmpantis, 2011, p. 60).

Nalmpantis’ solution to the problem was only to invoke and retrieve the output of two sensors separately. The sensor data is passed to a component only when it is required at that specific point in the flow of the system. Figure 19 shows no redundant data being passed, resulting in tighter component
interfaces. However, the consequence of this approach is a much deeper tree structure. The addition of one more Pipe and two more Sequencers consequently increases the amount of connections. As a result Sensor2 is executed twice, if differing components require the same data but at different points in the program.

Extrapolating this to larger systems, there is a cost to pay in the increased amount of complexity in the structures of these systems.

5.1.2 Final Implementation Structure

**Figure 20** shows the final arrangement of the C-based ABS system implementation, including all the system connectors and components. The system contains two independent wheels whose deceleration is dependent upon the retrieved interface sensor data and functional control unit calculations. These are accessed and executed twice in a braking cycle, once for each wheel. Due to the reduction in redundant data, the sensors are accessed 22 times for each wheel, with a total of 44 times across both wheels.

**Figure 20** The final C-based X-MAN implementation of an ABS system by Nalmpantis (2011, p. 69).

The specific constituent parts, shown in **Figure 20** as the sensors, PID and wheel components, will be discussed in greater detail in the next subsection,
along with a framework representing the X-MAN parts that make up the current Java X-MAN-like re-implementation.

5.2 Overview Of The Java Re-Implementations

The work of the current re-implementation falls into three parts, including:

1) An X-MAN-like version that is similar to the original in structure to the final Nalmpantis (2011) C-based implementation, shown in Figure 20 (p. 54).

2) A reduced X-MAN-like version that is closer to the original design (Figure 17, p. 51) and takes into consideration the accepted use of redundant data.

3) A separate object-oriented implementation used for the purposes of the evaluation that has the results that are reproducible from the first two.

Firstly, in order to discuss the first two X-MAN-like versions, the concrete re-implementation falls into a number of separate sections:

1) The method of passing data between the components and various trade offs, weaknesses and justifications for the approach that was taken.

2) A general framework for representing the X-MAN component model. This is based on a variation of the Composite design pattern and required for the current re-implementation. It also includes the specific exogenous connectors for composition and utilities.

3) The re-implementation of three subsystems of the original Nalmpantis (2011), specifically including the following components: (1) the sensors of the interface subsystem, (2) the PID (Proportional-Integral-Differential) component of the functional system, and (3) the wheel of the simulation subsystem.

4) Finally, the builder pattern that was used to create the object formations that are used at run-time, is briefly mentioned.
5.2.1 Data Passing Mechanism

One particular problem was passing data between the components via the composition connectors. The question was asked whether it was preferable to try to emulate the X-MAN’s means of passing data using references, as described in Subsection 3.4.4 (Coupling, p. 42). This differs in Java, because the uses of references are implicit, and only applied to objects opposed to primitive types. All primitive types are passed by value, and the parameters copied for safety reasons. The traditional object-oriented method of passing a value back from a function is to use return statements. This yields an address that contains the value, back to the caller of the function. However, the original Nalmpantis (2011) implementation did not apply return statements.

Four options for overcoming this problem became apparent:

1) The simplest way is to pass the value as a float array with a single element. The result is that the value in the single element is mutable by a function that assigns a new value. A disadvantage is that it increases the complexity of the code and the use of return statements is preferable (Reilly, 2006).

2) Another option would be to design an object that would “wrap” some functionality around a primitive, corresponding to the Decorator pattern (Gamma et al., 1995, p. 175). This allows the value to be mutated and passed by reference contained inside an object. Decorated primitive types, such as Integer or Float classes in the java.lang.Number package, do exist. However, for reasons not fully understood, the values that are wrapped are immutable in a fashion similar to their primitive equivalents. Thus, the wrapped primitive’s value cannot change as it is passed around as a parameter between functions.

3) Apache provide mutable wrapped primitives from the org.apache.commons.lang.mutable package, such as MutableFloat. Such decorated primitives are able to change because of the implementation of the Mutable interface (Apache, 2011).
4) The final option makes use of another data structure organisation that encapsulates all the data being passed, that is mutable and can be passed by reference. This takes the form of the ArrayList class from java.util package. The consequence of this approach is that both required and redundant data will be passed to all the implementations. This is because all the data used by the system will be contained in the ArrayList.

The first iteration of this implementation applied the first option. This is because it reflected the data passing of the original C-based implementation and afforded a better understanding of the original constraints. This has changed in future iterations of the current re-implementation, as a clearer solution became apparent.

The complexity of producing specific implementations of each connector, in order to reflect the differing interfaces of each component and composition connector meant that the first three options were discounted. Using these approaches meant that every composition connector had to be specific to the interfaces of the components, requiring a new connector for every set of components. This also impacted on reuse, because every time a component was added or removed from a composition connector, those changes had to be reflected in the method calls.

It was considered more important to make the composition connectors as generic as possible. This was done to represent as many differing domains as possible, in order to promote reuse and flexibility. Therefore, the fourth option, where the data contained inside an ArrayList that is passed between components was found to be a more appropriate approach.

However, it became apparent that the retrieval of values from the ArrayList through an index was becoming programmatically problematic. It was difficult to distinguish one value from another through the use of the index. Therefore, it was necessary to make sure that the right value was provided to an atomic component. Nalmpantis (2011, pp. 78-79) also recognised this as an issue his discussion about the limitations of the X-MAN tool. For the current re-implementation, this was resolved through the provision of keys that more
intuitively differentiated the values, all of which were contained in an extended version of the ArrayList, described next.

5.2.1.1 ExtendedArrayList

A modified version of ArrayList called an ExtendedArrayList was selected to solve this issue. Figure 21 shows the ExtendedArrayList inheriting from the ArrayList collection class from java.util. The ExtendedArrayList also provided some extra functionality to the ArrayList data structure, specific to the current re-implementation.

![Figure 21: The class diagram ExtendedArrayList extending ArrayList containing OrderedPair elements.](image)

The elements in the ExtendedArrayList are of the type OrderedPair class that implements the Pair interface, both derived from a tutorial (Oracle, 1995 ~ 2013). The class and interface were used to distinguish one element’s value from another inside the ExtendedArrayList class, by distinction of a unique key. The key and value is stored in OrderedPair objects. A particular value is retrieved by providing a key through the getValue interface and matching it with the maintained list of keys, and then returning it. New values could be added by providing a key and value through the addPair interface.

The ExtendedArrayList also makes use of two parameterized types: K for the type keys and V for the mapped values. Also known as generics, K and V allow the writing of a generic class that can be parameterized using any type (Poo, Kiong, & Ashok, 2007), and allows the types to vary. It is later specified when the class is instantiated. This offers greater reusability and flexibility of
a class. However, for the purposes of the current implementation, the
ExtendedArrayList is parameterized with a String for K and a Float for V. This
is consistent with all the other generic classes of the current X-MAN-like re-
implementations.

5.2.1.2 The Use Of The ExtendedArrayList

```java
public void invoke(ExtendedArrayList<K,V> valueList)
{
    Float value1 = (Float) valueList.getValue((K) "key1");
    Float value2 = (Float) valueList.getValue((K) "key2");
    Float value3 = value1 - value2;
    valueList.addPair((K) "key3", (V) value3);
}
```

Table 2 An example of the use of ExtendedArrayList.

Table 2 shows a typical invoke method, which shows how an instantiated
ExtendedArrayList object is used by atomic components:

1) An ExtendedArrayList is passed as a parameter to the invoke method, and
   allocated with the argument name valueList.

2) The getValue interface of valueList is provided the Strings “key1” and
   “key2” and cast into the parameterized type K. The method then iterates
   through the list of OrderPairs and performs String comparisons to check if the
   keys exist. If they do, the values are returned. If they are not found, the
   valueList returns null.

3) The returned values are then casted into Float type and assigned to the local
   variables value1 and value2.

4) The values are then used in a calculation and assigned to the local Float
   variable3.
5) The value of the local variable value3 is then casted to the parameterized type V and passed to the addPair interface, along with key “key3” to be stored as a new OrderedPair to be retrieved, removed or changed by subsequent components.

The two X-MAN Java re-implementations make use of only a single ExtendedArrayList that has all the values used by the system contained within it. It is passed to all the atomic components via the connectors, with the omission of components composed by the Sequencer connector. This exception is discussed later, but what effectively is being passed around is a reference to a single, system-wide ExtendedArrayList. There are valid criticisms regarding the use of the ExtendedArrayList that are acknowledged in the next subsection.

5.2.1.3 Criticisms

It can be argued that the ExtendedArrayList is replicating the same functionality as classes derived from the Map interface, such a Dictionary or HashMap. However, the advantage of using the ArrayList is the availability of an ordered index, which is used by the ExtendedArrayList in order to provide certain functionality. In Table 9 (Appendix B, p. 112) it is possible to see the outputs written to console from the Message class. This includes adding elements to the ExtendedArrayList and printing all the keys and values once the simulation is complete. To do this, the use of indices provided by the ArrayList class is required. On reflection, the same functionality provided by ExtendedArrayList could have been achieved by using a LinkedHashMap instead. Further, this approach would be much more efficient.

The use of the ExtendedArrayList resolves the problem of having very complex interfaces and specific connectors to the components that the connector composes. It also solves the problem of redundant data to a certain extent. This is because the ExtendedArrayList is passed as a reference and the data is not being copied by value. Therefore, the data exists in two places, both inside the ExtendedArrayList and inside the atomic components. However, there are consequences to consider that are highlighted next.
5.2.1.4 Complexity

The use of the system-wide ExtendedArrayList might be considered un-modular and therefore, less reusable because all the values used in the system are contained within it.

A consequence of this is that in very large systems, the amount of data contained in the system-wide ExtendedArrayList will dramatically increase. In this situation, some form of management and removal of elements would be required so that the list does not become too large and impact on performance.

There is also an indirect coupling between all the components because they all depend upon it for data that they require. Because all the components have access to data being passed through the system, another possible effect could be unexpected modifications to the values. This could impact upon the accuracy of calculated results and a great deal of care is required to make sure that the data is not improperly altered.

A way to resolve these problems is for each composite to instantiate its own specific ExtendedArrayList. Then the values that are passed are more localised and can be discarded when they are no longer required, resulting in a more modularized system. However, these require the instantiation of many ExtendedArrayList objects. Because each instantiation would be a relatively expensive operation and there would be an increase in system complexity, it was felt that using the single, system-wide ExtendedArrayList is sufficient for the purposes of the current re-implementation.

5.2.1.5 Performance

The use of the ExtendedArrayList has an impact on performance not only because it has both an index and a key to distinguish values, but also because it contains all the system’s data. In a worst-case situation, all 21 elements are traversed and compared in order to retrieve a single value. Using a HashMap would negate this traversal because values are found and retrieved with a relatively inexpensive look-up of a given key. However, the reason for the use of the ExtendedArrayList is to make understanding and debugging easier.
Despite this, there are few other tangible benefits and simply the ExtendedArrayList is an inefficient data passing mechanism. Although performance is not a major issue, significant optimizations could be achieved with the use of a HashMap.

The ExtendedArrayList is a consequence of the development of the system, because an ArrayList was originally used. With the additions of OrderedPair elements, it can now be effectively thought of as a Map. It was decided that using the ExtendedArrayList would be easier in terms of coding, debugging and flexible composition with the common interface invoke, that will be established next.

5.2.2 GeneralFramework

Next, it was necessary to consider the design of the generic representations of the various aspects of the X-MAN component model, for a X-MAN-like re-implementation in Java. This includes the encapsulated atomic and composite components that use the ExtendedArrayList and exogenous connectors that pass the ExtendedArrayList objects. Together they form the core parts of X-MAN component model.

5.2.2.1 A Variation Of The Composite Pattern

![Diagram of the general framework](image)

**Figure 22** An overall class diagram of the general framework that is a representation of the core parts of the X-MAN component model.

**Figure 22** shows an object-oriented design of an inheritance hierarchy, which is the abstracted representation of all the X-MAN types described in Subsection 3.3.3 (Encapsulated Components And Connectors, p. 36) and
illustrated in \textbf{Figure 10} (p. 36). This arrangement is a variation of the Composite pattern (Gamma et al., 1995, p. 163) and in order to explain it, four subsections relating to \textbf{Figure 22} (p. 62) will be discussed separately.

5.2.2.2 XMANPart

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{GeneralFramework} \\
\hline
\textbf{(A) XMANPart \textless K,V\textgreater} \\
\hline
\textbf{#print : Message} \\
\hline
\textbf{(A) #checkValueList(valueList : ExtendedArrayList \textless K, V\textgreater) : Boolean} \\
\hline
\textbf{(A) +invoke(valueList : ExtendedArrayList \textless K, V\textgreater) : void} \\
\hline
\textbf{#printInvokedPartName(message : String) : void} \\
\hline
\end{tabular}
\end{center}

\textbf{Figure 23} The XMANPart class diagram.

The abstract XMANPart class, shown in \textbf{Figure 23}, represents all the X-MAN types, and it is from which all classes are derived in the two X-MAN-like re-implementations. It specifies two abstract methods: \texttt{invoke} and \texttt{checkValueList}, that have to be implemented by derived subclasses.

The \texttt{invoke} method is where the specific implementation of each component exists, for example the unit of computation in the atomic components and data passing functionality in the composition connectors. The \texttt{checkValueList} is a method implemented by the atomic components and used to check the ExtendedArrayList. This is motivation for having a very basic form run-time checking.

It also provides a \texttt{print} instance variable that is instantiated from the \texttt{Message} class to all derived subclasses for printing messages to the console. The \texttt{printInvokedPartName} makes use of the \texttt{Message} class, to print the name of the invoked component. Please see Appendix B (p. 111) and Appendix C (p. 113) for examples of this.
5.2.2.3 AtomicComponent

The AtomicComponent class, shown in Figure 24, is a general representation of all X-MAN atomic components, from which specific atomic component type subclasses are derived. These subclasses, illustrated as AtomicComponent1 ... AtomicComponentN, contain the specific implementations of the inherited invoke method will be provided. In these cases, the invoke interface is analogous with the invocation unit, and the concrete implementation of that method is similar to the unit of computation of the X-MAN atomic component described in Subsection 3.3.3 (Encapsulated Components And Connectors, p. 36).

The set of keys that is required by each atomic component is passed to the AtomicComponent as an ArrayList object through the constructor. It is then stored in the requiredKeys instance variable, to be later compared to the valueList when a composition connector calls the checkValueList method. If all the values are present, the Boolean value true is returned. If not, an error message is displayed and the Boolean value false is returned. This
checking is done to make sure that the atomic component is provided with the data that it requires so that it is able to operate as expected.

5.2.2.4 CompositionConnector

The CompositionConnector class characterizes all exogenous composition connectors. All the distinct composition connectors originate from it, denoted as CompositionConnector1 … CompositionConnectorN in Figure 25. It is a kind of XMANPart, but also importantly, it contains a collection of XMANPart objects.

![Diagram of CompositionConnector class]

**Figure 25** The CompositionConnector class diagram.

There is a special kind of bidirectional relationship between the CompositionConnector class and the XMANPart class. This association, denoted *composes*, allows CompositionConnector to both inherit from, and compose together, all classes that derived from XMANPart superclass, including other instances of the CompositionConnector class. This is the core application of the Composite pattern, and the result is a formation of a hierarchical tree structure shown **Figure 14** (p. 41). This includes objects instantiated from AtomicComponent relating to the tree leaves, and
instantiated CompositionConnector objects the tree nodes. Because the collection is a resizable array data structure that stores a list of XMANPart objects, the resulting type of tree is nary. Further, because the list can contain other CompositionConnector objects, the tree can be nested to an arbitrary depth (Sargeant, 2012). This is because as a list of composedParts is iterated, the invoke method of each XMANPart of the list is also subsequently called by the connector. If that XMANPart is another connector then that can be subsequently called again, and so on.

5.2.2.5 Pipe And Sequencer Connectors

Figure 26 A diagram showing Sequencer and Pipe that inherits from CompositionConnector.

Figure 26 shows the composition connector classes, including Pipe and Sequencer. They are the concrete implementations of the CompositionConnector class, and representative of Pipe and Sequencer exogenous composition connectors of the C-based version of the X-MAN tool described in Subsection 3.3.3 (Encapsulated Components And Connectors, p. 36). Both connectors compose XMANParts together, by first accepting components as parameters through the object’s constructor, and then storing them as a list of XMANPart objects.

When the invoke method of a Pipe or Sequencer is called, it then calls two methods of the connected component as it iterates through the list of components. Firstly, the checkExtendedArrayList method is called from the AtomicComponents to check that the ExtendedArrayList passed to atomic
components contains the set of key-values that are required, in order to function as expected. For other composition connectors and composite components this is not necessary because they only pass the data. However, for atomic components the values are used and checking is necessary. Checking is achieved by comparing the keys contained in `valueList` with the strings contained in `requiredKeys`.

Next the `invoke` method of the composed components is called. However, there is difference between how the Pipe and Sequencer passes data. Specifically: (1) the Pipe passes data to and relays returned data between components, and (2) Sequencer only passes data to the components that are composed. This difference is demonstrated by what is passed to the `invoke` method of the composed components.

```java
public class Pipe <K,V> extends CompositionConnector <K,V>
{
    ...
    public void invoke(ExtendedArrayList <K, V> valueList)
    {
        for(XMANPart <K,V> part: composedParts)
        {
            part.checkExtendedArrayList(valueList);
            part.invoke(valueList);
        }
    }
}
```

**Table 3** The `invoke` method of the Pipe class.

*Table 3* shows the `invoke` method for the Pipe class. This shows that the Pipe connector passes the single, system-wide `ExtendedArrayList` called `valueList` to each composed component, as it iterates through the list of `composedParts`. This way data is not only passed to, but also between composed components. This is because all the atomic components are adding and mutating the same set of values contained in the `valueList`.

However, the Sequencer is significantly more complex to avoid the mutation of the `valueList` and therefore is passed on to the next components. This is a consequence of the use of the single, system-wide `ExtendedArrayList` and
involves the use of two more ExtendedArrayList objects. Please refer to Appendix A (p. 110) for a more detailed explanation.

5.2.2.6 CompositeComponent

![CompositeComponent class diagram](image)

Figure 27 The CompositeComponent class diagram.

In the X-MAN component model, the composite component encapsulates all containing the X-MAN types, and the CompositeComponent class embodies this in the General Framework. As is shown in Figure 27, the structure of the General Framework differs from the Composite pattern because of the inclusion of the CompositionConnector class, and because of its relationship with CompositionConnector class. In the Composite pattern, the Composite class had an aggregated association with the parent Component class.

However, because in the X-MAN component model the exogenous connectors decouple composition from X-MAN components, in the General Framework the aggregation association has been separated out and has been placed into the CompositionConnector class. The new relationship, indicated formed with, is a reflection of the logical structure of the X-MAN component model and an indication that there is a clear distinction between the classes CompositeComponent and CompositionConnector.

5.2.2.7 The Application Of The Generic Framework

An application of the Generic Framework and CompositionConnectors, replicates the hierarchical tree structures that are formed from the application of X-MAN component model, shown in Figure 28 (p. 69).
Through the use of exogenous composition connectors, the X-MAN component model produces a hierarchical tree structure. Because of the uniform representation, all X-MAN types via a common shared interface specified in the abstract XMANPart class. As in X-MAN, all instantiated objects that are derived from that class can be iterated over in a depth first traversal.

### 5.2.3 Utilities

The Utils package was created to hold useful classes that helped support development, testing and evaluation work. This includes the ExtendedArrayList and the Message class used for debugging and notifications during simulations.

#### 5.2.3.1 ExtendedArrayList

As discussed in Subsection 5.2.1.1 (p. 58), the ExtendedArrayList is the data passing mechanism for the two Java re-implementations. However, as shown by Figure 29 (p. 70) there is additional functionality that the ExtendedArrayList provides. This includes removing an OrderedPair and the
ability to copy elements in other ExtendedArrayLists to the current through the `addExtendedArrayList` interface. This functionality is specifically for the Sequencer composition connector and is used to make sure that data passed to but not between composed component, which described further in Appendix A (p. 110).

Figure 29 The generic ExtendedArrayList class diagram and its dependencies.

The ExtendedArrayList also includes the `printValues` interface that prints all the key-value elements. To do this, it makes use of the Message class, described next.

5.2.3.2 Message

The Message class’ primary responsibility was to be a single place for all the invoked classes to be able to send textual outputs to console. This included all the sequences of calls and operations that occur as a simulation is running, as well as testing and debugging.
The Message class (Figure 30) is an application of the Singleton pattern (Gamma et al., 1995, p. 127), which is an object that returns and only permits a single instance of itself. The reason why it was used was the due to the ability to keep trace of the depth of the first traversal of the tree structure. At various points through that traversal, various public methods would print specific messages for specific tasks and parts of the system. This includes adding and changing elements in an ExtendedArrayList and if debug is enabled, the results of calculations in the atomic component. Outputs of the use of the Message class can be found in Table 9 (Appendix B, p. 111) and Table 10 (Appendix C, p. 113).

5.2.4 Re-implementation Of The Antiskid System

The following sections refer to the specific reimplementation of the three subsystems of the Nalmpantis (2011) design and implementation discussed in Section 5.1, and shown in Figure 17 (p. 51). These make up parts of the two X-MAN-like re-implementations that are as close as possible to the original atomic components.
5.2.4.1 Interface Components

![Diagram of Interface Components](image)

**Figure 31** A diagram of the interface atomic component classes that inherit from the superclass AtomicComponent.

Firstly, there was a realisation of interface components from the original Nalmpantis (2011) implementation that were primarily concerned with the reading of sensor measurements, including modelling a caution light. Referring to **Figure 31**, the more significant components include:

1) **Sensor**: The purpose of the component is to read in a Float value from a text file. When a Float value is read then it is assigned to a Float variable and added to the ExtendedArrayList. The values that it reads varies, depending on the file that it reads from, and the file names vary on the key that is assigned to each instantiated sensor object. For the Nalmpantis (2011) implementation, the three values that the sensor reads are: aircraft speed, wheel speed and whether there is a degree of weight on the wheels (touch down).

2) **CheckTouchDown**: The component is used to check that the touchDown values that are read by two separate sensors are both the same value. Similarly to the logical conjugation in an AND gate, if they are, then the touchDown value is 1. If they are not, or both 0, the touchDown value is 0.
3) CheckFloatSensor: The component is used to find the average between either two aircraft or wheel speed sensor values.

### 5.2.4.2 Functional And Simulation Components

**Figure 32** A diagram of the functional and simulation atomic component classes that inherit from the superclass AtomicComponent.

Secondly, the functional and simulation components of the overall from the original Nalmpantis (2011) implementation were created. The functional components are primarily providing a reference speed used to calculate the degree of error, and a basic mechanism to stop the wheels locking up (wheel speed below 30%). The simulation components (Figure 32) include one that varies the final output of the system based on previous values, which include:

1) LowMedMax: This produces a reference speed to which the PID tries to adjust the braking, and as a result alters the wheel speed. The resultant reference speed can be altered to three different arbitrary levels: 87%, 95% and 100% of aircraft speed.

2) AntiLock: The component checks if the wheel speed is less than an arbitrary level of aircraft speed. If the wheel speed is less that 30%, then antilock is
activated, and if it is greater, then it is deactivated. The antilock value is then used later in the Wheel component.

3) Wheel: The component crudely checks if certain values are enabled or disabled and varies the output of the system based on these values, such as:

a. If the caution light has been enabled from the interface components then the resultant output of the system is 0 (i.e. no braking).

b. If the caution light has been disabled and touch down has occurred, then the output of the system is the same as the current wheel speed. This could be trying to simulate wheel speed catching up with aircraft speed.

c. If the caution light is disabled, touch down has not occurred, and antilock has been enabled (wheel speed is less than 30% of aircraft speed), then the result of the system is the same as the current aircraft speed. This is believed to be simulating the initial landing phase where no breaking occurs.

d. If none of the above situations have occurred then the resultant when speed is adjusted with a modifier provided by the PID.

5.2.4.3 PID Components

A specific composite component of the functional subsystem of overall anti-lock braking system is called the Proportional-Integral-Differential or PID. As previously described in Subsection 4.2.2 (Control Loop Feedback, p. 48), the PID calculates a modifier that is applied to braking. As is shown by the structure of the Nalmpantis (2011) PID design in Figure 33 (p. 75), it is made up of atomic and composite components and connectors.
Figure 33 The structure of the PID composite (adapted from Nalmpantis, 2011, p. 88).

Figure 34 A diagram of the PID atomic component classes that inherit from the superclass AtomicComponent.

Figure 34 shows the atomic component classes of the PID composite of the current re-implementation. These classes are the concrete realisations of the PID composite from the original Nalmpantis (2011, p. 52) implementation, including:

1) Error: An error is calculated by subtracting the expected reference speed from the actual wheel speed measured by the wheel sensors. The difference between these two values is then passed to the P, I and D objects via a Pipe connector.
2) P: The error value is multiplied by the value of variable $k_P$. The result is the *proportional error correction*, used to alter the braking pressure, which in turn regulates wheel speed.

3) I: The error is integrated with respect to time. The resulting integral is then multiplied with value of the variable $k_I$, which is used to keep brake performance as optimal as possible. In the original Nalmpantis (2011) implementation, the value of integral was written out to a file. Then, the next iteration, the I atomic component is used, that value is read back in and used in the calculation again. For the purposes of this project, the value of the integral is held as an instance variable.

4) D: The error is differentiated with respect to time. The differential is multiplied with the value of variable $k_D$. The purpose of this calculation is to compute a rate of *compensation* in order to correct the brake’s hydraulic system. In the same way as the integral variable in the I atomic component, the previous error value is held in the `preError` instance variable in the D atomic component. It is used in the calculation of the first derivative.

5) ADD3: The outputs of the three P, I and D objects are summated. This value is then used to control an anti-skid valve; thus, altering the application of braking that in turn regulates wheel speed.

5.2.4.4 Testing

In order to test that each atomic component was producing the right output given a set of inputs, the JUnit (2012) testing framework for Java was used. It contains a set of classes that are extended to provide functionalities, such as the assert function, which compares the actual and expected results in a test (Koskella, 2008, pp. 36-37). The JUnit framework is part of the xUnit family (Fowler, n.d.), which permits unit testing of multiple, differing languages. The family provides a code library that allows the writing of unit test files, but *test runners* (Koskella, 2008, p. 36) allow the automatic running of test files. They also provide a collective report on the results in textual or graphical form.
Some graphical runners include a helpful progress bar that shows the percentage of valid and invalid tests.

This organisation and automation of unit tests and tools are widely used to support software development. Mostly widely known for use with Test-Driven Development (TDD) in combination with processes such as regression (changes in the code base), integration (software modules tested in combination) and end-to-end system (including hardware) testing (Jay & Stevens, 2013). The motivation for unit tests is to quickly and repeatedly find potential errors, and to provide feedback on the validity of elemental parts of the system. The degree of this validity is specified by the tests, and the extent of test coverage (Jay & Stevens, 2013). Because of the X-MAN component model, the system is broken into small units of computation, the use unit tests mapped very cleanly onto the assessment of the correctness of those units, especially the atomic components of the original Nalmpantis (2011) implementation. Unit tests were particularly useful as a means of separating test and implementation code, which might have been more coupled without.

5.2.4.5 Building The Tree Structures

![Diagram](image)

**Figure 35** The simplified re-implementation of the original Nalmpantis (2011) ABS system.

Now that the GeneralFramework and the components of the original Nalmpantis (2011) implementation have been completed, a way of instantiating and forming them into the tree structures shown in **Figure 35** at
run-time was required. In the original version, the X-MAN tool formed these structures. However, for the current two versions the tree has to be completed programmatically. A variation of the Builder pattern (Gamma et al., 1995, p. 95) was employed to create composites for a modified version of the second simpler version, shown in Figure 17 (p. 51). The details of how this was achieved are outlined in Appendix E (p. 115). This builder pattern was also used to create the first version that is a closer representation the final design of the original implementation, which is shown in Figure 20 in Subsection 5.1.2 (Final Implementation Structure, p. 54) and Figure 42 (Appendix D, p. 114). Although not discussed in this dissertation for the sake of brevity, the techniques and underlying code is very similar in the second, simpler version.

5.2.5 The Object-Oriented Re-implementation

In order to compare approaches, a reasonable object-oriented (OO) design of the ABS system was created, that had reproducible simulation results from previous X-MAN-like re-implementations. In order to do that, there was an initial emphasis on finding domain concepts, or objects, that best described the problem domain (Larman, 2005, p. 7). This was achieved by returning back to Figure 15 (p. 48) used in the Systems Analysis (Subsection 4.2.1, p. 47).

Figure 36 A UML class diagram showing the association between ControlUnit and the Sensor and Cockpit interface classes.
Figure 36 (p. 78) shows the result of the identification of the distinct software objects and their associations, in order to match the original system requirements. The classes identified included: (1) Cockpit, (2) Aircraft and Wheel Sensors, (3) PID, and (4) ControlUnit.

The following responsibilities that have been identified for objects include:

1) Cockpit: This object allows a degree of manual override of the ABS system, from higher functioning systems that determine failure and a caution light displayed to pilot.

2) Sensor: This captures common properties that represent different types of sensors. They all have the responsibility of providing accurate measurements that the ABS systems needs, to be able to control braking, specifically:
   a. WheelSensor: Measures if there is weight on and the speed of the wheel.
   b. AircraftSensor: Measures the speed of the aircraft.

3) PID: Calculates the adjustment in braking in order to modify the current wheel speed and provides it to the ControlUnit.

4) ControlUnit: The ControlUnit will modify the output and ultimately the wheel speed through braking. This is determined by the collected sensor measurements, overriding pre-sets and the PID output calculation.

The objects collaborate with each other in a number of ways. The interaction between the Sensor and Cockpit interface concepts is clear. The PID provides a modifier to the wheel speed to the ControlUnit. Simply, PID offers a more refined and the ControlUnit offers a more crude level of braking control. However, it is the ControlUnit what determines what form of control is required, whether that is braking adjustment from the PID or not.

In order to be as objective as possible, a version of the Message class described in Subsection 5.2.3.2 (p. 70) was also used for OO re-implementation. An output of the test run using the Message class can be found in Appendix C (Table 10, p. 113).
SIMULATIONS AND METRICS

Introduction

The assessment of the X-MAN-like and object-oriented (OO) Java re- implementations involves 3 parts:

1) The reproduction of original simulation and the 6 original test cases used to evaluate the ABS project.

2) Simulations that are extended over time to understand better the adaptive modification that the PID has on wheel speed.

3) Metrics are gathered on the re-implementations. These can then be used to understand the degree of complexity in terms of cohesion, coupling and size.

6.1 Reproduction Of Original Simulations

Nalmpantis (2011, pp. 71 - 78) tested the ABS system in a number of ways to make sure it behaved as expected. This included a number of phases that included the unit, integration, system and regression testing during the development. It is assumed that this must have been achieved using external tools that were not specified, nor contained in the provided code solution.

Nalmpantis (2011) completed a series of six simulation tests using the X-MAN tool in the final test phase. They were designed to replicate the various aspects of braking simulation by varying sensor reading inputs and pre-sets. The sensor input values, read in from text files, determined these situations and included wheel speed, aircraft speed and weight on wheels. Using an XML file shown in Table 4 (p. 81), pre-set inputs were also used in the simulation tests, which include the values for antiskid (as), failure,
lowmedmax (lmm) and deltat (dt). These inputs and pre-sets of the final simulations are re-used to verify whether the various re-implementations in Java have been correctly reproduced.

Table 4 An example of XML test file containing the pre-set inputs and expected outputs required for simulation (Nalmpantis, 2011, p. 73).

Once the simulation is complete the output is verified with respect to the expected output, using comparison operators built into X-MAN such as EQ (equals) shown in Table 4. Because float primitives were used there was a slight imprecision due to rounding errors. Therefore a tolerance in the margin of error is required.

The current X-MAN-like and OO re-implementations in Java reproduced the same results as the original implementation, within a small margin of error. Table 12 (Appendix F, p. 121) shows the test cases and the results of the simulations. These reproducible results extend to the final test cases 6 – 1 and 6 – 2, that both use PID in order to adjust wheel speed. This will be discussed in more detail next.
### 6.1.1 Adaptive Skid Simulation

![Figure 37](image-url) The results of a comparison between aircraft and wheel speed Ming, Hong, and Rupeng (2011) using a more complex P-PBM-D controller (modified by Nalmpantis, 2011, p. 75).

In order to test whether the PID unit was working, Nalmpantis (2011) used a published simulation by Ming et al. (2011) that is shown in **Figure 37**. The results are not absolutely comparable because the published simulation used a more complicated controller that had a different I component called a pressure-bias-modulated (PBM) module. Through this, the "method to control the braking system is significantly better than using the original PID regulator" (Ming et al., 2011). However, Nalmpantis (2011) compared the results of the output of his ABS system with published results at the 5-second mark, ran the simulation for one hundredth a second (the delta in time is 100) and the results are shown in **Table 5** (p. 83). As can be seen, given this short iteration of time and a single cycle through the simulation the wheel speed is adjusted,
the results are within a rough approximation of the relationship to the wheel speed at $t = 5.1$ in the graph. As is also shown in Table 5, the X-MAN-like and OO Java re-implementations are able to produce the same results within a degree of error of 0.0001.

<table>
<thead>
<tr>
<th>Sensor Inputs</th>
<th>Test 6 - 1 &quot;time = 5 sec&quot;</th>
<th>Test 6 - 2 &quot;time = 5 sec&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preset Inputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiSkid</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Failure</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Low Med Max</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
</tr>
<tr>
<td>Delta T</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Touch Down 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Touch Down 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wheel Speed 1</td>
<td>43</td>
<td>42</td>
</tr>
<tr>
<td>Wheel Speed 2</td>
<td>43</td>
<td>42</td>
</tr>
<tr>
<td>Aircraft Speed 1</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>Aircraft Speed 2</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>XMAN C-based</td>
<td>42.17055</td>
<td>42.17055</td>
</tr>
<tr>
<td>XMAN Java</td>
<td>42.17056</td>
<td>42.17056</td>
</tr>
<tr>
<td>OO Java</td>
<td>42.17056</td>
<td>42.17056</td>
</tr>
<tr>
<td>XMAN C-based</td>
<td>41.56365</td>
<td>41.56366</td>
</tr>
<tr>
<td>XMAN Java</td>
<td>41.56366</td>
<td>41.56366</td>
</tr>
<tr>
<td>OO Java</td>
<td>41.56366</td>
<td>41.56366</td>
</tr>
</tbody>
</table>

Table 5 The reproduction of the results of the original C-Based version, with the X-MAN-like and OO Java re-implementations showing the same results.

The outputs of the simulation for test case 6 – 2 to console, using the Message class, are also show in Table 9 (Appendix B, p. 111) for the X-MAN-like re-implementation, and Table 10 (Appendix C, p. 113) for the OO re-implementation.

This shows that given the specific situation and the set of inputs and sensor readings, the results roughly correlate with the published simulation by Ming et al. (2011). However, what is not shown is what happens to the results if delta increments are increased or reduced, or if the simulation is repeated at a different point in time, say at 10 seconds rather than 5. The next chapter endeavours to briefly understand what happens as these factors are varied and what happens when the simulations are repeated over a greater length of time.

6.1.2 Simulations Over Time

One of the fundamental problems with the previous set of tests is that the looping mechanism was not used. This limited the extent of the simulation because the feedback loop property of the PID, described in Subsection 4.2.2
(Control Loop Feedback, p. 48), was not tested. This is because the modified values of the previous output needed to be fed back into loop as the current input to reflect changes in wheel speed. Also, changes in aircraft speed had to be provided. These changes are predetermined and not influenced by variations in wheel speed. This is unrealistic because any changes in wheel speed would have a direct and comparative influence on the speed of the aircraft. However, this factor was removed to reduce complexity and to purely focus on the adjustment of the PID controller on wheel speed. All of this is achieved more easily and over a greater length of time through the use of a high level looping mechanism.

Figure 38 shows one of the first attempts at such a simulation covering 17 seconds. The simulation shows the PID controller trying to regulate wheel speed and bring it into line with the reference speed, which is 87% of aircraft speed. The wheel speed oscillates in conjunction with the reference speed. The fluctuation is dramatic, but never matches the wheel speed with the reference speed. However, this simulation is not wholly realistic because the wheel speed exceeds aircraft speed. As shown in Figure 37, this would not be possible unless the wheel is accelerating. However, the wheel would not be realistically doing this under the deceleration of braking conditions.

![Graph](image.png)

**Figure 38** The first simulation over time covering 17 second.
Figure 39 shows a subsequently attempted simulation covering 12 seconds. This time the frequency has been increased by a factor of 10. Again, the simulation shows the PID controller trying to synchronize wheel speed with the reference speed. The degree of oscillation has been dampened and the wheel speed is in line with the reference speed in just over two seconds. But the wheel speed exceeds aircraft speed, and as with simulation shown in Figure 38 (p. 84), this would not be possible without wheel acceleration.

![Figure 39](image)

**Figure 39** The first simulation over time covering 12 second with an increase in frequency at which the controller polls for the measurements.

Finally, Figure 40 (p. 86) shows a final attempt of a simulation covering 12 seconds. The frequency has been increased by a factor of 10 once again. This time the PID controller is able to quickly regulate wheel speed and bring it in line with the reference speed. The wheel speed also does not exceed wheel speed. Wheel speed is once again reduced dramatically at the 6-second mark and the same behaviour is repeated.

These simulations demonstrate that the PID controller is working to a limited degree, provided that sampling is frequent enough. The PID controller can be shown to adapt the wheel speed to be in line with the reference speed, however this adaptation is not wholly accurate with realistic braking system behaviours shown in Figure 37 (p. 82).
Figure 40 The final simulation over time covering 12 second.

The object-oriented re-implementation was able to reproduce the same results as the X-MAN-like re-implementation in all simulations conducted.

6.2 Metrics

Intuitively, the design of the object-oriented approach (Subsection 5.2.5, p. 78) is good in terms of Cohesion and Coupling principles. The UML diagram (Figure 36, p. 78) shows that the Cohesion is good because every entity within the domain is well defined. The maximum Coupling is 2: between the AircraftSensor and the WheelSensor coupled with the abstract Sensor and the ControlUnit classes.

However, a more objective approach is needed to quantitatively measure and compare the object-oriented with the X-MAN-like re-implementations of the original Nalmpantis (2011) ABS system. To attempt this, metrics proposed by Sant’anna, Garcia, Chavez, Lucena, and Staa (2003, pp. 4-5) were used to establish the differences in coupling, cohesion and size. Internal attributes of classes are measured and used as part of an assessment framework that determined factors such as understandability and flexibility. These factors could then in turn be used to assess relative qualities such as reusability and maintainability. The metrics suite also includes the measuring of the separations of concerns in order to compare aspect-oriented and object-
oriented approaches. However, these metrics have not been used because they are not relevant to the current project.

In brief, the metrics by Sant'anna et al. (2003) are described by the following:

3.2 Coupling Metrics: Indicate the strength of interconnections between concepts. Higher values indicate that concepts are more coupled and thus have a higher dependency, and vice versa.

3.2.1 Coupling between Components (CBC): The amount that a component is coupled with other components. It counts the amount of use attributes, including: formal parameters, return types and local variables.

3.2.2 Depth of Inheritance Tree (DIT): The maximum depth of inheritance tree of the current concern, from node to root.

3.3 Cohesion Metrics: Determine the lack of cohesion by measuring the closeness of the internal relationships within a component.

3.3.1 Lack of Cohesion in Operations (LCOO): Measures the levels of cohesion by finding the difference between methods that access the same instance variables and those that do not. If the methods access separate instance variables, then the class represents two different concepts because the operations are using disjoint data.

3.4 Size Metrics: Measures the extent of the system’s design and source code.

3.4.1 Vocabulary Size (VS): Counts the number of classes in a system.

3.4.2 Lines of Code (LOC): Counts the number of lines of code. This assumes that the same programing style is used in both approaches in order for them to be compared using this metric.

3.4.3 Number of Attributes (NOA): Counts the number of attributes. Inherited attributes are not included.

3.4.4 Weighted Operations per Component (WOC): Counts the number of parameters of an operation to assess its complexity.
Table 6 shows the cohesion, coupling and size metrics gathered, based on the descriptions of the Sant'anna et al. (2003) paper. The table and the descriptions above both have the same numbering scheme used in the paper as a form of reference.

<table>
<thead>
<tr>
<th>Table 6 Coupling Metrics</th>
<th>X-MAN-like Java</th>
<th>OO Java</th>
<th>Difference Of X-MAN-like And OO</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 Coupling between Components (CBC)</td>
<td>96</td>
<td>108</td>
<td>-12</td>
</tr>
<tr>
<td>3.2.2 Depth of Inheritance Tree (DIT)</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3.3 Cohesion Metrics</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1 Lack of Cohesion in Operations (LCOO)</td>
<td>-19</td>
<td>-4</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3.4 Size Metrics</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1 Vocabulary Size (VS)</td>
<td>12</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3.4.2 Lines of Code (LOC)</td>
<td>554</td>
<td>460</td>
<td>94</td>
</tr>
<tr>
<td>3.4.3 Number of Attributes (NOA)</td>
<td>12</td>
<td>19</td>
<td>-7</td>
</tr>
<tr>
<td>3.4.4 Weighted Operations per Component (WOC)</td>
<td>22</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6 Cohesion, coupling and size metrics gathered from the two implementations.

The metrics only include elements that are in the domain of the ABS system that has been re-implemented. Where appropriate, the metrics have been gathered to take into consideration functionality that is both inherited and not inherited. For example, the metrics for the X-MAN-like implementation have not included inherited functionality with the exception of the Depth of Inheritance Tree (DIT) metric, where appropriate functionality both XMANPart and AtomicComponent classes were included (Figure 24, p. 64). However, in the OO re-implementation the inherited functionality was included throughout the measurements. As shown by Figure 36 (p. 78), the Sensor class is part of the domain of the ABS system and therefore is incorporated when calculating the AircraftSensor and WheelSensor classes.

The Messenger class described in Subsection 5.2.3.2 (Message, p. 70) was not included in these metrics. However, because both approaches make use of it, all uses or references held in variables to it were included in the results. The programming style was also kept consistent throughout both approaches in order to be objectively comparable.

Positive results signify an advantage in the object-oriented approach in the context of the current re-implementations, while negative result suggest a benefit in the X-MAN-like component-based approach.
6.2.1 Coupling Metrics

The Coupling between Components (CBC) metrics shows the greatest distinction between the approaches, with a difference of -12 suggesting that the X-MAN-like implementation maintains lower amount of coupling than the object-oriented approach. However, the object-oriented approach does inherit methods and attributes from one class less in the Depth of Inheritance Tree (DIT) metrics. Overall, the X-MAN-like Java re-implementation does show tangible difference and therefore shows a value in taking this methodology in terms of coupling.

6.2.2 Cohesion Metrics

Both approaches are shown to be uncohesive according to Lack of Cohesion in Operations (LCOO) metrics, but the X-MAN-like re-implementation are shown to be marginally less cohesive. This is because there is one method contained in the atomic components and therefore any attributes cannot be shared between multiple methods in order to demonstrate cohesiveness according to the measurements. This suggests that the object-oriented approach is more highly cohesive in comparison. The difference provided is an absolute or modulus value in order to better represent the comparison. This is because according to the methodology, the metrics produce a 0 if a class is cohesive or a negative number if is not. The actual difference would be -15, but this does not indicate an advantage in the X-MAN-like approach.

6.2.3 Size Metrics

The calculations have shown a sizable benefit overall to the object-oriented approach in terms of Vocabulary Size (VS) and Lines of Code (LOC). This difference can be explained due to the numerous amounts of differing atomic component classes and their associated methods. However, a benefit of the X-MAN-like approach is shown when using the Number of Attributes (NOA) metric. This is because most of the instance variables are contained in the inherited X-MANPart and AtomicComponent classes and therefore are not measured. There is no noteworthy differential Weighted Operations per
Component (WCO) metric and consequently no benefit shown in either approach.

Overall, the cohesion and size metrics have suggested a small advantage in the object-oriented approach, but there is a surprising benefit terms of lower coupling demonstrated by the X-MAN-like approach. However, any external qualities such as reusability and maintainability cannot be established because these metrics do not demonstrate any significant dissimilarity between the approaches. Further extrapolation is also compounded by a lack of similar studies of this kind in order to relate to. Additional empirical studies, such as the one conducted by the Sant'anna et al. (2003), could also be conducted with controlled tests in order to more definitively ascertain these qualities.
Chapter 7

COMPARING AND CONTRASTING THE APPROACHES

Introduction

Through this work many claims and counter claims about the benefits of taking the object and component approaches have been investigated. This chapter seeks briefly to revisit both claims and counter claims to assess whether there is any merit in them. This is achieved by answering a set of research questions:

1) What contrasts and comparisons can be drawn between X-MAN and X-MAN simulated in Java?

2) What are the advantages and disadvantages of the X-MAN-like Java re-implementation compared to the plain object-oriented re-implementation?

3) How helpful was inheritance in aiding the development, given that the X-MAN component model does not support it?

7.1 The Differences And Similarities

What contrasts and comparisons can be drawn between X-MAN and X-MAN simulated in Java?

The answer to this question has been broken down into a number of subsections.
7.1.1 Atomic Component Reusability

Table 7 An exemplar of an atomic component, called increment (Tran, 2012).

<table>
<thead>
<tr>
<th>//DATA@</th>
</tr>
</thead>
<tbody>
<tr>
<td>int delta = 1;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>//METHOD@</th>
</tr>
</thead>
<tbody>
<tr>
<td>void increment ( int m, int &amp;n )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>n = m + delta;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

There is a limit to the reusability of atomic components, because of a general problem in the way that they are designed. This is shown by the atomic component increment, in Table 7, which is taken from an official X-MAN tutorial (Tran, 2012). The atomic component has a variable called delta, which holds the value 1. This value is used to increment m, and then yield the result back by assigning to the value n. However, delta will always remain the same value, and cannot change once it has been initialized and deployed as a component.

There was a similar issue with the development of the PID composite in the original Nalmpantis (2011) implementation. This is because the stored values of \( k_P \), \( k_I \), and \( k_D \) are hard coded, and contained local variables within the respective P, I and D atomic components. This means the values of the variables could not vary easily if the PID is reused and applied in a different domain. If they were to change, then a component developer would manually change the assigned values in the code, or even copy or recreate the atomic component. A more elegant solution, realised in the current re-implementation in Java, would allow the data held by these variables to become externalised and to be passed as a parameter through the object’s constructor, which only assigns the value to an instance variable, as the object is being instantiated. This allows the values of the constant variables and the code to vary independently of each other, and decouples the data from the actual code. This is the application of Data-Driven Design, which is a form of the Protected Variations principle (Larman, 2005, p. 428).
7.1.2 Data Passing Mechanisms

As is discussed in Section 3.4.4 (Coupling, p. 42), X-MAN uses a list of arguments, and their types, not only to pass data to, but also from atomic components. The atomic components do not make use of the return statement as a mechanism to pass data. This is to remove the ability of a component to determine when to relinquish control back to the caller; thus, all instructions inside the component have to be executed sequentially. Instead, a reference parameter of an address to an extrinsic variable is passed with a set of arguments, which is mutated by value assignment in order to return data.

The issue with this approach, is that there is data passing, even in components that appear at first glance to have none. Therefore, the data passing is implicit, particularly when using sequencers. It also shows that there is a direct dependency between the connector and component, and that there is an indirect form of coupling between components via composition connectors.

The ExtendedArrayList was used as the data-passing mechanism in Java that simulated the reference passing aspect in X-MAN. The entire ExtendedArrayList is passed to the atomic components as an object reference. Values are added or mutated through the interface by providing a key for the equivalent mapped value. The ExtendedArrayList could be greatly optimized and can be effectively thought of as a Map. The benefits of using the ExtendedArrayList, are that concepts in the GeneralFramework can share a common invoke method, and therefore are more reusable.

A degree of coupling is necessary in order to have a working system. Because of the reference parameter, the indirect coupling between components in the X-MAN component model is at least between components that are connected directly through a composition connector.

In the approach taken for the X-MAN simulated in Java, the levels of coupling have been distorted because all the values that a system uses are contained in a single, system-wide ExtendedArrayList. Because all components have access to it, the level of coupling in the X-MAN simulated in Java is at least as
great as the amount of components that use the ExtendedArrayList. This is not scalable just because of performance reasons, but additionally it would be increasingly harder to cope with the overhead of managing it.

This level of coupling is a result of the way that the composition connectors were implemented. If the connectors were changed to instantiate a new ArrayList or Map every time a connector was invoked, then the level of coupling would be contained inside that composite. Therefore, the amount of coupling is the same in both cases, but the data-passing mechanism is more explicit in the X-MAN-like Java version.

### 7.1.3 Looping

Although the Loop connector was shown to be available at the time (Figure 13, p. 40), it was consistently not stated nor used in the implementations of various M.Sc. projects, including Kang (2010), Nalmpantis (2011) and Popoola (2011). Repeated execution of the simulation of the Nalmpantis (2011) implementation was not possible, because the loop connector was not used. This meant that the feedback loop properties of the system were not originally simulated in the adaptive skid simulations (Subsection 6.1.1, p. 82).

The benefits of using iterator-based looping constructs were shown in the subsequent simulations over time (Subsection 6.1.2, p. 83), in order to show a complete understanding of how the PID regulates wheel speed according to the reference speed.

The lack of use of a loop connector could be an artefact of the X-MAN tool, as Nalmpantis (2011, p. 80) explained that the simulations in X-MAN only supported discrete time and open loop controllers at that time.

### 7.1.4 States

As is mentioned in Section 3.4.4 (Coupling, p. 42) an X-MAN atomic component is essentially a C function. Although it has local variables available to hold values while the function is used, the Nalmpantis (2011) implementation did not make use of global variables that exist outside the
function, to retain values when the function is not used. As a result, the atomic components that implemented were unable to hold a state and it became more problematic to store and then use a value from a previous calculation.

An example of this problem, and how this was solved by Nalmpantis (2011) for the I and D atomic components of the PID composite, is briefly mentioned in Section 5.2.4.3 (PID Components, p. 74). The solution involved writing out the values of the integral and preError to a text file, so that they could be read and used again when the I and D are invoked once more.

When re-implementing the current XMAN-like versions in Java, the values of integral and preError could be simply held as instance variables, and the values reused the next time the invoke method of I and D classes are called.

This could also be solved in X-MAN, by making the local variables of the atomic components static. That way the values are held throughout the lifetime of the program, and static variables inside the atomic component retain changed values for subsequent method invocations. The fact that this was not done by Nalmpantis (2011) is interesting. This is either because Nalmpantis (2011) was not aware of this, or that in the X-MAN component model atomic components are not supposed to contain a state. The X-MAN literature and the result of discussions with K.K. Lau (John Sargeant, 2012, pers. comm.) were ambiguous about this issue, but if atomic components are not able to retain a state, then it seems to follow a purely functional approach. The reasoning might be that components are less reusable as a result of holding a state. However, the fact that states cannot be easily held, does make it difficult to model concepts that inherently require a state, such as I and D. Therefore, there is a distinct advantage in having instance variables.

However, this could be a misunderstanding by Nalmpantis (2011) and indeed, as is shown in Table 7 (p. 92), it seems that global variables can be initialized when components are deployed.
7.1.5 Structure

There is an interesting difference in comparing the structures of the various ABS implementations, including: (1) the final Nalmpantis (2011) implementation (Figure 20, p. 54), (2) the matching re-implementation of the Nalmpantis (2011) design (Figure 42, p. 114) and (3) the simplified version (Figure 35, p. 77).

Nalmpantis (2011) created a very deep and complex system structure in the final implementation, shown in Figure 20 (p. 54). This is a result of deciding to reduce the passing of redundant data, by passing data to the components only when they are required. The consequence of this was a decrease in the number of arguments passed as parameters to components, but also an increased amount of repeated execution of operations in order to provide the same data, but at different points in the simulation. Nalmpantis (2011, p. 79) later proposed a new connector to adapt interfaces of components and reduce redundant data passing.

The issue was resolved to a certain extent through the use of the ExtendedArrayList in the X-MAN-like re-implementation in Java. As the system’s data is contained in the ExtendedArrayList, a value could be added and used by multiple components that require it in the system. This approach not only resulted in a simplified structure shown in Figure 35 (p. 77), but also a more uniform interface across the GeneralFramework. However, a system having global data is problematic, and will later cause issues with reusability.

Otherwise this approach had benefits in terms of being easier to understand and being similar to the original design, shown in Figure 17 in Subsection 5.1 (Design, p. 51).

7.1.6 Tools

Combining both the GeneralFramework and Composite Tree Builder emulates aspects of the X-MAN tool. This relates to the “Builder” tool shown in Figure 41 (p. 97).
The Composite Tree Builder (Subsection 5.2.4.2, Building The Tree Structures, p. 77) uses GeneralFramework (Section 5.2.2, p. 62) to then instantiate and compose components that are used at run-time. This has similarities to the “Builder” tool of the idealized component life cycle. However, it differs from the X-MAN tools because composing components is achieved programmatically instead of visually. In principle this could also be achievable visually, in a way similar to GUI builders in the Visual Studio, NetBeans and Eclipse IDEs.

![Diagram](image)

**Figure 41** A ‘Category 1’ component life cycle also known as ‘Design without Repository’ (K.-K. Lau & Taweel, 2011).

However, the GeneralFramework and Composite Tree Builder do not support the assembler tool of the deployment phase. As shown in **Figure 41**, this means that it falls into the Category 1, and does not meet the idealised component life cycle (K.-K. Lau & Wang, 2007). This is because, although exogenous connectors are supported, the repository in the builder phase does not exist in order to retrieve prebuilt components. The X-MAN tool does support both of these aspects and can compose X-MAN tree structures by dragging, drooping and connecting pre-built components in the assembler view shown in **Figure 13** (p. 40).

The assembler phase and repository are needed to support the system life cycle shown in **Figure 2** in Section 3.2 (A Component Life Cycle, p. 29) that the component lifecycle feeds into. However, it is difficult to ascertain how useful they would both be to the development of the ABS system, until another different system is built and the components reused. So, although the
repository and assembler are not supported in the current re-implementation, and therefore not able to meet the desirable features of the idealized component life cycle, its benefits are not wholly clear in this context.

7.1.7 Encapsulation

Pfister and Szyperski (1996) argued that objects are too narrowly focussed, and that higher level concepts would be more meaningful. They also state that encapsulation is required for these higher levels constructs.

Nalmpantis (2011, pp. 34-35) agreed with this and along with K.-K. Lau and Elizondo (2010, p. 1166) state that the primary reason objects are not as reusable as components, a result of traditional data passing that involves too much coupling between control and computation. Through the use of exogenous composites, only X-MAN is said to ensure encapsulation of control and computation at all levels formed in the tree structure, and that this is something that object structures do not support (K.-K. Lau & Taweel, 2007). As a consequence, it is able to meet the requirements of the idealized component life cycle (K.-K. Lau & Wang, 2007).

As the work conducted in this project shows, control and computation can be separated using the Java object-oriented programming language. So, in the context of this particular application, the claim that objects cannot encapsulate control and computation at all levels is now wholly inaccurate. However, whether there are other applications where this would be true is yet to be determined, and care needs to be taken in drawing wider conclusions.

7.1.8 Reusability In Instantiation

The X-MAN tool is able to reuse code by retrieving templates of deployed components from the repository, and then composing components together by creating and renaming copies and connecting them in the Assembler view (Tran, 2012). These various template copies are all representative of a single source code text file stored in the repository. Every time one is dragged and dropped in the assembler view, an “instance” of a new component is created.
In Java, dynamic objects are instantiated from static classes and assigned a unique object ID. Once instantiated, references to those objects are returned. Components needed to be composed programmatically in the current project, but a similar tool would make this unnecessary. However, once that had been achieved, creating new instances of the entire system was as simple as calling a builder and retrieving a reference to the assembled system.

By contrast, Nalmpantis (2011) had to link two wheel components to every other component in the system twice over in order to create two wheels for his simulation. As shown in Figure 20 (p. 54) this is not a good solution. In theory, the ABS system could have been created once, and then placed into the repository in its entirety. This would later be retrieved back into the assembler copy of the complete system as a new composite. However, the composite component development was not supported and was restricted by the tool at that time (Nalmpantis, 2011, pp. 79-80).

Similarly, with the XMAN-like re-implementations, two different instances of the ABS system could be simply created by the Composite Tree Builder or a tool, and references returned in order to create two different simulations.

7.2 The Pros And Cons

What are the advantages and disadvantages of the X-MAN-like Java re-implementation compared to the plain object-oriented re-implementation?

The X-MAN component-based approach separates control from computation. This is through the use of exogenous connectors. This restricts system organisation into a hierarchical tree structure that is said to increase the modularity. Because of this, code is claimed to be more reusable.

Atomic components are just responsible for providing the computation that they encapsulate. However, in themselves, they do not wholly represent tangible concepts within the domain model. This was shown to be true in the final Nalmpantis (2011) implementation (Figure 17, p. 51).
This could be due to Nalmpantis (2011) deciding to reduce the amount of redundant data (Subsection 5.1.1, p. 52). As Figure 20 (p. 54) shows, atomic components were used repetitively in order to provide data just as and when it was required by a specific component. However, arguably this data could have been retrieved just a few times at appropriate points in the system flow. Through this decision and the use of the exogenous connector, the coherent concepts of the system design were not easily understandable (Figure 42, p. 114). The final implementation demonstrated code reuse through the repeated calling of the Sensor atomic components 22 times, but it is questionable whether this is a practical form of reuse. This was improved with the use of a simpler version of the X-MAN-like re-implementation (Figure 35, p. 77), which through the use of the GeneralFramework and the ExtendedArrayList, had a design which was closer to the original system design (Figure 17, p. 51) and more reasonable levels of reuse. Regardless, the relationships between the different parts and the behaviour of the system are still not entirely clear.

The OO re-implementation used traditional message passing and mixed control and computation. It had a lot more flexibility in terms of the structure and allowed a far simpler organisation and intuitive comprehensible design (Figure 36, p. 78). This is due to responsibilities being assigned to classes that were much closer to the real-world concepts shown in Figure 15 (p. 48). The inter-relationships and collaborations are much more understandable. Intuitively the cohesion and coupling between components can be seen to be good, which suggests that is it reusable. However, when compared to X-MAN-like re-implementation, the measurements on cohesion, coupling and size outlined in Subsection 6.2 (Metrics, p. 86) have not strongly indicated this. Additional empirical studies and the use of other metrics might result in a stronger understanding of external factors like reusability.

Simulations of the X-MAN-like simulations took much longer to complete than the Java object-oriented re-implementations. Where X-MAN-like simulation took ~16 seconds to complete, the object-oriented Java re-implementation took ~3 seconds. However, although some of this could be explained by the use of exogenous connectors, a more efficient
implementation of a data passing mechanism would probably result in a performance more in line with the object-oriented re-implementation.

Atomic components were easier to unit test because they were just methods that only encapsulate their computation, and functionally provide an output given a set of inputs. However, the amount of cases that were needed to test increased in order to cover an increased amount of possible variations in the use of the class. This is because classes have states that are shared between multiple methods, which meant that the amount of testing required had to increase in line with those probable uses of those methods and attributes.

### 7.3 Inheritance

How helpful was inheritance in aiding the development, given that the X-MAN component model does not support it?

Both inheritance and composition offer different methods of reusing functionality.

Functionality reuse with inheritance is easily achieved by simply extending functionality from a superclass to subclasses. However, because this is achieved statically at compile time, this also means that this form of reuse cannot be achieved again at run-time (Gamma et al., 1995, pp. 18-20). It also means that subclasses are dependent upon the superclass (Rajlich, 1998) and that inheritance is said to break encapsulation (Synder, 1986).

The advantage of object composition is that functionality can be reused and varied dynamically at run time. This is achieved through a well-defined interface that decouples implementations. However, as a result of this dynamism, it is much harder to determine the behaviour of the system until it can be fully apparent at run-time (Rajlich, 1998). Object composition is said to have the result of fewer and smaller classes and reduced class hierarchies (Gamma et al., 1995).
In the object-oriented re-implementation, the restrictions of the X-MAN component model have been removed, and inheritance is available and used in the design and was shown to be useful for pure code reuse.

The commonality of two different types of related sensors, aircraft sensor and wheel sensor, have been abstracted into a generic Sensor. The abstracted functionality is reading in values from a text file. This means that should a new specific sensor be required, it could simply be added as a new subclass. This would reuse the code contained in the abstracted Sensor. This is what is known as the Protected Variations principle in GRASP (Larman, 2005, p. 428).

The argument is that the same could be achieved using composition, by having a general sensor and the more specific sensor contained in separate components. Then the specific sensor would need to be composed with the general sensor, to reuse the code contained in the general sensor. However, any shared states would need to be passed as parameters between the components. It would also involve repeated compositions to achieve the same level of code reuse.

So although it is possible to use composition, it is not as intuitive as using inheritance. If composition were to replace the use of inheritance, it would be necessary to keep composing the general components with the specific components repetitively. In addition, the real power of inheritance is that it provides the right kind of conceptual structure and generality of types. The same operations are available uniformly because a subclass is a type of superclass.

Although inheritance was not directly used in the re-implemented parts of the original Nalmpantis (2011) ABS system, inheritance was used consistently to help support composition. Just as inheritance was used in the majority of design patterns, it was used in the GeneralFramework in order to support uniform interfaces and polymorphic operations, to allow behaviour to vary in the subclasses. It was used in the Composite Tree Builder to vary the builders (Figure 45, p. 118) and therefore the composites that are formed. It also shares common code and interfaces between the builders through abstracted composite builder detailed Appendix E (p. 115). Inheritance is also applied in
the ExtendedArrayList to extend functionality of an ArrayList. Finally, Figure 48 (Appendix G, p. 122) by Kang (2010, p. 27) shows a depiction of the X-MAN exogenous component model that uses inheritance and is remarkably similar to the GeneralFramework.

Inheritance is still necessary and complementary to composition. This is because “the set of components is never quite rich enough” (Gamma et al., 1995, p. 20), meaning that it is not always possible to assemble all the necessary functionality by simply composing components (Rajlich, 1998). Inheritance is used to create new components that can be composed with old components; thus, composition and inheritance work together.

Inheritance is enormously powerful on its own, but because of the simplicity with which you can quickly reuse functionality, inheritance on its own can easily be misused and misunderstood. However, it can bring many benefits and enhancements as shown by the work in this project, provided it is used correctly, as outlined in Subsection 2.2.5 (Rules For Inheritance, p. 25).
CONCLUSION

The aim of this project was to test the argument that the object paradigm and objects themselves are insufficient to solve the major issues facing the software industry. There has been a claim by component-based software advocates over the past two decades that composition solves many problems in software development, primarily in the reuse of pre-existing units of software.

The present project has shown that historic arguments are mostly a result of the misuse and misunderstanding of objects at a time when the comprehension of the object paradigm was in its infancy, particularly in regard to the use of inheritance. The use of best practice and design had been sporadic and yet needed to be formalized. Instead, it can be argued that objects already have most of the advantages of components and, given good practice via the application of generally accepted principles and patterns, the object-oriented approach can be enhanced to emulate the features that components are claimed to have.

However, component researchers took a survey from component experts to extrapolate which are the most desirable features for component-based software. This has resulted in the idealized component lifecycle that was used to distinguish what a component is from other paradigms and technologies. A new component model called X-MAN was developed in order to meet these new aspirations.

In order to demonstrate uses of component claims, M.Sc. students have developed various applications using the X-MAN tool. In particular, Nalmpantis (2011) implemented an anti-lock braking system (ABS) to be used by aircraft during the landing phase of a journey. The system was developed
in the C-based version of the X-MAN tool, which is the embodiment of both the X-MAN component model, and idealised component life cycle.

The Nalmpantis (2011) project shows that the XMAN component model and idealized life cycle code enables reuse by the composition of pre-existing components through the use of exogenous connectors. Further, in his dissertation entitled *Component-based Software For The Avionics Domains*, he explains that existing programming languages and other models are fundamentally not able to provide the same, specifically the object-oriented paradigm (Nalmpantis, 2011, pp. 8, 13, 34-35).

The approach taken in the current project was to re-implement the existing Nalmpantis (2011) ABS system in a number of ways using Java, in order to compare and contrast the object and component approaches.

To start with, the original system was successfully re-implemented twice. To achieve both of these, a General Framework and the Composite Tree Builder were developed to support the composition and form of the X-MAN hierarchies. Then, a simpler version was created, trading off data passing for component duplication that was easier to understand. The system was then re-implemented once more using an object-oriented approach and all re-implementations have been shown to produce the same results in the various braking simulations.

From the knowledge gained from implementing the X-MAN-like and OO re-implementations in Java, the current project evaluated the relative strength of both component and object arguments and approaches. It has compared and contrasted the two methodologies, and explored whether there were any benefits in the two methods.

The work has discussed some of the historic limitations to the X-MAN component model, including limiting atomic component reuse, ambiguous data passing mechanisms, structural complexity, the issue of component states and the unsupported looping mechanism. However, X-MAN is an ongoing research endeavour and these issues might be addressed in the future.
Although C-based version of X-MAN does decouple computation from control through the use of exogenous connectors, what has also been demonstrated is that coupling exists in the connectors as the reference parameter. This data passing mechanism is direct, implicit and extrinsic. Further, it is an important contradiction to point out and needs to be clarified because it is claimed that the X-MAN component model has no coupling in literature (K. K. Lau, Nordin, & Ng, 2011), and yet without some form of coupling the systems would simply not be able to function.

The project has shown that use of objects can emulate many aspects of the X-MAN component model, however it has also shown some of the weaknesses of the current implementation with regards to the idealized component life cycle. However, a tool could be created that supports the missing component repository and deployment phases. This tool could include greater optimizations in the data passing mechanism and a higher degree of modularization in the connectors. It would also help to understand if there is any real benefit in the application of the life cycle.

This project has established the importance generally of the accepted GRASP principles and design patterns and their continued relevance, not only in object-oriented design but also in regard to component-based concepts and technologies. This has also shown that good OO design is not only important to promote reuse, but also to aid understanding. The flexibility of using traditional message-passing and code reuse through inheritance does mean it is more likely that they might be misused and misunderstood, however the benefits of this flexibility in a clearer representation of graph structures and subclass typing outweigh this. As the X-MAN component model restricts and modularizes concepts into precise hierarchical structures, through this rigidity some intuitive relationships between the concepts are unclear. However, the C-based version of X-MAN is designed for particular applications regarding embedded systems of the automotive and aviation industries where these restrictions might be more beneficial.

Although this project has shown some tentative conclusions relating to the specific scenario, there are compounding factors, which imply that extrapolating further is problematic. This extends to the possibility that the
original Nalmpantis (2011) implementation might not be an exemplar of the best practices and uses of X-MAN. Further matters are accounted for next.

8.1 Future Work

This section will briefly outline how the work so far completed could be developed to provide a more widely applicable picture.

8.1.1 Reusability

The idealized component life cycle and X-MAN component model are said to promote reuse. The PID composite has applications in many domains other than braking; therefore, it is believed to be a generic, self-contained and reusable component. It would be interesting to test this assertion and the idea of reusability of the PID composite. Further exploration might include the possibility of replacing the PID with another form of controller, or removing and replacing parts of the PID to form variants of the controller, including the one-term P controller or two-term PI controller (Warwick, 1989, pp. 306-328).

8.1.2 HashMap

There were some limited benefits in using the ExtendedArrayList in terms of understandability and debugging. However, optimization and efficiencies could be found in either the use of a pure HashMap or an ArrayList. Essentially, the ExtendedArrayList could be thought of as both. In order to get the benefits of both, the HashMap and ArrayList could be used as part of a tool, which could switch between using the HashMap for development work and testing. Once testing is complete, the tool could switch to using an ArrayList when the system is used as an actual application.

8.1.3 Increased Modularity

The use of the single, system-wide ExtendedArrayList results in increased coupling between all the components that use it. Instead, the CompositeComponent of the GeneralFramework could be developed to instantiate a new HashMap and ArrayList. This could then be used to pass
data around each of the components that the composite encapsulates, in a more modular approach. That way, the direct coupling between components will be localised to components that are directly connected within that composite.

8.1.4 Different Metrics

The metrics used to assess the cohesion and coupling between the X-MAN-like and OO approaches are ambiguous. It does not necessarily consider some of the implications, such as the use of the ExtendedArrayList and the Message class. Therefore, the metrics could be distorted and do not necessarily give a true impression of the differences between the two approaches. Other metrics could be used to see if they present a clearer distinction. Object-oriented (and aspect-oriented) metrics for measuring external attributes like reusability are still an active and on-going research area.

8.1.5 Trade Offs

Nalmpantis (2011) justified the complex system organization of the final implementation as a means for redundant data to be passed between components. This has later been described as a trade off between: (1) increased depth and repetition of operations and (2) the increased complexity in component interfaces due to additional data being passed between components. For simplicity, it was decided to prioritise the latter. However, it would be interesting to find out, if it is possible to find the relative cost and whether there was a tangible benefit in taking either approach.

8.1.6 Tool To Support The Idealised Component Life Cycle

Some necessary aspects of the X-MAN tool have been imitated, including supporting composition and forming the tree structures in the builder phases. However, a more visually complete tool and technology could be developed to support and understand the complete idealized component lifecycles. This includes the ability to create a repository to store pre-built components and the assembler view to support specific system development in the
deployment phase. This could be achieved using reflection to dynamically load classes into the tool (Sosnosk, 2003).

8.1.7 Wider Applicability

The work conducted here could be extended to other applications to see if it is more generally applicable. This includes more exemplars such as the work by Kang (2010) and Choong (2012), who similarly to Nalmpantis (2011) used X-MAN to solve certain avionics and automotive problems in the engineering domain.

8.1.8 Broader Questions

It would not be appropriate to draw more general conclusions from this one study, but a wider range of studies could attempt to answer the following questions:

1) In what circumstances is a component lifecycle really ideal, or do the constraints that are imposed outweigh the benefits? Does the existence of class libraries negate the requirement for the existence of the component lifecycle?

2) It is possible to design an XML structure that represents an arbitrary object structure. This would then allow the storing of the objects in a component repository of the idealized component life cycle as XML documents. However, this would face the issue of schema evolution. But more fundamentally, is it necessary to store computation in this way? Or is the storage of classes and associated data required for instantiating those classes enough in normal circumstances?

3) What are the trade-offs between reusability as in X-MAN and design principles such as High Cohesion?

4) Does the restriction of the X-MAN component model cause unnecessary levels of code duplication?
Appendices

Appendix A

```java
public class Sequencer <K,V> extends CompositionConnector <K,V> {
    
    public void invoke(ExtendedArrayList <K,V> valueList) {
        
        ExtendedArrayList <K,V> intermediaryList = new ExtendedArrayList <K,V> ();
        ExtendedArrayList <K,V> returnList = null;
        
        for(XMANPart <K,V> part: composedParts) {
            returnList = new ExtendedArrayList <K,V> ();
            returnList.addExtendedArrayList(valueList);
            part.checkExtendedArrayList(returnList);
            part.invoke(returnList);
            intermediaryList.addExtendedArrayList(returnList);
        }
        valueList.addExtendedArrayList(intermediaryList);
    }
}
```

Table 8 The `invoke` method of the Sequencer class.

Table 8 shows the Sequencer is differently implemented because the outputs of the components are not channelled as the input of the next component. To achieve this, `valueList` is not passed as an argument to the `invoke` method of the composed parts. Instead the keys and values from `valueList` are copied to a new `ExtendedArrayList` called `returnList`, and that is then passed to the composed parts. Once the values of `returnList` have been mutated, they are copied into an `intermediaryList`. The values from `valueList` are copied to a new instance of `returnList`. This process is repeated for all the elements in `composedParts`. Once all elements have been iterated through, then the values contained in `intermediaryList` are finally replicated back into `valueList`. This is an inelegant solution to mimicking the behaviour of a sequencer, and
an artefact and consequence of using the ExtendedArrayList as a data passing mechanism.

Appendix B

========== Presets
Added: < antiSkid, 1.0 >
Added: < failure, 0.0 >
Added: < lowMedMax, 2.0 >
Added: < deltaT, 100.0 >

========== Test 6 - 2 "time = 5 sec"
Invoke: GeneralFramework.CompositeComponent
Invoke: GeneralFramework.Pipe
Invoke: Interface.CautionLight
Added: < cautionLight, 0.0 >
Invoke: GeneralFramework.CompositeComponent
Invoke: GeneralFramework.Pipe
Invoke: GeneralFramework.Sequence2
Invoke: Interface.Sensor
Read: < test/Interface/touchDown0Value1.txt >
Added: < touchDown0Value1, 0.0 >
Added: < (addAll) touchDown0Value1, 0.0 >
Invoke: Interface.Sensor
Read: < test/Interface/touchDown0Value2.txt >
Added: < touchDown0Value2, 0.0 >
Added: < (addAll) touchDown0Value2, 0.0 >
Invoke: Interface.CheckTouchDownSensor
Added: < touchDown, 0.0 >
Invoke: GeneralFramework.CompositeComponent
Invoke: GeneralFramework.Pipe
Invoke: GeneralFramework.Sequence2
Invoke: Interface.Sensor
Read: < test/Interface/wheelSpeedValue1.txt >
Added: < wheelSpeedValue1, 42.0 >
Added: < (addAll) wheelSpeedValue1, 42.0 >
Invoke: Interface.Sensor
Read: < test/Interface/wheelSpeedValue2.txt >
Added: < wheelSpeedValue2, 42.0 >
Added: < (addAll) wheelSpeedValue2, 42.0 >
Invoke: Interface.CheckFloatSensor
Added: < wheelSpeed, 42.0 >
Invoke: GeneralFramework.CompositeComponent
Invoke: GeneralFramework.Pipe
Invoke: GeneralFramework.Sequence2
Invoke: Interface.Sensor
Read: < test/Interface/aircraftSpeedValue1.txt >
Added: < aircraftSpeedValue1, 47.0 >
Added: < (addAll) aircraftSpeedValue1, 47.0 >
Invoke: Interface.Sensor
Read: < test/Interface/aircraftSpeedValue2.txt >
Added: < aircraftSpeedValue2, 47.0 >
Added: < (addAll) aircraftSpeedValue2, 47.0 >
Invoke: Interface.CheckFloatSensor
Added: < aircraftSpeed, 47.0 >
Invoke: Functional.LowMedMax
Added: < referenceSpeed, 40.89 >
Invoke: Functional.AntiLock
Added: < antiLock, 0.0 >
Invoke: Functional.PID.Error
Added: < error, -1.1100006 >
Invoke: Functional.PID.P
Added: < p, -0.32190016 >
Invoke: Functional.PID.I
Added: < i, -0.11100007 >
Invoke: Functional.PID.D
Added: < d, -0.003441002 >
Invoke: Functional.PID.ADD3
Added: < pid, -0.43634123 >
Invoke: Simulation.Wheel
Added: < result, 41.56366 >

=============== Results
Print: [0] < antiSkid, 1.0 >
Print: [1] < failure, 0.0 >
Print: [2] < lowMedMax, 2.0 >
Print: [3] < deltaT, 100.0 >
Print: [4] < cautionLight, 0.0 >
Print: [5] < touchDown0Value1, 0.0 >
Print: [6] < touchDown0Value2, 0.0 >
Print: [7] < touchDown, 0.0 >
Print: [8] < wheelSpeedValue1, 42.0 >
Print: [9] < wheelSpeedValue2, 42.0 >
Print: [10] < wheelSpeed, 42.0 >
Print: [11] < aircraftSpeedValue1, 47.0 >
Print: [12] < aircraftSpeedValue2, 47.0 >
Print: [13] < aircraftSpeed, 47.0 >
Print: [14] < referenceSpeed, 40.89 >
Print: [15] < antiLock, 0.0 >
Print: [16] < error, -1.1100006 >
Print: [17] < p, -0.32190016 >
Print: [18] < i, -0.11100007 >
Print: [19] < d, -0.003441002 >
Print: [20] < pid, -0.43634123 >
Print: [21] < result, 41.56366 >

Table 9 An output of the Message class for a simple test using the X-MAN re-implementation in Java.
Appendix C

========== Test 6 - 2 "time = 5 sec"

Invoke: OOImp.ControlUnit
Invoke: OOImp.Cockpit
   Calc: [ antiSkid(1.0) || failure(0.0); therefore, cautionLight = 0f ]
Invoke: OOImp.Wheel Touch Down
Invoke: OOImp.Wheel Sensor
   Read: < test/touchDownValue1.txt, 0.0 >
Invoke: OOImp.Wheel Sensor
   Read: < test/touchDownValue2.txt, 0.0 >
Calc: [ touchDownValue1(0.0) && touchDownValue2(0.0); therefore, touchDown = 0f ]
Invoke: OOImp.Wheel Speed
Invoke: OOImp.Wheel Sensor
   Read: < test/wheelSpeedValue1.txt, 42.0 >
Invoke: OOImp.Wheel Sensor
   Read: < test/wheelSpeedValue2.txt, 42.0 >
Calc: [ averagedSensorsValue(42.0) = (sensorValue1(42.0) + sensorValue2(42.0)) / 2 ]
Invoke: OOImp.Aircraft
Invoke: OOImp.Aircraft Sensor
   Read: < test/aircraftSpeedValue1.txt, 47.0 >
Invoke: OOImp.Aircraft Sensor
   Read: < test/aircraftSpeedValue2.txt, 47.0 >
Calc: [ averagedSensorsValue(47.0) = (sensorValue1(47.0) + sensorValue2(47.0)) / 2 ]
Calc: [ wheelSpeed(42.0) > (0.3 * aircraftSpeed(47.0) or (14.1)); therefore, antiLock = 0f ]
Invoke: OOImp.PID
Calc: [ error(-1.1100006) = referenceSpeed(40.89) - wheelSpeed(42.0) ]
Calc: [ p(-0.32190016) = kP(0.29) * error(-1.1100006) ]
Calc: [ newIntegral(-111.00006) = oldIntegral(0.0) + (error(-1.1100006) * deltaT(100.0)) ]
Calc: [ i(-0.11100007) = kI(0.0010) * integral(-111.00006) ]
Calc: [ derivative(-0.011100006) = (error(-1.1100006) - preError(0.0)) / deltaT(100.0) ]
Calc: [ d(-0.003441002) = kD(0.31) * derivative(-0.011100006) ]
Calc: [ pid(-0.43634123) = p(-0.32190016) + i(-0.11100007) + d(-0.003441002) ]
Calc: [ result(41.56366) = wheelSpeed(42.0) + pid(-0.43634123) ]

Table 10 An output of the Message class for a simple test using the object-oriented re-implementation in Java.
Figure 42 The re-implementation of the final Nalmpantis (2011) structure, with more repetition of operations and a deeper, more complex tree structure.
Appendix E

Composite Tree Builders

The following discusses the Builder pattern (Gamma et al., 1995, p. 97) used to build the simplified version of X-MAN-like re-implementation shown in Subsection 5.2.4.5 (Building The Tree Structures, p. 77). The builder pattern was also used to create the more complex version that was closer to the original implementation shown in Figure 42 (Appendix D, p. 114). The complex version will not be discussed this this document to be concise, but the underlying approach is the same.

Composite Builder

![Diagram of Director, Abstracted CompositeBuilder and Derived ConcreteCompositeBuilders]

**Figure 43** The Director, the abstracted CompositeBuilder and the derived ConcreteCompositeBuilders.

**Figure 43** shows the overall implementation of the builder pattern for the currently re-implementation, which includes of three parts:

1) CompositeBuilder: This abstract class specifies the interface and functionality for all the derived builder subclasses. In the case of the current re-implementation, it specifies the interfaces to add specific atomic and composite components that are contained in an ArrayList called partsToCompose. They are then composed with the Pipe and Sequencer composition connectors through the composeWithPipe and
composeWithSequencer interfaces. The ability to nest composition connectors by adding one composition connector to the partsToCompose ArrayList of another is also available via addConnectorToPartsToCompose interface. Finally, there the getComposite interface that allows the creation and the retrieval of a CompositeComponent object reference.

2) Directors: A director class makes use of the general interfaces of the CompositeBuilder, but also in order to construct composites it also makes use of specific interfaces of ConcreteBuilders. In the original pattern, there is one director that builds differing complex objects by varying the concrete builders. However, in this version, there are different directors, denoted DirectorN, which build use different ConcreteBuilders to build different structures. So therefore it is more correct to say that this is a variation of the GoF Builder pattern (Gamma et al., 1995, p. 97).

3) ConcreteBuilders: The set of specific builders, denoted ConcreteCompositeBuilder1 … ConcreteCompositeBuilderN, that each specify interfaces and functionality that is individual to the composite component that they create. Such composites include the Interface (Touchdown, Wheelspeed and AircraftSpeed), LowMedMax and Simulation composites shown in Figure 35 (p. 77) in Subsection 5.2.4.3 (PID Components, p. 74). The next few subsections will discuss each composite builder and the directors that make use of them.
Interface Composite Builder

Figure 44 The UML class diagram of the interface director and concrete builders that various interface composites.

Figure 44 shows the interface director and composite builders for the interface composites. Specifically, the builders include:

1) InterfaceCompositeBuilder: This abstract class specifies the interface and functionality for all the derived interface builder subclasses for the current re-implementation. This includes the non-specific functionality to create the interface composite tree structures including two sensors, a measurement checker and a sequencer and pipe composition connector that composes them. The type of sensor varies according to the instance variable key provided to the objects constructor, which defines the name text file that the sensor reads. The type of sensor checker varies depending on the specific subclass derived from InterfaceCompositeBuilder. These are discussed next.

   a. SpeedInterfaceCompositeBuilder: This concrete implementation creates Interface composites that specifically require the FloatChecker atomic component described in Section 5.2.4.1 (Interface Components, p. 72). For the current re-implementation, this includes the two sets of aircraft and wheel speed sensors. These composites are varied according to the String key that is passed to it by the director at run-time. An illustration of how this is done is shown by Figure 45 (p. 118) and in Table 11 (p. 118).
b. TouchDownCompositeBuilder: In a similar fashion to the SpeedInterfaceCompositeBuilder, this builder provides a specific interface to create an interface composite with a CheckTouchDownSensor atomic component, also described in Section 5.2.4.1 (Interface Components, p. 72). Figure 45 and in Table 11 also clarifies this.

Figure 45 The creation of the three different Interface composites using the same Director, by varying the key name and the builders used.

Table 11 A code example of how the interface composites are created, including: 1) aircraftSpeed, 2) wheelSpeed and 3) touchDown.

```java
1) new InterfaceCompositeDirector.build(new SpeedSensorsInterfaceCompositeBuilder("aircraftSpeed"));
2) new InterfaceCompositeDirector.build(new SpeedSensorsInterfaceCompositeBuilder("wheelSpeed"));
3) new InterfaceCompositeDirector.build(new TouchDownInterfaceCompositeBuilder("touchDown"));
```
**LowMedMax Composite Builder**

![UML Diagram](image)

**Figure 46** The UML class diagram of the director and concrete builder that creates the LowMedMax composite.

**Figure 46** shows the director and concrete composite builder for the LowMedMax composite shown in **Figure 35** (p. 77). Through the interfaces specified by the LowMedMaxBuilder, the LowMedMaxDirector is able to create the composite by combining an AircraftSpeed composite (created using an InterfaceCompositeBuilder described previously) with a LowMedMax atomic component (described in Section 5.2.4.2, p. 73).

**Simple Implementation Builder**

![UML Diagram](image)

**Figure 47** The UML class diagram of the director and concrete builder that create the Simulation composite.

**Figure 47** shows the implementation director and concrete composite builder. The SimpleImplementationDirector accepts a HashMap called setup that is
used to configure all the variable names and values used by current implementation. This allows the data to vary between simulations. Then, through the interfaces specified by the SimpleImplementationBuilder, the director is able to create the composite by combing the three composites in the previous two sections (TouchDown, WheelSpeed, LowMedMax) and the remaining atomic components, including: CaughtionLight, AntiLock, ErrorCal, P, I, D, ADD3 and Wheel. The resulting composite that is returned is the final Simulation composite of the re-implementation shown in Simulation composite in Figure 35 (p. 77). This final composite was used in the various ABS simulations in Chapter 6 (Simulations And Metrics, p. 80).
Table 12. The reproduction of results of the tests for the XMAN-like and OO Java re-implementations with the original Nalmpantis (2011) C-based version.

<table>
<thead>
<tr>
<th>Sensor Inputs</th>
<th>Test 1 &quot;failure in air&quot;</th>
<th>Test 2 &quot;failure on ground&quot;</th>
<th>Test 3 &quot;antiskid off&quot;</th>
<th>Test 4 &quot;touchdown protection&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiSkid</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Failure</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Low Med Max</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
</tr>
<tr>
<td>Delta T</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Touch Down</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Wheel Speed 1</td>
<td>20</td>
<td>60</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Wheel Speed 2</td>
<td>20</td>
<td>60</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Aircraft Speed 1</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>Aircraft Speed 2</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>Test 5 - 1 &quot;antiblock protection&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiSkid</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Failure</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Low Med Max</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
<td>2 (87%)</td>
</tr>
<tr>
<td>Delta T</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Touch Down</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wheel Speed 1</td>
<td>17.3</td>
<td>17</td>
<td>43</td>
<td>42</td>
</tr>
<tr>
<td>Wheel Speed 2</td>
<td>17.3</td>
<td>17</td>
<td>43</td>
<td>42</td>
</tr>
<tr>
<td>Aircraft Speed 1</td>
<td>60</td>
<td>60</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>Aircraft Speed 2</td>
<td>60</td>
<td>60</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>Result</td>
<td>XMAN C-based</td>
<td>XMAN Java</td>
<td>OO Java</td>
<td>XMAN C-based</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Appendix F
Appendix G

Figure 48 The exogenous connectors component model for X-MAN (Kang, 2010, p. 27).
REFERENCES


