Formal Methods for an Agile Software Development Methodology

by

Fisokuhle Hopewell Nyembe

Student number: 37233858

Submitted in fulfilment of the requirements for the degree of

Master of Science

in

Computing

at the

School of Computing

College of Science, Engineering and Technology

Supervisor: Prof. John Andrew van der Poll

Co-Supervisor: Prof. Hugo H. Lotriet

December 2022

Declaration

I, Fisokuhle Hopewell Nyembe, declare that this research work entitled ‘Formal Methods for Agile Software Development Methodology’ has been composed by myself. All the work included in this paper is my own, except where otherwise indicated. This work has not been submitted for any other degree or professional qualification.

[illegible]

Fisokuhle Hopewell Nyembe

December 2022

Acknowledgments

I would like to thank Prof John Andrew (André) van der Poll for his unending support and supervision through this daunting work which required his encouragement, insight, and guidance for years. Lastly, I would like to acknowledge the knowledge provided by Prof. Hugo H Lotriet who also joined us halfway in the process of completing this research work.

Dedication

This work is dedicated to my late grandfather and brother, Fakazi and Collen.

Table of Contents

List of Figures

List of Schemas

List of Tables

Acronyms and Keywords

Table 0-1: Acronyms and Keywords

Acronyms	Short Descriptions
Agile Software Development (ASD)	A software development model that combines iterative and incremental processes
DevOps (Development and Operations)	It's a software development methodology that integrates IT operations
Established Strategy (ES)	A strategy for drawing up Z specification
Formal Methods (FM)	Mathematical techniques used to verify software system's properties
IKIWISI	I'll Know It When I See It
PMI®	Project Management Institution
RAD	Rapid Application Development
SAFe	Scaled Agile Framework
Scrum	An Agile framework with emphasis on managing software development task in time-boxed iterations
SDLC	Software Development Lifecycle
Semi-Formal Methods	Object-oriented design using UML diagram that describes the structure of a system
UML	Unified Modelling Language
SPLE	Software Product Line Engineering
URDAD	Use Case, Responsibility Driven Analysis and Design
Waterfall Software Development (WSD)	A linear-sequence model for software development

Abstract

Business software systems are in high demand, which has led to the availability of a wide range of competitive market solutions. These ranges are also fuelling parallel demand for effectiveness and high-quality business software systems. To achieve efficiency, dependable methods ought to be adopted. One effective technique that has arguably excelled above others is the Agile Software Development Methodology (ASDM). Agile approaches possess the capacity to produce software in a way that is flexible to changes, making them, arguably, a preferred method for software development. Scrum, a recommended Agile methodology, emphasises the prioritisation of feature coverage and incremental project structures. Because iterative methodologies encourage engagement from cross-functional teams, including consumers, Agile provides flexibility in responding to changes in user requirements.

Despite its fast turnaround time, Agile may lack certain qualities, amongst other, producing software that is provably correct, as offered through the use of formal methods (FMs) for software development. Formal Methods usually embody the use of discrete mathematics and logic to develop highly dependable software. Using a case study approach the researcher investigates the processes embedded in the Scrum methodology by tracing the processes involved in the day-to-day operations of a Scrum team. Possible ambiguities and omissions in the processes and deliverables are identified and it is investigated to what extent the use of FMs using the Z specification language may address these. Aspects considered include formal specifications of the Scrum user stories and the use of the sprint backlog board to trace the progress of the project. The value proposition of using an FMs approach is illustrated throughout and a framework for embedding FMs in Scrum is developed on the strength of the analyses. The findings are validated through a hypothetical case study.

Keywords: Agile Software Development Methodology (ASDM), DevOps, Formal Methods, Formal Specification, Framework, Scrum, User Stories, Z.

Okucashuniwe

Izinhlelo zesofthiwe yebhizinisi zidingeka kakhulu, okuholele ekutholakaleni ezinhlangeni ezibanzi zezixazululo zemakethe ezincintisanayo. Lezi zinhlanga ziphinde zibhebhethekise isidingo esifanayo sokusebenza ngempumelelo nezinhlelo zesofthiwe yebhizinisi esezingeni eliphezulu. Ukuze kuzuzwe ukusebenza kahle, kufanele kusetshenziswe izindlela ezithembekile. Indlela eyodwa esebenzayo eye yaphumelela ngaphezu kwamanye Indlela Yokuthuthukiswa Kwesofthiwe Esheshayo (ASDM). Izindlela ezisheshayo zinamandla okukhiqiza isofthiwe ngendlela evumelana nezimo ezinguqukweni ezizenza, ngokungangabazeki, zibe indlela ekhethwayo yokuthuthukiswa kwesofthiwe. Uhlaka lokuphathwa kwephrojekthi, indlela esheshayo ephakanyisiwe, igcizelela ukubeka phambili ukufakwa kwesici kanye nezakhiwo zephrojekthi ezikhulayo. Ngenxa yokuthi izindlela eziphindaphindayo zikhuthaza ukusebenzelana okuvela emaqenjini ahlukahlukene, okuhlanganisa nabathengi, indlela esheshayo inikeza ukuguquguquka ekuphenduleni izinguquko ezidingweni zabasebenzisi.

Naphezu kwesikhathi sayo sokushintsha ngokushesha, indlela esheshayo ingase intule izimfanelo ezithile, phakathi kokunye, ukukhiqiza isofthiwe elungile njengoba inikezwa ngokusebenzisa izindlela ezisemthethweni (ama-FM) zokuthuthukiswa kwesofthiwe. Izindlela Ezihlelekile ngokuvamile zihlanganisa ukusetshenziswa kwezibalo ezihlukene kanye nengqondo ukuze kuthuthukiswe isofthiwe ethembeke kakhulu. Ngokusebenzisa indlela yesifundo esiyisibonelo umcwaningi uphenya izinqubo ezishumekwe endleleni yohlaka lokuphathwa kwephrojekthi ngokulandela izinqubo ezithintekayo ekusebenzeni kwansuku zonke kweqembu lohlaka lokuphathwa kwephrojekthi. Okungacaci kahle okungaba khona kanye nokweqiwa ezinqubeni nokulethwayo kuyahlonzwa futhi kuyaphenywa ukuthi ukusetshenziswa kwama-FM kusetshenziswa ngulimi olucacisa inothi elisekelwe esifanekisweni kungase kubhekane nalokhu. Izici ezicatshangelwayo zihlanganisa ukucaciswa okusemthethweni kwezindaba zabasebenzisi bohlaka lokuphathwa kwephrojekthi kanye nokusetshenziswa kohlu lwezinto zomsebenzi iqembu lakho elihlela ukuziqedela ngesikhathi sephrojekthi ukulandelela inqubekelaphambili yephrojekthi. Isiphakamiso sokusungula ukusebenzisa indlela ye-FM sikhonjiswa kuyo yonke indawo futhi uhlaka lokushumeka ama-FM ohlakeni lokuphathwa kwephrojekthi luyathuthukiswa ngamandla okuhlaziya. Okutholakele kuqinisekiswa ngocwaningo lwe-sifundo sokucatshangelwa.

Amagama asemqoka:

Agile Software Development Methodology (ASDM)

Indlela esheshayo yokuthuthukiswa kweSofthiwe

DevOps

Indlela yokuthuthukiswa kwesofthiwe

Formal Methods

Izindlela ezisemthethweni

Formal Specification

Ukucaciswa Okusemthethweni

Framework

Uhlaka

Scrum

Uhlaka lokuphathwa kwephrojekthi

User Stories

Izindaba Zomsebenzisi

Z

inothi elisekelwe esifanekisweni

Opsomming

Daar is ’n groot aanvraag na besigheidsagtewarestelsels, en dit het die beskikbaarheid van ’n wye verskeidenheid mededingende markoplossings tot gevolg gehad. Hierdie reekse gee ook aanleiding tot parallelle vraag na doeltreffendheid en besigheidsagtewarestelsels van hoë gehalte. Om doelmatigheid te bewerkstellig, moet betroubare metodes in gebruik geneem word. Een doeltreffende tegniek wat stellig ander tegnieke oortref het, is Agile Software Development Methodology (ASDM). *Agile*-benaderings beskik oor die vermoë om sagteware te genereer op ’n manier wat aanpasbaar is by veranderinge – en dit maak bes moontlik van hierdie benaderings ’n voorkeurmetode vir sagteware-ontwikkeling. Scrum is ’n aanbevole Agile-metodologie wat die prioritisering van eienskap-insluiting en inkrementele projekstrukture beklemtoon. Omdat herhalende metodologieë betrokkenheid van kruisfunksionele spanne, insluitende verbruikers, aanmoedig, bied Agile buigsaamheid ten opsigte van reaksie op veranderinge in verbruikersbehoeftes.

Ten spyte van die vinnige omkeertyd daarvan, kan sekere eienskappe by Agile ontbreek – onder andere om sagteware te genereer wat bewysbaar korrek is soos wat dit aangebied word deur die gebruik van formele metodes (FM’s) vir sagteware-ontwikkeling. Formele metodes behels gewoonlik die gebruik van diskrete wiskunde en logika om hoogs betroubare sagteware te ontwikkel. Die navorser gebruik ’n gevallestudiebenadering om die prosesse te ondersoek wat in die Scrum-metodologie ingebed is, deur die prosesse wat by die dag-tot-dag-werksaamhede van ’n Scrum-span betrokke is, na te gaan. Moontlike dubbelsinnighede en weglagtings in die prosesse en lewerbares word geïdentifiseer en daar word ondersoek in watter mate die aanwending van FM’s wat die Z-spesifiseringstaal gebruik, dit kan oorbrug. Aspekte wat oorweeg word, sluit in formele spesifikasies van die Scrum-gebruikerstories en die gebruik van die Sprint-agterstandbord om die projekvordering na te gaan. Die waardeproposisie van die gebruik van ’n FM’s-benadering word deurgaans geïllustreer en ’n raamwerk vir die inbedding van FM’s in Scrum word ontwikkel op grond van die sterkte van die ontledings. Die bevindings word gevalideer deur ’n hipotetiese gevallestudie.

Sleutelwoorde: Agile Software Development Methodology (ASDM), DevOps, Formele Metodes, Formele Spesifikasie, Raamwerk, Scrum, User Stories (gebruikerstories), Z.

Chapter 1: Introduction

This is a dissertation on the use of Formal Methods (FMs) in an Agile methodology. Formal Methods have been shown to facilitate the production of highly dependable software yet is hard a software engineer to achieve the necessary competency level (Huisman et al., 2020). Agile on the other hand hastens the software development process, yet may lead to challenges (i.e., lack of planning, scope creep, budgeting), especially with respect to mission-critical software development (Moyo, 2021).

In this introductory chapter, the focus is on three main topics, namely, literature review, problem statement and the formulation of research questions. In the literature review, the purpose is to identify gaps and weaknesses in prior research work to discover areas that need development in the chosen research focus area. On the basis of the under-researched areas the research focus area for this dissertation will be delimited and discussed. For the identified research problem, a series of research questions were formulated.

The main purpose of the chapter is to demonstrate the importance of the research topic and the research objective and to argue for the appropriateness of the research design selections, including the research strategy and methodology to achieve mastery of the subject matter, ultimately with the aim to assist software developers achieve efficiency in their processes and quality in their output products.

1.1 Background

There is a significant growth in demand for business software systems, resulting in the availability of a wide variety of competitive market offerings. These software offerings are also resulting in a concurrent demand for efficiency and quality (Hussain et al., 2019). Adopting tried-and-true approaches is necessary in the quest for efficiency. Numerous software development approaches have been developed in response to the need for efficiency. The Agile Software Development Methodology has been one methodology that has risen above other methodologies (Holbeche, 2018). The major reason why Agile methods are seen as a preferred alternative is because of their ability to create software in a manner that is responsive to change (Holbeche, 2018). It is also considered to have the ability to balance flexibility and structure (Highsmith, 2003). A popular Agile technique called Scrum highlights the organisation of projects and feature coverage according to priority (Rush et al., 2020). This iterative methodology enables flexibility in reacting to any exigent circumstances by facilitating the participation of cross-functional teams, including the customer. According to Holbeche (2018), agility is no longer limited to software projects, but now forms part of overall business strategies.

Achieving methodological efficiency in the development of software is however not enough. Equal consideration has to be given to ensuring high-quality output (O'Regan, 2020). Using mathematically based techniques known as Formal Methods has the potential to provide the consistency, completeness and ultimately the quality of the software system (O'Regan, 2020). Formal Methods have traditionally been associated with the rigidity of traditional 'conveyor belt' development methodologies (Larsen et al., 2010). Therefore, this research aims at formulating a framework to blend the Agile Software Development and the Formal Methods for companies to keep up with the increasingly demanding software systems business.

1.2 Literature review

The literature review is intended to provide an overview of existing research related to the research focus area and to identify areas that are under-researched that need to be addressed as part of the current project. By initially coming up with the research problem, I was able to narrow the literature review in order to remain within context. In this literature review I focus primarily on the collection of scholarly material related to the research topic. The literature review assists in guiding the extent of the research problems, questions and objectives. In gathering the information, I realised that although there has been significant research on the Agile Software Development (ASD) and Formal Methods (FMs), a combination of the two techniques has been under-researched.

1.2.1 Agile Software Development

Below I discuss aspects that make up Agile Software Development. These topics have been presented with the intention of keeping us in-touch with what has been researched in the chosen subject. This methodology has the ability to create more value and rapidly respond to change. It is also attributed to have the ability to balance flexibility and structure (Highsmith, 2003). Owing to its widespread use, the research utilised the popular Agile technique called Scrum, which highlights the management of projects and features coverage according to priority and also because working software is the primary measure of progress (Rush et al., 2020). Scrum is also regarded as a development framework for delivering and maintenance of complex software systems. It is a conceptual framework that enables individuals to confront multiplex adaptive challenges while efficiently and innovatively producing goods of the best quality. It is characterised by piece-meal project cycles, known as “Sprints” that are usable for delivering planned, designed, built, and tested reviewed software systems (Hatcher, 2019).

1.2.1.1 *Requirements are often not upfront*

Contrary to the above definitions of a Waterfall software development where full system requirements are available upfront and enough time is allocated to planning processes prior to the development, the Agile Software Development (ASD) practice known as IKIWISI (I’ll Know It When I See It) implies that full system requirements may not always be available upfront. This approach also suggests that users can better describe their full requirements after the initial idea has been translated into a functioning Prototype (Szalvay, 2004). This is unlike the Waterfall Software Development (WSD) where customers are expected to thoroughly specify the desired system, usually without having an opportunity to periodically review the progress and request changes.

Figure 1.3 illustrates an Agile design with 3 iterations.

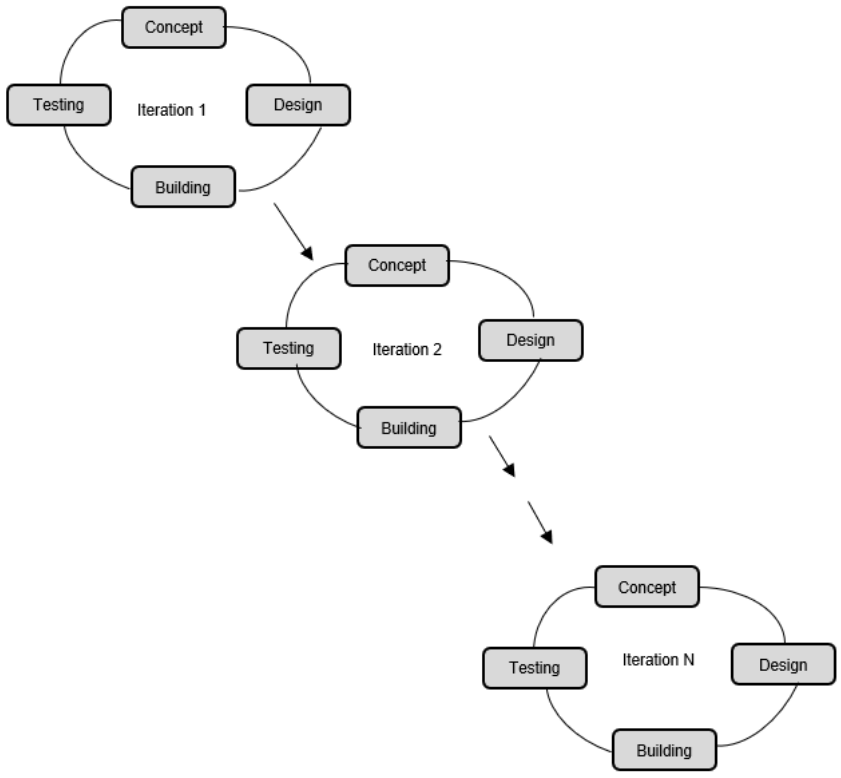


Figure 1.1 Agile Software Methodology (https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm)

The above figure is a representation of a 3-iteration Agile software build. Each iteration consists of the development of the concept, design, building and testing. All of these development tasks occur simultaneously, and this is mainly what distinguishes Agile Software Development from (e.g.) the Waterfall Software Development.

1.2.1.2 *The usage of User-Stories*

Tomayko (2017) suggests that there has been a significant movement towards iterative methodologies and Agile Software Development (ASD) in particular. The ASD uses *User-Stories* to interpret how the system should function. These User-Stories assist in advancing the end-user perspective on the system, and they are the beginning and end points of the requirements coverage. The User-Stories are a place to start and are simplified in a non-technical language. They are also continuously developed throughout the development as more becomes known about the software product.

1.2.1.3 *Responding better to change*

The ASD methodology responds better to changing requirements and, therefore, becomes suitable for fast paced environments (Tomayko, 2017). ASD enables habits such as process controlling, particularly when managing highly complex and ever-changing software requirements. Although Agile has simpler and clearly defined processes, it is (naturally) not the only answer to effectively manage every dynamic and complex software project. The ASD has become less structured in the recent years and has mostly been characterised by the three (3) faces of simplicity (Tomayko, 2017):

1. Minimalism
2. Quality design
3. Generative rules

1.2.1.4 *Requirements elicitation*

Arguably, the major difference between the traditional Waterfall Software Development and Agile Software Development is in how the system requirements are elicited. In comparison to the WSD, the main principles of ASD are to facilitate quicker delivery, quicker change and changes more often (Beedle et al., 2010). Contrary to the WSD, Agile has a lesser appetite for thorough requirements analysis (Franch et al., 2018). There are three main requirements analysis techniques in an Agile methodology, namely:

1. JAD (Joint Application Development): It involves a continuous, rigorous interaction amongst stakeholders. This technique forms a large part of Scrum which prioritizes individuals and interactions over processes and tools.

2. Modelling: It integrates the analysis of requirements and the designing. The modelling technique relates to the 11th principle of agility. This principle highlights the importance of best architecture, requirements and design. The 11th principle of agility is discussed in Section 2.2 in Chapter 2.

3. Prioritization: This involves the prioritization of the system feature. The Scrum method uses techniques with storyboards, which organise the project and feature coverage according to priority.

1.2.1.5 *ASD's minimal documentation*

One of the ASD's principles is that of minimal documentation, and this implies that no formal requirements are to be produced (Franch et al., 2018). In this instance, the features of the system become piece-meals that will lead developers into a fully functioning product required by the client. These features are interpreted in the form of User-Stories, are recorded on story boards and are tracked daily. Franch et al. (2018), acknowledge the challenges in the management of requirements in ASD – these difficulties are as a result of the pressure that comes with expectations of fast deployments.

ASD embody three main practices that are used to record features coverage:

1. Requirements / User-Stories reviews
2. Unit testing
3. Evolutionary prototyping

1.2.1.6 *The major Agile Software Development shortcomings*

With Agile, teams never know what the end result (or just a few cycles down the line) will look like from the very beginning; it is, therefore, difficult to estimate what it will cost, how long it might take, and which resources will be needed in the beginning (especially when the project gets larger and more complex) (Bhavsar et al., 2020).

The documentation in Agile projects happens continuously, and often "just in time" for the output, rather than from the beginning. In this manner, it becomes less detailed and is often put on the back burner (Hatcher, 2019). Agile methodology may be beneficial for bringing products to market faster, but it also has many drawbacks. Due to the fact that teams often work on each component in separate cycles, the finished product usually seems fragmented instead of unified (Hatcher, 2019).

It is relatively easy for Agile to get side-tracked by delivering unexpected features since it requires minimal planning at the beginning. Furthermore, it means that projects have no end because it is impossible to visualize what the "final product" will look like (Moyo, 2021). Considering Agile delivers in increments, tracking progress requires a broader perspective. As a result of the "see-as-you-go" nature, it is impossible to set many KPIs at the beginning of a project. Progress is difficult to measure this way (Bhavsar et al., 2020).

In this research paper, the goal is to formulate a framework that will reduce some of the above-mentioned challenges in the Agile Software Development Methodology by including Formal Methods. The following section introduces Formal Methods (FMs) which make up the second component of this research, Agile being the first component. Formal Methods are the use of mathematical-based techniques for improving on the integrity, consistency, and completeness of an information system (Schaefer et al., 2011).

1.2.2 Formal Methods

Formal Methods (FMs) use mathematical notation to detail the precision of the software systems' properties. Formal Methods at the starting point usually focus on formal specifications, which is a way to formally describe system requirements (Spivey, 1998). Below I describe different perspectives and approaches for Formal Methods.

1.2.2.1 *Formal Methods defined*

Formal Methods in software engineering use discrete mathematics and logic to develop a system. The FMs simply describe what the system must do, and not how it is to be achieved. A formal specification serves as a reliable reference point to verify the information system functions as determined by the customer. The system's properties ought not to unduly constrain the specification as to how the information systems' correctness is achieved (Spivey, 1998). One of the major advantages of specification formalism is overcoming limitations of resilience (Madni et al., 2018). A simple formal specification in the successful Z specification language of users logging onto and out of a system is given below as an example (Butler, 2001):

$$\begin{array}{|l}
\multicolumn{1}{l}{\_Log\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Users, in, out : \mathbb{P}\ Staff \\
\hline
in \cup out = users \wedge \\
in \cap out = \{\} \\
\hline
\end{array}$$

Z Schema 1.1 Formal Method's log in/out example (Butler, 2001)

The above example shows a state space with three sets of users in the system and two states of staff members:

- All *users* registered in the system.
- Registered users who are currently *in.*
- Registered users who are currently *out*.

There are two predicates below the short horizontal line. These state that:

- The union of all in and out users make up all the users in the system.
- No user should be both logged on and logged out.

The following mathematical notation are used in the schema:

- $\cap$ denotes a binary intersection,
- $=$ denotes simple set-theoretic equality,
- $\{ \}$ denotes an empty set,
- $\wedge$ denotes logical conjunction, and
- $\cup$ denotes a binary union.

1.2.2.2 FMs within a software development lifecycle

Synthesised from Dongmo (2016), Formal Methods could cover the following SDLC phases.

1. *Requirements elicitation phase*: Mathematical notation can be applied as part of thorough requirements engineering.
2. *Specification phase*: A formal specification could capture user requirements otherwise captured in natural language.
3. *Design phase*: System design conforms to a formal specification.
4. *Implementation and maintenance phases:* The verification of the implemented software should be continuous throughout the lifetime of the system. Aspects of quality and accuracy (correctness) are paramount.

1.2.2.3 FMs in the requirements elicitation phase

The requirements elicitation phase is regarded as being most crucial and most challenging. The consequences of getting this critical phase wrong are far-reaching and can persist throughout the life of the software system (Pandey et al., 2013). An adequate requirements analysis function exposes and predicts error prone areas in the proposed system (Pandey et al., 2013). Much of the research on Formal Methods has been around the requirements gathering phase. In most of these research, it is identified that the requirements gathering phase enables the formal specification to accurately validate the requirements (Pandey et al., 2013). The challenge in ASD therefore becomes continuously changing requirements, which adds to the complexity of the project.

When introspectively analysing issues arising from the software system development, one can identify that a significant portion of them come as a result of properly specifying requirements. The root causes of defects found in system testing are as a result of requirements being unclear, imprecise, incomplete and ambiguous. If the above specifications' shortcomings are not addressed, the purpose and objective of the testing process becomes narrow.

1.2.2.4 Are Formal Methods ready for Agile?

In all the efforts in attempting to use FMs in an Agile Software Development, what then is to be gained, particularly because of the many opposing differences between the two? (Nemathaga and van der Poll, 2019). In their work, Gleirscher et al. (2019) assess benefits that will come with combining FMs and ASD. They also assess the readiness of ASD to support FMs techniques in order to have synergy in the processes. Larsen et al. (2010) identify the purpose of the Formal Methods as that of eliminating defects in complex computer systems. They further describe FMs as a response to complexity. Such a response is used to analyse and model software systems as a mathematical entity. These mathematical analyses, therefore, enable every competent stakeholder to verify and refute aspects of the requirements specifications in all development phases. It is important to note that Larsen et al. (2010) dispel the widely held view of regarding Formal Methods as a software development methodology on their own.

A misreading that Larsen et al. (2010) deal with is that FMs are only effective as a post-factor verification. In their arguments, they also advise against viewing Agile Software Development as a methodology that can be implemented in all software development environments. Each software development enterprise should adopt only the ASD characteristics that are suitable for their environment and their resources (O'Regan, 2020). Similarly, with any methodology and processes, only the applicable techniques are adopted based on the environment and sometimes the product being developed.

In a similar vein, formalising requirements ought to be intended at simplifying the specifications, otherwise it will be irrelevant including them in ASD which aims at rapidly completing a solution with 'minimal documentation'. Introducing FMs should not be burdensome; forms of static analysis and automatic verification can be used to guard that key properties are preserved from one iteration to the next (O'Regan, 2020). The tools enabling FMs must also facilitate synergy in existing development methodology and enough research must be conducted in making this a reality, as Larsen (2010) continue to claim.

1.3 The research problem statement

With the wide technology exposure in recent decades, there has been a steep growth in the demand for business systems. As a result of the market demand, competitiveness has increased, and consumers have more market offerings to choose from. These various software outputs have also resulted in the rise in demand for an efficient methodology and techniques to develop them. Therefore, the existing problem in the markets is ensuring high standards of quality software output within an effective software development methodology. To achieve efficiency, dependable methodologies have to be adopted. As a result of the demand for dependable methodologies, various options have emerged in different eras of software development. ASD has been identified as the methodology that facilitates rapid development of software. However, this rapidity often leads to faulty software systems, particularly the security critical systems.

On the other hand, the use of FMs facilitate the development of provably correct software systems. However, FMs may be cumbersome to use, leading to perceived delayed delivery of software systems. Therefore, combining ASD and FMs could be beneficial, but there has been limited research in the attempt to have the two techniques complementing each other.

1.4 The research purpose, questions, and objectives

Given the research problem indicated above, the aim of this research is to develop a framework that combines the best of FMs and ASD, thereby assisting the software development industry to improve on software quality. Given the widespread use of Agile, and the limited use of FMs, I suspect a best practice would be to embed FMs as a component of Agile. Consequently, this research sought answers to the following questions:

1.4.1 Research questions

- **RQ1:** What are the advantages and disadvantages of:
    - **RQ1.1:** Agile software development?
    - **RQ1.2:** Using Formal Methods (FMs) for software development?

- **RQ2**: To what extent can Formal Methods be implemented in an Agile Software Development Methodology?
    - **RQ2.1**: At what developmental phases could FMs be embedded in an Agile development process?
    - **RQ2.2**: What will business enterprises achieve by embedding FMs in an Agile development process?

The major objective of the study was to formulate a framework to blend the Agile Software Development and the Formal Methods for companies to keep up with the increasing demand for quality and efficiency in software systems business. This objective was broken into the following five sub-objectives:

1.4.2 Research objectives

- Identify the advantages and disadvantages of Agile and FMs software development.
- Identify what business enterprises would achieve by merging Formal Methods into Agile Software Development Methodology.
- Determine whether Formal Methods can be implemented in both old software requirements (regression) and new software requirements which are rarely upfront in ASD.
- Determine for which Agile development phases it may be appropriate to implement FMs.
- Develop a framework for embedding FMs in an Agile methodology.

Having formulated the above research questions and objectives, I will discuss the research design. The research design is based on the Saunders et al. (2018) Research Onion, created as per figure 1.4 below.

1.5 The Research Layout

In the below figure, we present the dissertation structure. This layout will guide the execution of the research work in order to achieve what was proposed in module MPSET92 (Master Proposal - CSET).

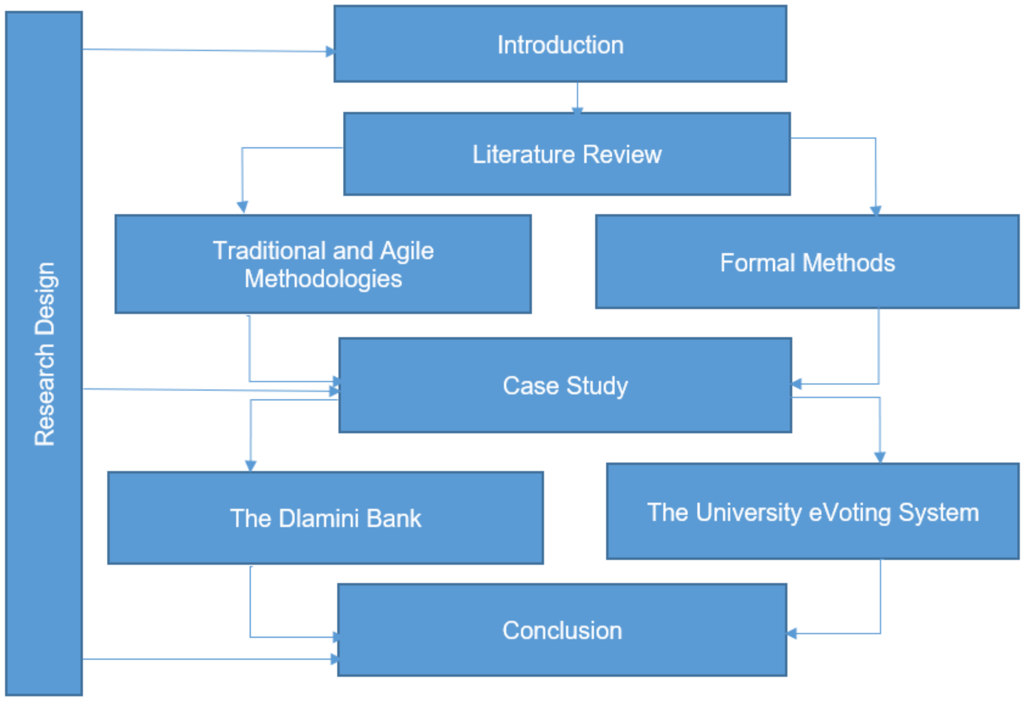


Figure 1.2 Dissertation Layout

The above Figure 1.2, I present the dissertation layout for how the research is conducted. The figure shows that the dissertation begins with the introduction of the research, then the literature review of software development methodologies and Formal Methods. The essence of the dissertation is in the presentation of the two hypothetical case studies where we introduce Formal Methods into an Agile Methodology framework called Scrum. Chapter 7 contains a conclusion, where I revisit all that is done in order present the sufficient coverage of problem and objectives of the dissertation as was proposed. The conclusion chapter also includes a presentation of what the study has contributed to the body of knowledge.

1.6 Summary

The work in this chapter began with the introduction of the dissertation, where I identified gaps in prior research works so that I could discover areas that needed development in the chosen subject matter. From those under researched areas the researcher was able to formulate the research problems on which the work is based and the potentiality of using Formal Methods in an Agile Software Development Methodology. This choice of the research topic was driven by intentions to contribute methodical and efficient ways to produce dependable software products.

In Chapter 1 the researcher also did a brief literature review of three software development methodologies, namely: Waterfall Software Development, V-Model and Agile Software Development which is the focal point of this dissertation. I was also able to discuss Formal Methods in the context of this research.

The research methodology framework of the dissertation was presented in the form of a research design based on Saunders et al.'s (2018) research process onion. In the work, I chose the case study research strategy. These choices were informed by intentions to know more about the uniqueness of the environment under the study and ultimately formulate recommendations in which FMs can be used in an ASD environment.

1.7 Conclusion

By large, success in software development remains an ever-shifting goalpost and the efforts to improve the methodical ways of achieving successful projects and quality software will continue well after this research. With all the research and methods in place, software development projects are still facing the same challenges that were faced decades ago. At most, their efficiency is not absolute, and the results are less qualitative software output. Although organisations that have been able to successfully use the Agile technique demonstrate the effectiveness of this methodology, the rapidity of the method emphasises less the quality of the final software product in comparison to the Waterfall Methodology (Tomayko, 2017). The body of research assembled in this research work shows that, in terms of both efficiency and quality problems, experiences are essentially universal. The attempt in this research work was to help find out if Formal Methods can be implemented in an Agile Software Development Methodology.

In the following chapter, the researcher explores relevant literature on numerous traditional software methodologies and puts a spotlight on the Agile Software Development methodology which is the topic of focus.

Chapter 2: Literature Review on Traditional and Agile Methodologies

In Chapter 1, I introduced the dissertation. I also laid the foundation for the work through a brief literature review where I paid a close look at the Waterfall Software Development and the V-Model Software Development, which are considered as traditional methodologies. I then reviewed Agile Software Development and the Formal Methods which are the focus of this dissertation. The introduction of the work is based on the problem statement that was presented and the purpose of the research is to find answers to the problem. The problem statement stated in Chapter 1 is a composition of the formulated research questions and therefore answering these questions became the objective of this dissertation.

In Chapter 2, I provide a comprehensive literature review on the traditional software development methodologies. I explore the Waterfall Software Development Methodology, which is a sequential process of software development and, just like in a Waterfall, different development phases cascades from one phase to another. I then review the Incremental Model which divides the product into builds, where sections of the project are created and tested separately. I also discuss prototyping as a methodology. It is where a throwaway Prototype is built from the initial customer requirements and subsequently presented back to the customer to confirm if this is indeed what they require. Another significant methodology is a Spiral model. Regarded as a methodology for high-risk projects, the Spiral Model combines the characteristics of both the Waterfall model and the Prototype model. Lastly, as part of the traditional software development methodology, I review the Rapid Application Development (RAD) which is a results-oriented development lifecycle designed to give much faster development.

In the second section of Chapter 2 I review the Agile Software Methodology (ASD). ASD is the popular solution that many are realising (Kim et al., 2021). It emphasises continuous interaction with the customer, as the Agile methodology is a throughput-focused method for providing value to customers as soon as feasible.

2.1 Traditional software development methodologies

Software development methodologies are intended to provide frameworks to plan, execute, and manage processes for the development of software systems (Akbar et al., 2017). There have been many methodologies adopted, including Waterfall, incremental, prototyping, spiral, structured, object-oriented, RAD, and Agile methodologies. Each one of these has its positives that are advocated and negatives which are criticized. The pursuit of improved methodologies is based on the attempt to achieve success and efficiency in project delivery (Akbar et al., 2017). At times, the selection of a methodology may be dependent on the marketing and research biases which support certain new or industry practices, while at other times organisations can depend on standards for consistency and repeatability (Akbar et al., 2017).

A broad overview of the history of software development reveals that the period up to and including the 1960s was referred to as the "functional era", the 1970s as the "schedule era", during which the Waterfall methodology was developed, the 1980s as the "cost era", and the 1990s and later as the "efficiency and quality era" (Akbar et al., 2017). This 1990s era came with many methodologies, including the Object-Oriented Software Development (Akbar et al., 2017). Because software development firms are heavily dependent on it due to the desirable results of software development approaches, software is becoming more and more desirable and a significant source of revenue for businesses.

The most important decisions are made in the beginning during the design stage and once the software system is well designed, the project then continues with the development phase and therefore becomes very predictable. In this case, the development stage of the software product follows what will be the “perfect” design of the system that was predicted in the initiation of the project. Because the software’s main feature is specified at the planning phase followed by the designing phase, planning is useful when the project is complex, and the level of risk possibly higher. Some projects that are estimated to take longer are often, even currently, still developed using traditional methodologies (Akbar et al., 2017).

The traditional methodologies of software development comprise of stages which means that prior to entering the next stage the earlier one must be completed. At the end of each stage prerequisite detail is acquired. That is the roadmap from both the side of the designer and customer; it prompts and enables the discussions of traditional software development methodologies that have created the present framework plans. In this literature review, I trace back the development of the different software methodologies through the years.

Tanzania is a developing economy where, similarly to RSA, their software development industry is crucial to unlocking the Fourth Industrial Revolution. Figure 2.1 shows the distribution of traditional software methodologies per software product developed in Tanzania.

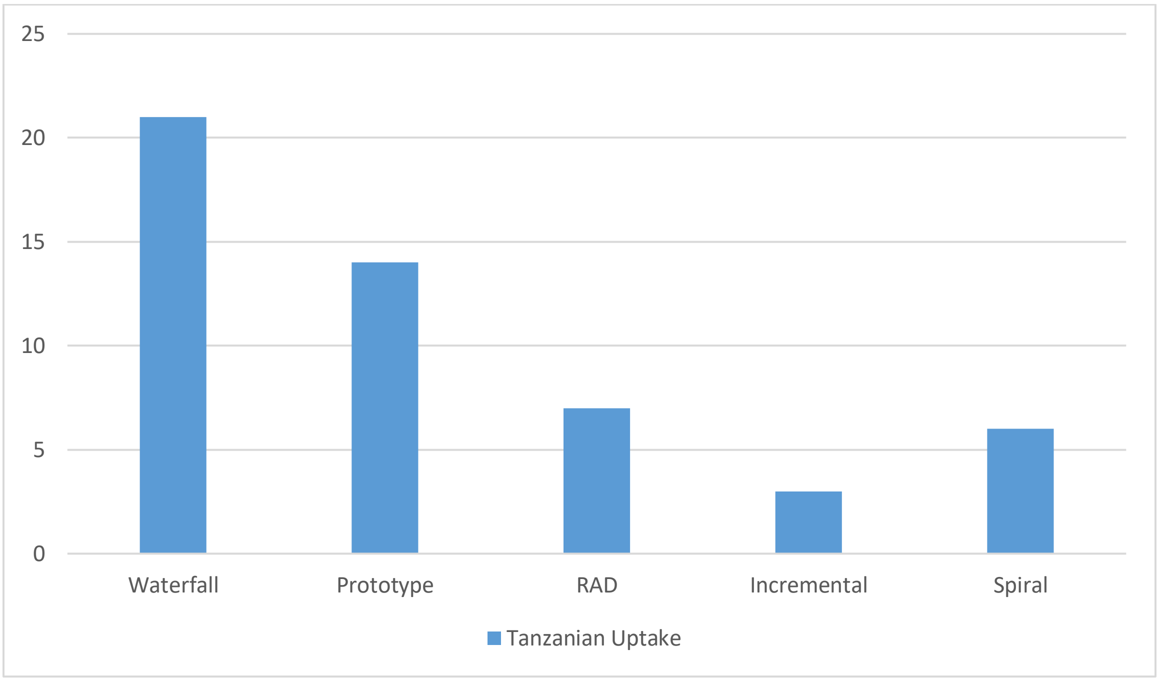


Figure 2.1 The distribution of traditional software methodologies in Tanzania (Mushashu et al (2019)

In a Tanzanian survey by Mushashu et al. (2019), they discovered that the Waterfall model was highest on the list of most adopted traditional software development methodologies totalling 21 software product outputs out of 51. “It is followed by the prototyping which had 14 software products of the 51, and 7 of the software products were developed using Rapid Application Development methodology” (Mushashu et al., 2019). Not a single one of the firms surveyed adopted the V-model development methodology, but they were familiar with it (Mushashu et al., 2019).

There are however contrasting findings in different parts of the world. A survey conducted by Akbar et al. (2017) shows a different picture than the one painted by Mushashu et al. (2019) above.

Figure 2.2 below presents the distribution of traditional software methodologies globally according to a survey conducted by Akbar et al. in 2017. The global economies surveyed are China, Pakistan, and Saudi Arabia.

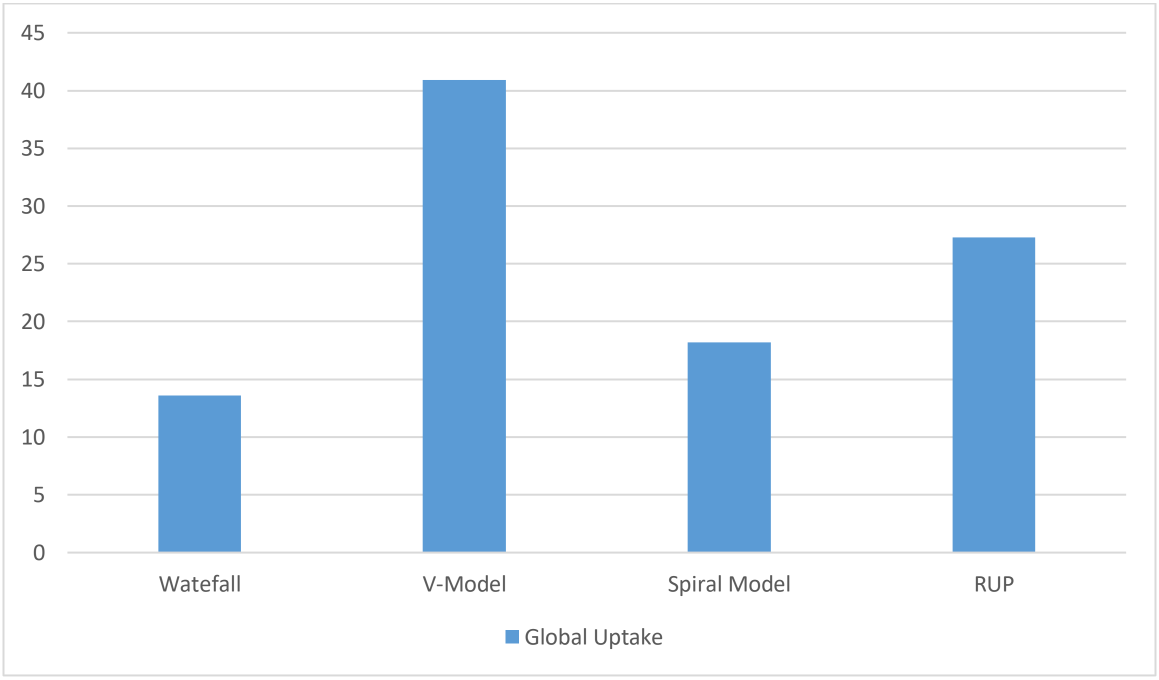


Figure 2.2 distribution of traditional software methodologies in Globally (Akbar et al., 2017)

In a world view, a survey by Akbar et al. (2017) shows that the V-model or V-shaped model is the most used traditional methodology with over 40% of uptake within the pool that was sampled. One may also bear in mind that the V-Model is an enhancement of a Waterfall methodology that adds parallel testing with every cascading phase of the software development (Wang et al., 2019). In the survey, they discovered that only 13.6% of the participants still use the Waterfall Methodology. At 27.3%, the second biggest uptake was the RUP (Rational Unified Process) which is based on the Unified Modelling Language and is accredited to IBM. It is followed by the Spiral model at 18.2%.

2.1.1 Waterfall Software Development Methodology

Waterfall Software Development Methodology's software engineering was an idea presented in a publication paper by Winston Royce in 1970. However, he cynically introduced it as a flawed software development method that has vulnerabilities which may include its inability to accurately predict and interpret the software testing (Royce, 1970). However, it is natural that every methodology will have its advocates and critics. The Waterfall methodology has had visible success which is evident through the sustained adoption and implementation by many software companies. The building and hardware manufacturing tactics that were in use in the 1970s can be linked to the concept behind the approaches. This background results in a very organised approach to software development (McCormick, 2012).

The Waterfall methodology, as the name itself signifies, is a sequential process of software development. Reconcilable with actual Waterfall where water falls from one height to another of lower latitude, the software methodology uses the same escarpment where one phase completes its role then cascades to another phase until a complete software product is produced. These cascading phases of software development in this methodology include requirement specification, system design, integration, testing, implementation and maintenance. Evidence shows that when software developing organisations adopt the Waterfall Model, they usually spend a significant amount of time and effort in each phase of the development to ensure that all the requirements for the phase are met. The basis of the software development model philosophy is that spending considerable amount of time in ensuring accuracy of the initial design is in a way of correcting bugs in advance. Once the design phase has been completed, it becomes easier to code and implement exactly what was designed and what the customer required without having to change (except for maintenance at a later stage). In most organisations, usually the analysis, design and coding teams are split and each team focuses on their phase in the fulfilment of the developmental process (McCormick, 2012).

The Waterfall methodology model emphasises documenting details for every step of all the phases during the development of the software products. Other phases including that of testing can become overburdened by the amount of documentation such as test plans which result in the phase having to have its own sub-phases that could be equivalent to the overall project management tasks. Having this kind of a stern process can be seen less progressive and inflexible in a world of rapid software development as the ‘fixed’ documentations will not allow added requirements and designs that may be forced by new market inventions (Mushashu et al., 2019).

This methodology can then be more ideal in the development of projects where the final product can be easily and completely predicted from initiation, where also the design will not need major and continuous makeovers during development. When the customer has supplied a detailed list of criteria that are unlikely to be changed, it is an obvious decision. Regardless of the flaws, mentioned above, Waterfall also has the potential of ascertaining development costs beforehand.

Figure 2.3 below, illustrates the cascading phases of a Waterfall Development Methodology.

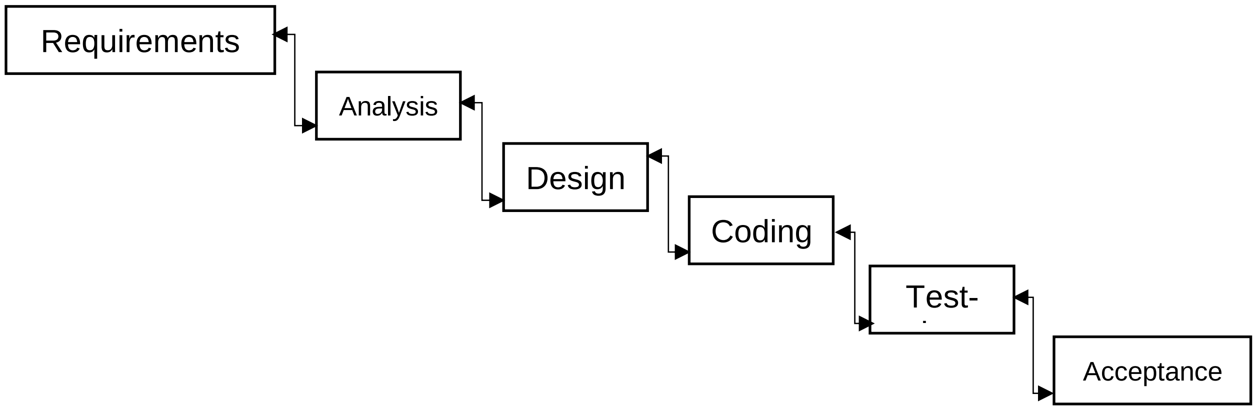


Figure 2.3 Waterfall Development Methodology (https://www.tutorialspoint.com)

The above figure represents the multiple phases that form a Waterfall Development Methodology. In this methodology, one phase has to be completed before the next phase can start. i.e., the requirements extraction phase has to be completed before specifications documents can be created and this cascade continues until the software system is promoted and ready to be used.

From Winston W. Royce’s first introduction of the Waterfall methodology in 1970, it became widely used in the field of software development. In the figure 2.3 above, the Waterfall Development Methodology is illustrated and explained. It is presented with a well-designed entity which outlines plans for the project. Once the structured planning is completed, the process becomes simpler and more effective, and this also applies to the rest of the phases. The methodology consists of algorithms or flowcharts, which are intended to plan out the functions that must be performed in order to complete one phase to another. In the software development process, different programme models are used in the planning of the various phases of developing software applications (Mushashu et al., 2019). One such model is the Waterfall model.

2.1.2 Incremental Model

The incremental model is a methodology that dissects work into small builds which are known as increments. Unlike the Waterfall methodology where the customer is presented with the developed software right at the end, in this approach the customer is presented with the developed work after each increment. This is to solicit feedback from customers sooner so that reworks don’t have to include everything. A similarity between the Waterfall and the incremental model is that the overall requirements are given and gathered in the beginning of the development process. In incremental models, however, the customer is enabled to update the requirements when each increment is presented until the last increment which concludes their satisfaction (Alshamrani et al., 2015).

Figure 2.4 indicates how user requirements are broken down into multiple increments.

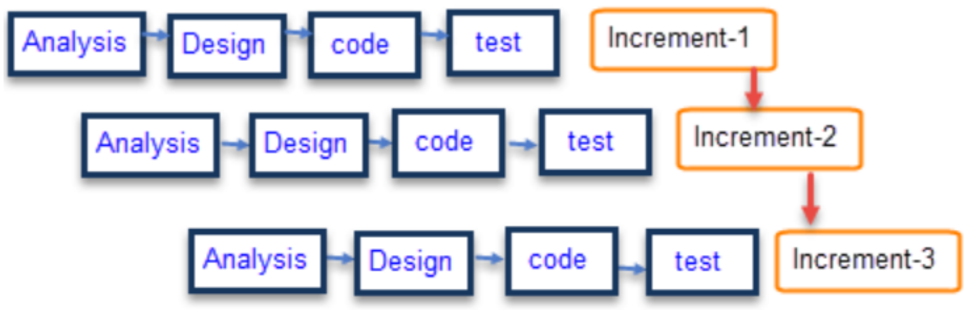


Figure 2.4 Incremental Software Development Model (Sabale et al., 2012)

This model provides some Waterfall model attributes but in an iterative way. Furthermore, the incremental model provides a linear sequence which delivers software in a piece-meal method. The model aims at addressing basic requirements and core products in the beginning or with the first increment. In this fashion, supplementary features (some known, others unknown) are delivered in the remaining increments after getting customers' feedback and confirmation. The incremental model continuously builds pieces of the system until a full end-to-end system is completed while it is slowly adding increased functionality (Alshamrani et al., 2015). In this way, each subsequent release will add a function to the previous one until all designed functionalities are implemented and fulfil the requirements.

2.1.3 Prototyping

The prototyping methodology was formulated to address Waterfall's shortcomings and limitations. When applying this methodology, the customer requirements are not frozen, instead a dummy Prototype is created and built from the beginning in order for the customer to guide if the build is following what they require. When prototyping, the same phases as in the Waterfall are applied, except that they do not follow the stringent sequence (Tanvir et al., 2018). While the Prototype is being built, the user gets a real insight into how the development team visualizes the final product and therefore guides them. The continuous customer involvement is intended at ensuring that the realization of requirements mismatch is identified early. Prototyping is a thorough demonstrative approach which is a feasible, large and complex software system development where full and complete customer requirements may not be upfront. However, prototyping is often not used, as it is perceived to be more costly than Waterfall (Khalifa et al., 2000).

Unlike the Waterfall methodology, prototyping focuses on the visualization of the full software product instead of documenting what is expected of the final product. This method has a potential to have the Prototype approved in advance if it meets the customer requirements early (Tanvir et al., 2018). By allowing users to view and interact with a Prototype, prototyping encourages more user participation and enables users to offer more detailed and accurate feedback. The creation of the Prototype instead of focusing on the documentation can also reduce misunderstandings that come with reliance on natural language (Sabale et al., 2012) and therefore the final product's feel and performance are also accepted.

In this research literature review I've described what Prototypes are, how the prototyping process works, and how software development methodologies include prototyping for exploration, experimentation, or evolution. The software development methodologies that use prototyping of some form are categorized in Figure 2.5 below, as well as how the evolution of a Prototype occurs.

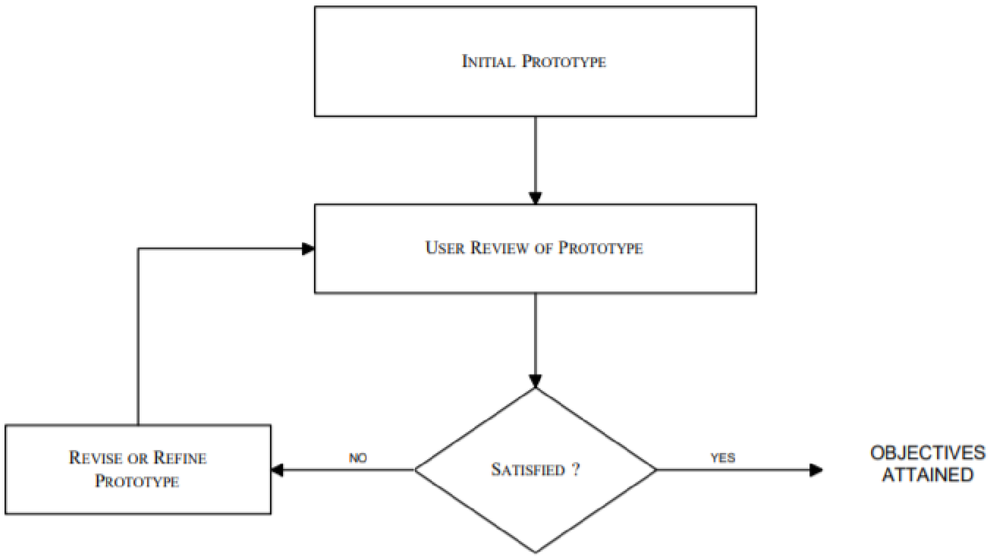


Figure 2.5 The evolution of a Prototype software development (Rodriguez et al., 2020)

In figure 2.5, Rodriguez et al. (2020) represent the evolution of a Prototype. In the process flow above, an initial Prototype would be presented to the user for review and based on the responses from the users, the Prototype would be modified accordingly until it becomes acceptable and accepted as complete. The core characteristic attributes of the Prototype methodology are similar in different projects and different industries as generally identified by the researcher in the subject matter (Rodriguez et al., 2020).

2.1.4 Spiral model

Introduced in 1986 by Barry Boehm, it appeared in an article titled "A Spiral Model of Software Development and Enhancement". Also known as the Spiral Life Cycle Model, it is another iterative form of software development model which can be ideal for projects posing higher risk. This methodology consists of multiple attributes sourced from the Waterfall and Prototype methodologies. These multiple attributes are then arranged in a spiral form as presented in the below figure. Every loop represents a development phase. However, the number of loops are dependent on the size of the project (Krishnan, 2015). Figure 2.6 presents the Spiral model diagram where each loop has four quadrants:

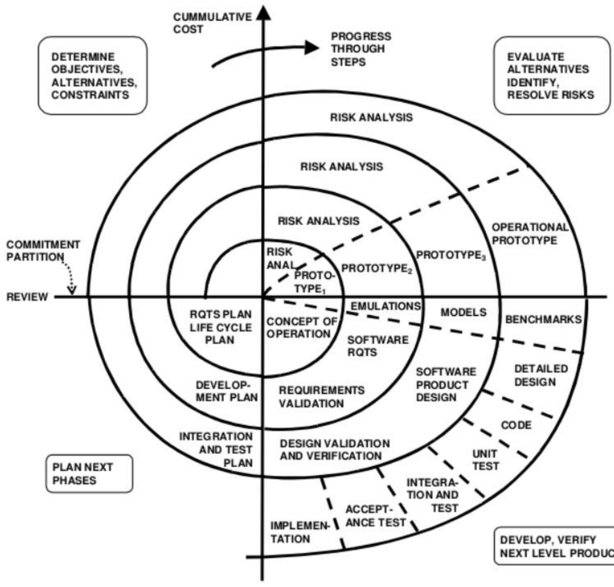


Figure 2.6 Spiral model diagram (Krishnan, 2015)

- In the first quadrant, the purpose is to express the development's objectives and the foreseeable constraints. In this quadrant, the development team is made to comprehend the overall purpose of the project and provide inputs to eliminate project constraints.
- In the second quadrant, the risks identified are dissected and analysed. This analysis includes the formulation of alternatives. Technical and operational issues are prioritized, then the mitigation of the identified risks from the formulated future actions.
- The third quadrant shows the implementation of the actual development work. This is where the product planned for in the above quadrants is put to work. This quadrant is also tasked with the testing of the developed product.

- The fourth quadrant formulates a plan for the upcoming phase. It assesses the progress and informs decisions after evaluating constraints. In this quadrant, the developers have the option to continue to work on the project or to terminate it. If problems were identified that could be resolved, those problems could be re-solved and further steps could be planned. Similar phases are involved in subsequent loops of spiral models. Here, analysis and engineering efforts are applied. This type of life cycle is used for big, expensive, and complicated projects. A project can be stopped if it is determined that it has a high risk of failure and cannot be managed. Assessments at different phases can be done either internally or externally. When high risk analysis is required for a mission-critical project like launching a satellite, the spiral model is ideal.

Spiral models are also referred to as meta-models since they combine the properties of several SDLC models. Both Waterfall and Prototype models are taken into account in the Spiral model. As I do software development in the Spiral model, I do so systematically over a number of loops (similar to the Waterfall model) with a Prototype built after completing each phase and shown to the user (as in the Prototype model). The approach to risk assessment and reduction as well as to follow a systematic approach are enhanced in this way (Krishnan, 2015). In contrast to more traditional requirements-driven, model-based, or other transformation-oriented approaches to software development, the spiral model integrates risk into its approach. A risk management factor can be used to estimate how much time and effort I am willing to devote to other project activities, including planning, change management, quality factors, formal technical reviews, and testing.

2.1.5 RAD (Rapid Application Development)

Like the previously mentioned evolutionary methodologies. Rapid Application Development (RAD) was formed out of frustration with the Waterfall software design process, which frequently resulted in solutions that were outdated or unproductive by the time they were launched. James Martin originated the term "rapid application development" (RAD) in his book "Rapid Application Development" in 1991. According to Martin, RAD is a development lifecycle designed to deliver significantly faster development and higher quality outputs than the standard lifecycle. It is designed to fully utilise the most advanced development software of the most recent generation. Martin identifies the four main components of rapid development as tools, method, people, and management.

Additionally, he emphasises the RAD software development life cycle, which enables companies to generate goods more quickly while simultaneously saving money and time. He also stated that RAD is concentrating on building Prototype models as quickly as possible in order to obtain user input. The RAD methodology is time-driven rather than requirements-driven, yet the software's functionality is defined by the concise requirements and ongoing customer interaction (Akbar et al., 2017). Figure 2.7 below presents a Rapid Application Development process that is evolutionary and a combination of both the Waterfall and Prototype models.

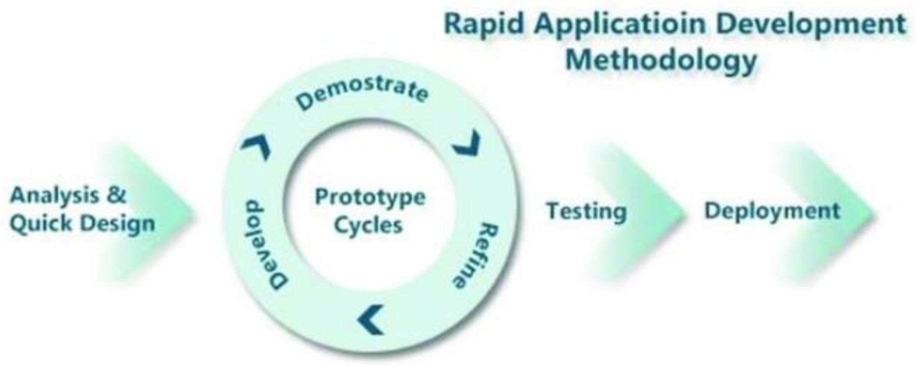


Figure 2.7 Rapid Application Development process (Sabale et al., 2012)

As soon as the customer hand overs the requirements, they are briefly analysed and designed. They then develop a Prototype that would be presented to a customer for their guidance towards the fulfilment of their requirements and that is represented by the Prototype cycle in figure 2.7. Once the Prototype fulfils the customer's requirements, testing is conducted followed by the deployment. Fundamentals of the RAD methodology thus include:

- Choosing the most effective combination of techniques and specifying the steps to take to get there.

- To develop a final product, Prototypes are used, which are then transformed into evolutionary models.
- Gathering requirements and reviewing design using workshops instead of interviews.
- Automating many of the techniques, as well as selecting tools that support the model, Prototype, and code reusability.
- Time boxed development makes it possible for teams to rapidly construct the system's core and refine it in subsequent releases.
- Outlining guidelines for success and describing pitfalls to avoid.

This incremental model is also known as the Rapid Application Development model. The RAD paradigm's elements or functionalities are produced continuously, as if they were small projects. The developments are given a set amount of time to complete, then delivered and built into a functional Prototype. Similar to a Prototype, this can quickly provide something for the customer to view and use, as well as provide feedback on the delivery and their requirements (Cosmas et al., 2018). However, RAD has its shortcomings as follows:

- Relies on great team and individual performances to determine the needs of the company.
- RAD can only be used to construct systems that can be made modular.
- Needs developers and designers with advanced skills.
- High reliance on scarce modelling abilities
- Not suitable for less expensive projects due to the high expense of modelling and automated code generation.

2.2 Agile Software Development

There is adequate evidence of the universality of the problems in the software development industry. There have also been numerous solutions that the industry and scholars have formulated. Currently, Agile Software Development is the popular solution that many are realizing (Kim et al., 2021). A throughput-oriented approach to delivering value to customers as rapidly as possible is known as an Agile approach. It can also be used as a task management framework to apply familiar implementation approaches to the task's completion. The software development industry is under a lot of pressure to provide software quickly and affordably in order to preserve or expand its market dominance (Hatcher, 2019). The main benefit of moving away from conventional approaches was that they followed a paradigm that presupposed scope could be defined up front, a plan could be put in place, and the plan could be executed with little change. This paradigm assumed scope could be defined up front, a plan could be put in place, and the plan could be executed with little change.

Being nimble is the quality of being able to move rapidly. The incremental and iterative approach to software development lies at the heart of this software development methodology. Self-organising and cross-functional teams collaborated to create the needs and solutions (Moyo, 2021). It's a straightforward method of software development that has been used since the 1990s. The heavyweight models, which were renowned for being rigidly regulated, disciplined, and micromanaged, served as the foundation for the development of this model (Moyo, 2021). Customer satisfaction is the most crucial element of this strategy, and it may be reached by providing functional software at an inexpensive price in a timely way (Moyo, 2021). As opposed to the Waterfall model, when the software is provided over months, it is supplied in regular intervals. Agile modelling uses the many working models that are presented to the customer as an indicator of progress. The software product can readily be modified since it is created in small batches. There is a lot of space for collaboration between businessmen and developers since needs from businesspeople arrive often. High emphasis is placed on the value of technical excellence and software design (Hatcher, 2019). The software development team frequently has to adapt to shifting conditions.

Agile modelling is a technique for modelling and documenting of software-based systems that leverages practice. These procedures, which may be applied in more flexible ways, have replaced traditional modelling approaches in software development projects. It works in combination with numerous Agile methodologies such as extreme programming, Agile unified process, and Scrum models (Moyo, 2021). One of this model's greatest benefits is its ability to adapt to shifting project needs. As opposed to other approaches, which frequently result in wasted work, this assures that the development team's efforts are not squandered (Moyo, 2021). The changes are quickly put into practice, which will save time and work in the future. Because of the face-to-face communication and continual consumer input, the development team and client have very few to no assumptions. The use of natural language to document requirements allows for some uncertainty. As a consequence, the client receives high-quality software in the quickest period of time feasible, which pleases the client (Moyo, 2021).

The Agile paradigm has apparent benefits for small projects, but it can be challenging to predict how much time and effort a large project would require throughout the software development life cycle. There isn't much focus on design and documentation since the demands shift so frequently. As a consequence, there is little possibility that the project will deviate from its planned course. Another problem is that the project will be delayed if the customer service person is unclear. Before their resources are merged with those of seasoned developers, who are in a better position to make the decisions required for Agile development, novice programmers have less flexibility to operate (Hatcher, 2019).

Scrum is a well-known Agile software development methodology. Scrum is a development, delivery, and maintenance strategy for complex products. It's a paradigm for approaching difficult adaptive problems and delivering high-value products in a fruitful and innovative way. Short project cycles, known as "Sprints" are used to plan, design, build, test, review, and deploy a usable deliverable (Hatcher, 2019). Scrum is described as "lightweight", "easily understood" and "tough to master". The Scrum framework is comprised of Scrum Teams and its associated duties, tasks, activities, objects, and guidelines. Every component of the framework serves a particular purpose and is crucial to the adoption and success of Scrum. Scrum is defined by a small group of people who are very adaptable and flexible. Scrum Teams iterate and incrementally deliver products, maximizing possibilities for feedback. The Scrum Team is made up of a Product Owner, Development Team, and Scrum Master. Scrum Teams are also distinguished by their capacity to self-organise and collaborate across departments.

In ASD, requirements are defined from the perspective of the user, commonly in the form of User-Stories (Moyo, 2021). The User-Stories are sorted into a backlog and prioritized. A backlog is just a list of User-Stories that have been prioritized. User-Stories are assigned to a set of interim releases from the backlog (refer to Table 1). One of the major reasons companies are favouring the ASD is that customers can change their mind from one release to the next in order to keep the value of the endeavour in line with business and market realities (Hatcher, 2019). For simplification, the software releases are known as Sprints and they are intended to deliver quality working product.

Figure 2.8 below is a generic User-Story template, which is how requirements are formatted in Scrum. These User-Stories are then put in the product backlog (Table 2.1) and are regarded as tasks.

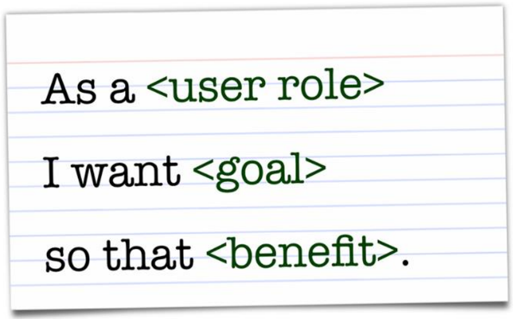

As a <user role>
I want <goal>
so that <benefit>.

Figure 2.8 User-Story Template (Hatcher, 2019)

The above figure represents a structure of a User-Story that qualifies a software requirement to be a Sprint task. For a software requirement to become a User-Story, the template above must be completed with the user role, goal and the benefit of the requirement.

A Sprint is a time-boxed release management practice where a product of the highest possible value is created. It could take multiple Sprints to build enough value to deliver a useful product to the customer in time for a release (refer to table 2.1). As outputs are committed to manufacturing, value accumulates incrementally. Sprints are of short duration (usually two weeks to one month) in which full specification, development, and testing are completed. In other words, a single iteration contains the entire development cycle.

Every Sprint lasts the same amount of time (#Sprint ≤ a month). User-Story completion can be monitored during Sprint execution to track specific progress. User-Stories can be tracked inside a Sprint with associated status using software tools. The tracking and management of Sprints and User-Stories are shown in Table 2.1.

Table 2-1 Tracking, prioritization of User-Stories towards the satisfaction of customer requirements.

User-Stories	Sprint Ready	Priority	Status	Sprint
1	Yes	Medium	In Progress	1
2	Yes	Low	Complete	1
3	No	Low	In Progress	1
4	Yes	Medium	To do	2
5	Yes	High	To do	2
6	Yes	High	To do	2

In table 2.1 above, the User-Stories are broken down into tasks and are allocated into Sprints. These allocations are also based on the prioritization of the tasks and are a piecemeal towards the achievement of a working software. The Sprint column shows that the first three User-Stories are allocated to the first Sprint and are to be done within two weeks, then the remaining three User-Stories are to be done in the following Sprint.

Below I present a real-life example of an ATM withdrawal customer requirement into a Scrum process of fulfilling the user requirements. Instead of having a detailed requirements specification that will include all the functionalities of the ATM system, a piecemeal approach is used, and, therefore, only one function (withdrawal) is analysed, developed, tested and presented to the user at the time.

2.2.1 Example 2.1

In the practical examples below, I will use a withdrawal User-Story that will be simplified in a step-by-step process towards achieving a functional ATM system.

Requirement (User-Story):

1. As a bank customer
   I want to withdraw cash from my bank account through ATM
   So that I have access to my money at more places.
2. Acceptance criteria:

- The customer must be in possession of a valid bank account as well as a valid bank card
- Cash withdrawals can only be made once the customer is logged in.
- System checks to see if the request amount exceeds the balance.
- If so, the system displays the balance and asks the user to enter a new amount.
- If amount entered is less than the account balance, cash is dispensed, and the new balance is displayed.

The User-Story in Example 2.1, therefore, becomes the only specification at the time that is given to the developers and after it passes the testing or quality gate, it is presented to the customer for acceptance. This approach assists the delivery by keeping the customer informed about the tangible progress towards the fulfilment of their requirements and complies to an Agile principle of being able to add and modify requirements in real-time. Having discussed Agile Software Development's purpose, advantages and disadvantages above, I next discuss DevOps which is an Agile practice that embeds software development into the broader business operations in an effort to have thoroughly supported software systems that talks to everyday business needs (Masombuka, 2020).

2.2.2 DevOps

There is also a newly found appreciation for both the production and the afterlife of a software product beyond its release. This appreciation insists on a collaboration between the software development and business operations, thus known as DevOps. The development team works with code, whereas operations personnel work with live systems and are frequently in contact with clients. DevOps combines both of these skill sets. Others in the industry feel the word refers to new development, testing, release, support, and metrics collection requirements. This standardization sanitizes the re-lease and support processes for many businesses. Collaboration, automation, measurement, information sharing, and the use of online services underpin DevOps (Erich et al., 2014). The links between DevOps features and software quality attributes are depicted in Figure 2.9.

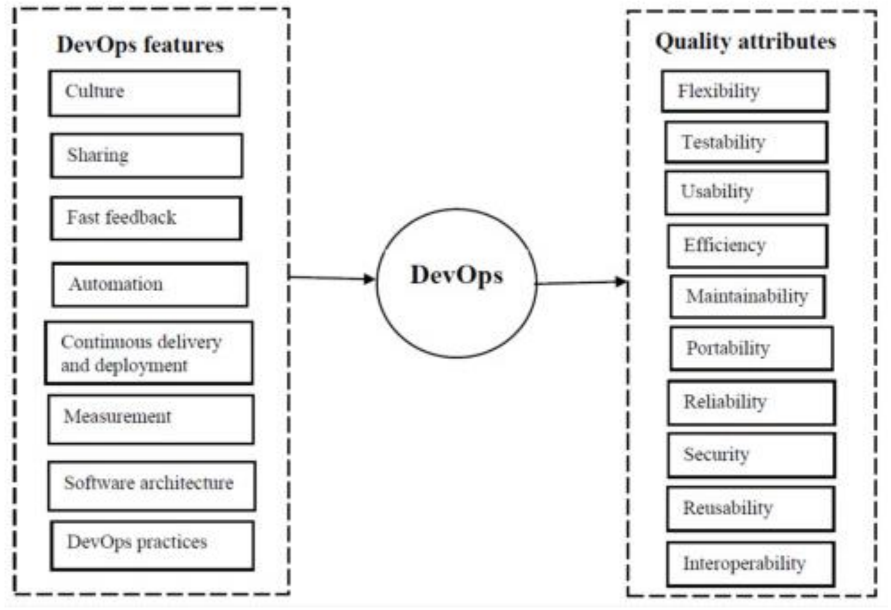


Figure 2.9 Relationship between DevOps features and software quality (Mishra et al., 2020)

It is worthwhile to look at DevOps' continuous delivery and software for on-time completion with quality since it is an element of fundamental relevance for its success. Continuous delivery entails streamlining and automating the deployment process (Mishra et al., 2020). Within the DevOps context, continuous delivery can be utilised to help in product delivery.

Every organisation's culture has an impact on how workers work and share responsibility for the end product's quality. Shared duties, open communication, trust, and mutual respect are all important aspects of the DevOps culture. When it comes to quality assurance, the interaction of these components is crucial. Figure 2.10 shows an overview of the DevOps process from the point of view of the software architect.

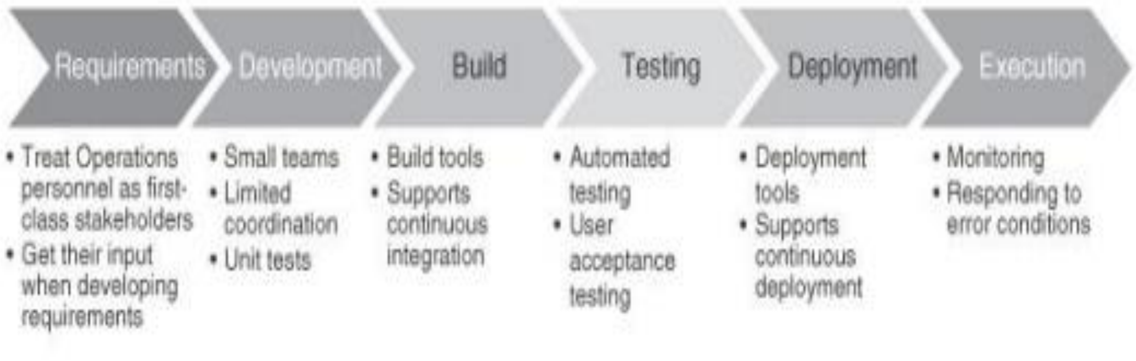

Requirements
Development
Build
Testing
Deployment
Execution
• Treat Operations personnel as first-class stakeholders
• Get their input when developing requirements
• Small teams
• Limited coordination
• Unit tests
• Build tools
• Supports continuous integration
• Automated testing
• User acceptance testing
• Deployment tools
• Supports continuous deployment
• Monitoring
• Responding to error conditions

Figure 2.10 The DevOps process (Malassu, 2020)

The process in figure 2.10 is only defined as a goal to reduce the time between changes made to the code and deploying the code into production (Malassu, 2020). This definition does not specify which methods should be used in order to work towards this goal, which is in contrast to many other common definitions that tend to emphasise the connection between DevOps and Agile methodology.

Quality assurance is critical in bridging the gap between development, operations, customer service, and the customers themselves. It is critical that enterprises, or anyone advocating for a DevOps transformation, do not view DevOps solely through the lens of development, which is where DevOps is predominantly geared (Masombuka, 2020). Until recently, software development and operations were viewed as two distinct disciplines. The research on DevOps demonstrates that researching the two together has some relevance. This is due to the fact that many companies are reintegrating development and operations. I recognize that academic research should not be largely influenced by industry trends, which are frequently the topic of hype. Academic research, on the other hand, should complement industry innovations by locating information that supports or refutes the value proposition of these developments (Masombuka, 2020). The below figure 2.11 presents the pre-deployment (Dev) and the post-deployment (Ops) phases/tasks.

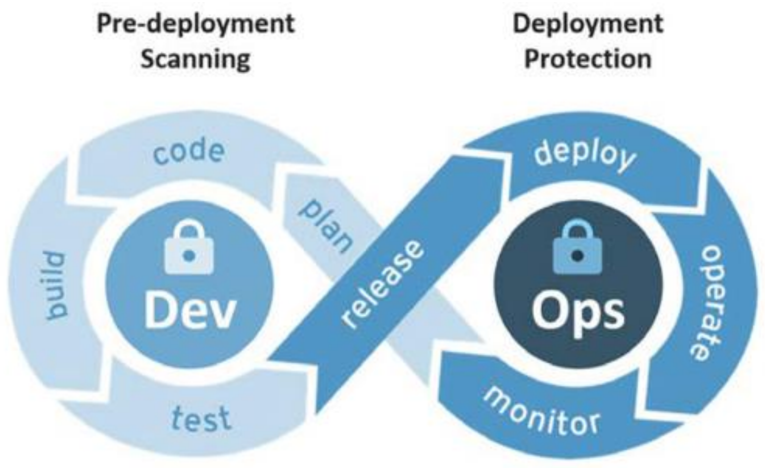


Figure 2.11 DevOps process model (Masombuka, 2020)

The above figure presentation of the pre-deployment and post-deployment tasks indicates the unending cycle that integrates software development and operations.

In the practical example relating to the ATM User-Story Example 2.1 above, the following Example 2.2 I will still use a withdrawal User-Story to show how DevOps work and non-DevOps environments work.

Team A: Develops and delivers an ATM withdrawal function and hands it over to Team B
Team B: Is responsible for the support and maintenance of the ATM withdrawal function once it is deployed.

- Team B is likely to become frustrated in their task to maintaining a system that they only have an idea why it was built. It will still take a long time for them to be comfortable and know the functions of the system.
- There is likely to be continuous conflicts between Team A and Team B, with both parties accusing each other and defending their work.

In the non-DevOps environment's picture painted above, the exemplary difference with the DevOps is that Team A and Team B will be collapsed and only one team will develop, support, and maintain systems and their functionalities.

With DevOps being relatively new, it has come at an expense of many misunderstandings. Below I have listed three major myths (Kim et al., 2021).

Myth 1— DevOps replaces Agile: DevOps ideas and practices are consistent with Agile, and many people believe DevOps is a natural progression from the Agile journey that began in 2001. Many challenges related with configuration and release management processes are solved as processes become completely automated. (e.g., keeping the configuration management database and definitive software libraries up to date).

Myth 2— DevOps and information security and compliance are incompatible: Information security and compliance professionals may be concerned about the lack of traditional controls (e.g., segregation of duties, change approval processes, and manual security reviews at the conclusion of the project). That isn't to say that DevOps firms don't have strong controls in place. Instead of performing security and compliance operations at the end of a project, controls are integrated into every stage of everyday work throughout the software development life cycle, resulting in improved quality, security, and compliance.

Myth 3— DevOps stands for "NoOps", meaning the elimination of IT operations. Many people misunderstand DevOps to mean that the IT Operations department has been completely eliminated. This is, however, a rare occurrence. While the type of IT Operations job may vary, its importance remains constant. IT Operations works with Development far sooner in the software life cycle, and IT Operations continues to collaborate with Development even after the code has been delivered to production.

IT Operations (together with QA and Infosec) become more like Development when it comes to product development, with the product being the platform that developers utilise to work safely, rapidly, and securely. DevOps has allowed for the creation of a world in which product owners, development, QA, IT Operations, and Infosec collaborate not just to aid each other, but also to ensure the overall success of the organisation (Kim et al., 2021). Development is concerned not only with adding user features, but also with actively ensuring that their work flows easily and regularly across the full value stream, without causing confusion or disruption to IT Operations or any other internal or external customer.

Testing and Infosec activities occur only at the end of a project, too late to address any flaws detected; and practically any key action needs too much human labour and too many handoffs, leaving critical chores until the end. I looked at the manufacturing revolution of the 1980s to better grasp the possibilities of DevOps. Manufacturing companies improved plant efficiency, customer lead times, product quality, and customer happiness by using lean principles and practices, allowing them to compete more effectively (de Vries et al., 2016). Prior to the revolution, the typical wait time for orders at manufacturing plants was six weeks, with less than 70% of orders being shipped on time. Average product lead times had fallen to less than three weeks by 2005, thanks to widespread adoption of lean methods, and more than 95% of orders were fulfilled on time (Kim et al., 2021). Organisations that did not adopt lean practices lost market share, and some even went out of business. What was acceptable in prior decades is no longer acceptable.

According to Kim et al. (2021), by the 2000s, breakthroughs in technology and the adoption of Agile concepts and practices had reduced the time necessary to develop new features to weeks or months but deploying them into production still took weeks or months, frequently with disastrous results. "Every industry and corporation that is not bringing software to the core of their business will be disrupted", said Jeffrey Immelt, the former CEO of General Electric. More than ever, how I manage and do technological work determines whether or not I will succeed in the marketplace, let alone survive. IT Operations will be in charge of providing clients with stable, reliable, and secure IT services, making it difficult, if not impossible, for anyone to make production adjustments that could affect output.

I used to embrace approaches like structured programming, which foretold failure and left people helpless to modify the outcome. Burnout, with its accompanying feelings of exhaustion, cynicism, and even hopelessness and despair, was often the result of this impotence. Many psychologists believe that one of the most harmful things I can do to fellow human beings is to create institutions that generate emotions of powerlessness. Instead of project teams that are reassigned and shifted around after each release, never receiving feedback on their work, DevOps necessitates that I maintain teams intact so that they may continue iterating and improving and using what they've learned to better achieve their goals. Controlled, predictable, reversible, and low-stress releases are also available. It's not just feature releases that are calmer; all kinds of issues are discovered and fixed earlier when they're smaller, cheaper, and quicker to fix. They spent 50% less time resolving security concerns by incorporating security objectives into all phases of the development and operations processes (Kim et al., 2021).

DevOps, on the other hand, demonstrates that with the correct architecture, technical processes, and cultural standards in place, small teams of developers can build, integrate, test, and release changes into production fast, safely, and autonomously. DevOps-adopting companies can raise the number of releases each day linearly as the number of developers increases, as Google, Amazon, and Netflix have done.

2.2.3 DevOps in Agile

The design phase has been replaced in Agile by a less cyclic, democratic development-to-deployment process. Simultaneously, engineering prowess enables teams to automate much of the tracking that previously needed 24/7 staffing of human system monitors, improving the work-life balance of engineers all around the world. The primary motivation for automating was to collect performance information for benchmarking and stress tolerance. Those who did adapt were compelled to shift roles away from the traditional development/QA/support architecture and form more collaborative teams to design fault-tolerant systems with a functionally unlimited life cycle (Roche, 2013).

DevOps' emphasis on continuous deployment necessitates a development organisation's ability to achieve appropriate quality in the short time between a source-control submission and a release. This means that deployment must be undetected to users, who may have their software version altered out from under them in the middle of a session. It also implies that any issues that arise over the course of such a covert roll-out will be addressed quickly and effectively. The most beneficial contribution of the DevOps culture appears in the software support position to a greater extent and embedding the accompanying culture throughout the product lifecycle pushes valuable enhancements all the way back to the application design phase (Roche, 2013).

The testing schedule should evolve away from the archaic Waterfall approach and toward one that recognises the value of testing and release simulation. This enables DevOps to encourage a programmatic study of client scenarios or User-Stories. Extending proactive client scenario analysis should effectively automate the range of use-case paths that a user might (and, perhaps more significantly, is likely to) encounter (Masombuka, 2020). As a result, cantankerous roles have almost no place in a Scrum Team, and everyone's goal is to have a production environment that is also used as a QA environment. Figure 2.12 shows the DevOps in an Agile ecosystem.

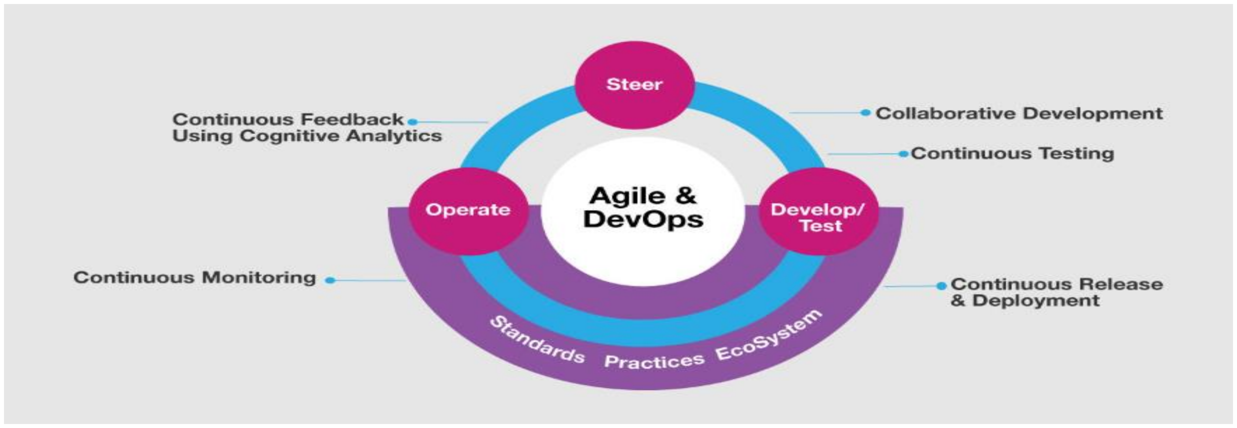

Steer
Continuous Feedback Using Cognitive Analytics
Collaborative Development
Continuous Testing
Operate
Agile & DevOps
Develop/ Test
Continuous Monitoring
Continuous Release & Deployment
Standards Practices EcoSystem

Figure 2.12 DevOps in Agile ecosystem (Roche, 2013)

This above ecosystem shows the continuousness of both the pre-deployment (Dev) and post-deployment (Ops) activities (Roche, 2013). Figure 2.12 emphasises on the continuousness of the process of software development that goes beyond the release of the software product. From figure 2.12, I have extracted the following practices:

- The development and testing is collaborative.
- The end-to-end testing of the features is automated.
- The software release is continuous, and the deployment can be done at any time.
- Version control takes priority in managing the frequent releasing of features.
- The production environment is proactively monitored, and continuous feedback is shared with everyone for the purpose of continuous improvement.

As an illustration, using the continuous deployment methodology, a bank's value-added service loan to customers could be released to production at any time after Christmas, ensuring that the feature or value-added service package is kept ready to go-live and made available after Christmas via a feature toggle.

In the following topic, I discuss Scaled Agile Framework (SAFe) which is an enterprise level Lean-Agile practice. This framework is divided into three segments Team, Programme and Portfolio which will be discussed.

2.2.4 Scaled Agile Framework (SAFe)

The Scaled Agile Framework aims to include lean concepts and Agile techniques at the enterprise level. To maximize the advantages of the Agile methodology, a number of frameworks have been put out to give guidance for expanding Agile development across the organisation. The Scaled Agile Framework is a relatively new and well-known paradigm (SAFe). Despite these concerns, SAFe has quickly acquired traction in the software development community and has emerged as a viable option for firms looking to scale up Agile development (Turetken et al., 2017). It addresses scalability by scaling up "some" Agile principles in addition to linking new practices and concepts with basic and scaled Agile activities (such as release train, business, and architecture epics, and portfolio backlog). The benefits it seeks to provide include a speedier time to market, more productivity and quality, lower project costs, and decreased risks (Turetken et al., 2017).

Even though some of these advantages have allegedly been proven by successful SAFe adoptions, most of these tales are self-reported and have a limited scope. Academic study is required since SAFe is being used more and more in business and practice. A review of the main SAFe sources, however, indicates the absence of a clear roadmap to guide firms through the necessary SAFe adoption and preparation. Instead, rather than providing any particular implementation strategy or method, the SAFe focuses entirely on identifying the best practices, roles, and artefacts of Agile and lean concepts. Companies trying to implement SAFe may find it difficult to set priorities and take charge of efforts to use Agile and SAFe approaches.

SAFe aims to integrate the existing bodies of work from Scrum, XP, lean, and Product Development Flow. In conclusion, there are three levels in the framework: team, programme, and portfolio. These three levels have fictitious borders that work as a model for separating their differences in scope and size. At the team level of the framework, Agile teams are in charge of planning, creating, and testing software in fixed-length iterations and releases. On this level, the SAFe framework blends Agile technological techniques with Agile project management concepts like Scrum and XP. User-Stories, for example, are an XP notion, whereas Sprint Planning and daily stand-ups are standard Scrum components. Each iteration adopts a ‘Definition of Done' and retrospectives. To promote better integration among teams, teams operate on the same cadence and iteration lengths. These Agile teams are usually made up of nine people (Turetken et al., 2017).

The SAFe Enterprise Levels, shown in Figure 2.13, are a visual depiction of the framework that may be used as both an organisation and a process model for Agile requirements practices.

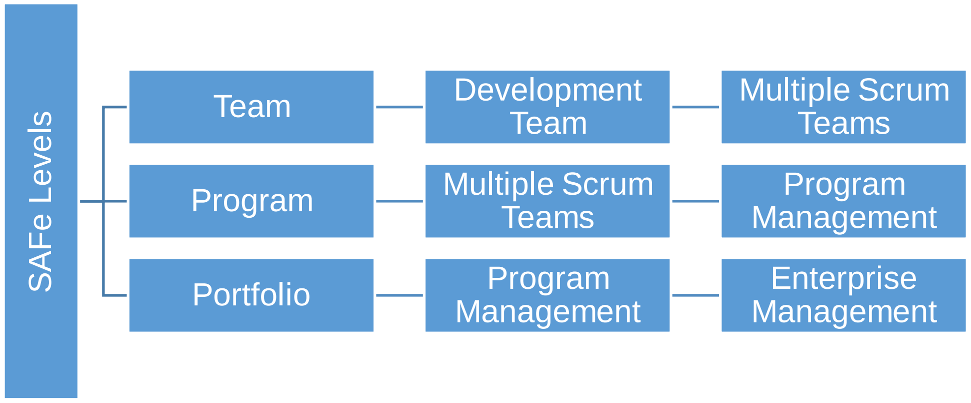


Figure 2.13 SAFe levels and teams

In summary, the framework is divided into three levels: team, programme, and portfolio. The composition of these levels is also shown in Figure 2.13. These three levels have artificial borders that serve as a model for abstracting the scope and size between them (Gustavsson, 2019).

1. Team Level

SAFe takes a multi-team approach to enterprise software scaling. Many teams collaborate with one another in a mutually beneficial way. But, regardless of the responsibilities assigned to each team, each team is Agile by nature (Gustavsson, 2019).

Each Scrum Team consists of 5-9 persons working toward a programming goal, as is typical of Scrum Teams. Once every two weeks, a Systems Team, also known as a Design Build Test (or D/B/T) Team, is in charge of testing and delivering software.

A "Sprint" is defined as a two-week span. During a Sprint, each team uses Extreme Programming (XP) methodologies to deliver their share. SAFe varies from standard Agile in that teams are interconnected, and Sprints can happen at the same time. One of SAFe's distinctive goals is to establish a perceived rhythm that synchronizes the progress of all teams. In the team programming environment, the goal is to encourage consistency over variety.

Figure 2.14 presents a practical example of a Team Level structure.

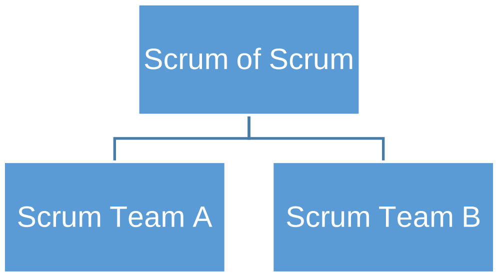


Figure 2.14 Example of a Team Level

In figure 2.14:

- Scrum A: Is responsible for the withdrawal function of the system.
- Scrum B: Is responsible for the deposit function of the system.

Although these teams are responsible for developing and supporting different functions in the ATM system, a Scrum of Scrum (SoS) is initiated where both these teams discuss their developments and impediments faced, particularly the code that may have a level of integration. Not every member of Scrum A and B belongs to the SoS, but the a few members particularly the Scrum Masters are chosen to represent the different teams.

2. Programme Level

An Agile Release Train is made up of three to five teams working together on a piece of software for a Programme Iteration (PI). During a PI, it is the principal vehicle for delivering value. Within the programme, the PI is a greater unit of measurement. A PI has numerous teams who were timed to complete a Sprint synchronously, whereas a single Sprint has only one team generating one component of the software system. The Innovation and Planning Iteration (IP), which occurs at the end of the PI, affects all of the development (Gustavsson, 2019). All of the work done during the PI is tested and displayed, and there is an Inspect and Adapt session.

Figure 2.15 presents an example of a Programme Level structure.

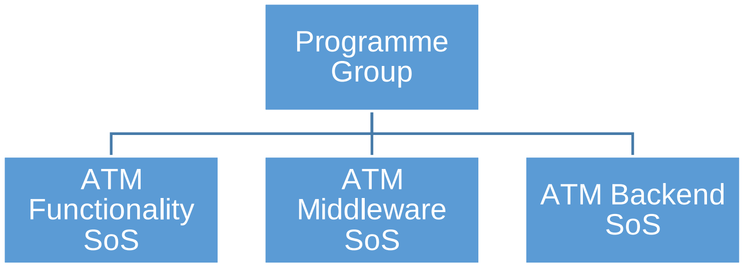


Figure 2.15 Example of a Program Level

In the programme level an SoS dealing with the ATM functionality, an SoS dealing with backend from a programme group working in sync on a piece of software development for a Programme Iteration (PI). The programme group consists of different SoSs that develop different end-end functionalities of the software systems.

3. Portfolio Level

The Portfolio is made up of various value streams that have been grouped together. Through issues like strategy, investment finance, programme management, and governance, it is linked to the total enterprise software.

Each subject aids in overall budget planning for a six to twelve-month period. The development needed to realise themes is defined by large development activities from many programmes within the company (Gustavsson, 2019).

Figure 2.16 presents a practical example of a Portfolio Level structure.

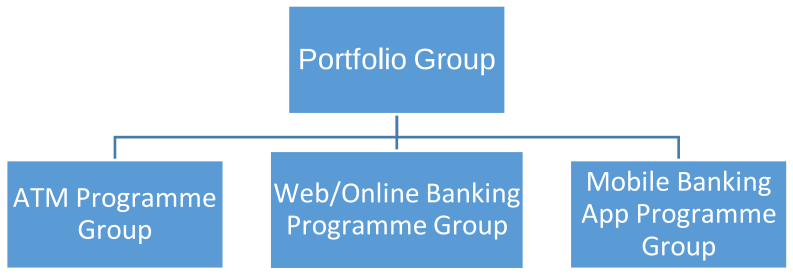

Portfolio Group
ATM Programme Group
Web/Online Banking Programme Group
Mobile Banking App Programme Group

Figure 2.16 Example of a Portfolio Level

This is at an enterprise/organisational level where different programme groups plan, strategise, and budget. In addition to the ATM Programme Group, the portfolio group may include Web/Online Banking Programme Group and Mobile Banking App Programme Group.

Many businesses have adopted the SAFe framework since its 2011 introduction, and they've written white papers or technical reports about their experiences with it. In a couple of months, these claims assert that they have improved in a number of areas, including greater ROI, 20–30% faster time to market, 40–50% fewer post-release problems, better alignment with customer demands, and a 20–50% increase in productivity (Turetken et al., 2017). Turetken et al.’s (2017) research additionally includes issues like the need to define the right degree of requirement information at the right point in the lifecycle and the need to maintain releasability throughout the development lifecycle owing to late defect discovery.

According to the studies, good release planning requires proper preparation, orchestration, and facilitation of remote programme events. The results also confirm that geographically dispersed teams have reduced productivity as a result of a lack of alignment and programme execution (Turetken et al., 2017). According to Version One and Collabnet's

12th State of Agile Report, SAFe is now the most frequently used large-scale Agile framework, with a use rate of 29%. According to Scaled Agile Inc., there are 300,000 SAFe-certified practitioners in 110 countries, and SAFe-certified experts work for 70% of Fortune 100 firms. In practice, enterprises are unable to fully implement Agile development principles in a short amount of time. Maturity models can help businesses by pointing them in the right direction when it comes to practices and how they should be adopted and implemented (Turetken et al., 2017).

A maturity model is a theoretical framework made up of a collection of best practices that help businesses enhance their operations in a certain area. The main objective of mature models is to illustrate the stages of development. Maturity models are generally held ideas about how an organisation's capabilities develop in a stage-by-stage way along a desired, logical path. They are based on the assumptions of predictable patterns of organisational growth and change. Generally speaking, maturity models are distinguished by a finite and arranged hierarchy of maturity levels, each of which describes the traits or actions necessary to reach that level.

2.4.5 Scrum

Scrum is a methodology for managing projects, while Agile is a way of thinking (Bhavsar et al., 2020). When you switch to the Agile Software Development Methodology, your entire team must be committed to re-evaluating how they provide value to their clients. You may start thinking in this way by using Scrum to apply Agile ideas to your regular communication and work (Baham, 2019). Scrum is a heuristic framework built on continuous learning and situational adaptation. It acknowledges that at the beginning of a project, the team is in the dark about everything and will pick things up along the way. With built-in prioritization and quick release cycles, it's intended to let teams adjust naturally to changing circumstances and user requirements so that your team can continuously learn and get better (Bhavsar et al., 2020).

The core Scrum process components and roles are defined below.

Table 2-2 Scrum process components and roles (Bhavsar et al., 2020)

Process Components	Definition
Daily Scrum Meetings	A meeting when the Scrum Team shows what they accomplished during the Sprint.
Sprint Backlog	A list of the Product Backlog items the team commits to delivering plus the list of tasks necessary to delivering those Product Backlog items.
Product Backlog	A prioritized list of desired product functionality.
Increment	A concrete steppingstone toward the Product Goal.
Sprints	The intervals into which the development process is divided.
Sprint Planning Meetings	A meeting where the Product Owner describes the highest priority features.
Sprint Reviews	A meeting when the Scrum Team shows what they accomplished during the Sprint. Typically, this takes the form of a demo of the new features.
Sprint Retrospectives	A brief, dedicated period of time set aside at the end of each Sprint to deliberately reflect on how the team is doing and to find ways to improve.
Roles	**Description**
The Development Team	Professionals who do the work of delivering a potentially releasable increment of “Done” product at the end of each Sprint. These professionals mainly consists of Business/System Analysts, Developers and Testers.
Product Owner	A person who is responsible for maximizing the value of the product and the work of the Development Team.
Scrum Master	A person who is responsible for ensuring Scrum is understood and enacted.

At this stage, I present a figure Scrum process flow. Figure 2.17 provides an overview picture of a full Sprint. The Sprint is a piece of a project broken down into a two-week iteration. The product backlog becomes the broken-down piece of requirement that the Development Team will convert into a Sprint Backlog and finish within the two weeks timebox (Bhavsar et al., 2020).

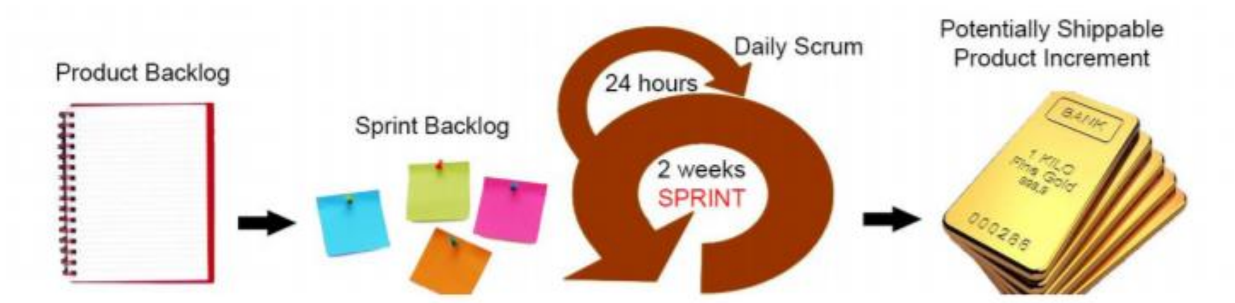


Figure 2.17 Scrum process flow (Bhavsar et al., 2020)

A Scrum process flow for a work item is shown in Figure 2.17. Work items are initially pushed into the Product Backlog. The Development Team adds work items to the Sprint Backlog in order to achieve the Sprint's DoD (Definition of Done). Completed work items are confirmed during Sprint Review and then provided as a product increment (Bhavsar et al., 2020).

2.4.5.1 A day in Scrum environment

Below, I portray a picture of how a typical day of a Scrum's Development Team and a Scrum Master unfolds.

Figure 2.18 Illustration of a Development Team gathered around the Sprint Backlog board (https://watisscrum.nl/sprint-backlog/)

08:30am As per figure 2.18, the Development Team gathers around the Sprint Backlog/User-Stories board and each one of them updates their allocated User-Stories' statuses (To do, Doing, Done)

8:32am The Development Team holds a daily Scrum meeting, where each individual answers the questions:

Table 2-3 Daily Scrum's questions and typical answers from the Development Team

Questions	Typical Answers
What did you do yesterday?	I completed 50% of the User-Story's GUI development.
What will you do today?	I will complete the remaining 50% of the User-Story's GUI development.
Are there any impediments in your way?	Yes, the database team has not completed migrating the data and therefore integration cannot be finalized.

8:45am The daily Scrum meeting ends, and members of the Development Team go back to ensuring they fulfil their commitments.

8:46am The Scrum Master collates the impediments, including defects.

9:30am The Scrum Master meets with other Scrum Masters in what is known as the Scrum of Scrums (SoS) in discussing each team's progress towards the completion of a shippable software product.

During the day The Scrum Master ensures that the Development Team's impediments are removed.

During the day The Development Team attends to the iterative tasks from the Sprint Backlog. These tasks include coding and testing of the User-Stories that will ultimately form shippable software products.

2.3 Summary

In chapter 2, I presented a literature review on Traditional Software Development Methodologies and Agile Software Development techniques. In the traditional methodologies, I selected to focus on five specific methodologies that were found to be most popular in surveys conducted by Akbar et al. (2017) and Mushashu et al. (2019). This was to highlight how software development methodologies have evolved through time and how they have assisted software systems to advance to where they are right now. Following the discussion on the historic traditional methodologies, I then focused on the Agile Software Development which largely forms part of the dissertation.

2.4. Conclusion

Over the recent years, Agile Software Methodologies have shown a consistent rise in adoption (Hatcher, 2019). Most of the software industry's efforts have been dedicated to the cultivating of this methodology instead of formulating new methodologies. Researchers have chosen to cultivate ASD and come up with enhanced techniques that can apply to different environments. Techniques like DevOps and SAFe are for the incorporation of other enterprise structure into Agile as the agility concept has become useful beyond the development of software. Although there has been a significant uptake of Agile Software Development, the traditional methodologies are still largely prevalent in the software development industry. The WSD and V-Model are particularly still used in 55% of the organisations surveyed by Akbar et al. (2017). In this survey, the researchers take into consideration that these two methodologies are significantly similar and therefore the prevalence of one is somehow seen as correlative.

In the following chapter, I explore the Formal Methods literature, as I attempt to identify to what extent Formal Methods can be implemented in an Agile Software Development methodology.

Chapter 3: Formal Methods

Following the Chapter 1 introduction, Chapter 2 contained timeline work on the different traditional software development methodologies and Agile Software Development (ASD) techniques. The ASD has evolved rapidly in recent years (Hatcher, 2019) with new techniques and versions being introduced regularly. The ASD therefore should not be seen as absolute, as continuous improvement remains a permanent feature of this methodology.

Formal Methods (FM) is a set of mathematical approaches for formally specifying and deriving a programme from its specification, which I examine in Chapter 3. Formal Methods can be used to formally express the requirements of a proposed system, to derive a programme from its mathematical specifications, and to offer proof that the actual programme meets those criteria. They've mostly been used in the sphere of safety-critical applications (O'Regan, 2020).

3.1 Introduction

It has long been believed that Formal Methods are the best way to assist the software industry produce more reliable and trustworthy software. Despite this firm conviction and several individual success stories, there doesn't seem to be any appreciable change in the development of industrial software. The software industry as a whole is actually developing quickly, and the gap between what Formal Methods can do and standard software development practice does not seem to be closing (in fact, it could be expanding) (Huisman et al., 2020). Using Formal Methods for large software systems represents mathematical soundness, i.e., a method that can be proven. Formal Methods are reportedly not well adopted by industry practitioners despite considerable advancements in the FM field over a long period of time and compelling evidence of their benefits (Nemathaga and van der Poll, 2019). This issue has been the subject of several hypotheses, some of which contend that they increase the time of the software development cycle, necessitate difficult mathematics, require inadequate tools, and are incompatible with other software products. There is scant evidence to support any of these claims (Nemathaga and van der Poll, 2019).

3.2 Implementing Formal Methods

Formal Methods (FMs) denote the use of discrete mathematics and logic to develop provably correct, or at least highly dependable software. One of the aims of the use of FMs is to eliminate ambiguity and uncertainty of natural language specifications by expressing constructs and operations using mathematical logic symbol formulas or formal diagrams with clear meaning. In terms of formal specification languages, examples include languages that use mathematical symbols and letters, such as logical or process algebra, and schematic specification languages, such as state-diagram (Bowen, 1996). The term "formal verification" refers to the use of mathematical and logic verification methods to determine whether or not the system design and requirements created using the formal specification process are met.

Every formal approach has a distinct relevant domain, and the results may be obtained by using it in this area. Furthermore, as noted in the preceding section, the ease of use should be addressed for the application, as well as numerous concerns such as the selection of a dependable tool to support the formal approach.

One of the successful formal methods is Z, a formal specification language based on first-order logic and a strongly-typed fragment of Zermelo-Fraenkel set theory (Enderton, 1977).

- The idea to construct a Z Schema of the system is to specify a model with the below characteristics (Spivey, 1998):
    - high level
    - idealized details
    - does not detail implementation specifics.
- A model of the system consists of (Spivey, 1998):
    - description of *system state space*
    - description of *system operations*.
    - Natural-language prose.

An important part of a formal specification is to also describe it in natural language for the sake of users who may not be proficient in the use of mathematical formalism.

3.3 Formal specification using Z

A Z specification is developed using the Established Strategy (ES) for constructing and presenting a specification. The ES consists of the following steps (van der Poll and Kotze, 2005):

1. Natural language description and basic types
2. Definition of the State Space
3. Initial state of the system and proof that such state exists, i.e., it can be realised.
4. Operation Schemas
5. Calculate the precondition of each abstract operations on the state.
6. Table showing all the partial operations together with their inputs, outputs and pre-conditions.
7. Definition of all schemas that present error conditions.
8. Use the Z schema calculus to make all the partial operations total (robust).

Next, we illustrate the use of Z in terms of the ES through developing a specification of which the functionality is to maintain a phone book with the names and phone numbers of people.

3.3.1 Natural language description and basic types

As per the ES, the first step is to define the basic types to are to be used in the system. Basic types in Z serve much the same purpose as types in a programming language, e.g., *Integer*, *String*, and so forth.

Our basic types are:

[*NAME, PHONE*]

NAME represents the set of all the names that could ever be entered into our phone book, while *PHONE* represents the set of all the possible phone numbers.

Although a person may have more than one phone number and/or a phone number may be shared amongst multiple persons, for the purposes of this example; we assume that that is never the case.

3.3.2 Definition of the State Space

Our phone book state space may be defined by the following schema.

(Assume that *NAME* is a set of names, and *PHONE* is a set of phone numbers.)

PhoneBook

$known : \mathbb{P}\ NAME$

$tel : NAME \nrightarrow PHONE$

$known = \mathrm{dom}\ tel$

Z Schema 3.1 PhoneBook state-space schema

- The declarations part of this schema introduces two variables, called components in Z: *known* and *tel*.
- The component *known* represents the set of all the names presently in our phone book. Since it is a set of names, it is of type $\mathbb{P}\ NAME$, i.e., it is a subset of *NAME.*
- This variable will be used to represent all the names that we know about — those that we can give a phone number for.
- The value of *tel* will be a partial function from *NAME* to *PHONE*, i.e., it will associate names with phone numbers.
- The declarations' part is separated from the predicate part by the horizontal line.
- The predicate part contains the following invariant:
    - The domain of *tel* equals the set *known.*

3.3.3 Initial state of the system and proof that such state exists.

InitPhoneBook

$PhoneBook'$

$name' = \emptyset \wedge phone' = \emptyset$

Z Schema 3.2 InitPhoneBook schema

Z Schema 3.2 represents the initialisation of the PhoneBook where the *NAME* and *PHONE* are still empty sets.

Proof:

$\vdash$ *PhoneBook ' • InitPhoneBook* (3.1)

(3.1) presents that a state can be realised such that it satisfies the requirements of *Init-PhoneBook*. I then need to show:

$\vdash \exists\ known' : \mathbb{P}\ NAME;\ \ tel' : NAME \nrightarrow PHONE\ |$

$known' = \emptyset \land tel' = \emptyset$ (3.2)

The proof of (3.2) follows trivially since the empty set values are specified in schema *InitPhoneBook*. The proof indicates there is indeed an initial state from which the system may start. The proof follows trivially, since the empty set values are specified in schema *InitPhoneBook*.

3.3.4 Operation Schemas

- In specifying a system operation, we must consider:
  - the objects that are accessed by the operation, and of these:
    1. the objects that are known to remain unchanged by the operation (cf. value parameters).
    2. the objects that may be altered by the operation (cf. variable parameter).
  - the pre-conditions of the operation, i.e., the conditions that must be true for the operation to succeed.
  - the post-conditions —the conditions that hold after the operation, provided the pre-condition was satisfied before the operation.

Below is a schema to add a name and phone pair to the phone book.

AddName

$\Delta\ PhoneBook$

$name? : NAME$

$phone? : PHONE$

$name? \notin known$

$tel' = tel \cup \{name? \mapsto phone?\}$

Z Schema 3.3 AddName schema for altering PhoneBook

- This schema accesses *PhoneBook* and may change it (viz. Δ).
- Two inputs: a name (*name?)* and phone number (*phone?*).
- Pre-condition: the name is not already in the database.
- Post-condition: *tel* after the operation is the same as *tel* before the operation with the addition of maplet *name?* ↦ *phone?*
- Appending a *′* to a variable means ‘the variable after the operation is performed’.

Next is a Z schema specifying the lookup operation:

Find

$\Xi\ PhoneBook$

$name? : NAME$

$phone! : PHONE$

$name? \in known$

$phone! = tel(name?)$

Z Schema 3.4 Find schema for searching the PhoneBook

- Return to the telephone book example and consider the ‘lookup’ operation: we put a name in and get a phone number out.
    - this operation accesses the *PhoneBook* state.
    - it does not change it (viz. Ξ).
    - it takes a single ‘input’ - a name for which we want to find a phone number.
    - it produces a single output —a phone number.
    - it has the pre-condition that the name is known to the database.

The preceding schemas illustrate the following Z conventions:

- placing the name of the schema in the declarations part ‘includes’ that schema—it is as if the variables were declared where the name is (cf. the well-known object-oriented inheritance).
- ‘input’ variable names are terminated by a question mark.
- the only input is *name?.*
- ‘output’ variables are terminated by an exclamation mark.
- the only output is *phone!.*
- the Ξ (Xi) symbol means that the *PhoneBook* schema is not changed (i.e., it remains invariant).
- if we had written a Δ (delta) instead of Ξ, it would mean that the *PhoneBook* schema may change.
- the pre-condition is that *name?* is a member of *known.*
- the post-condition is that *phone!* is set to *tel*(*name?*). In standard Z this is written as *tel name?*, yet I believe the brackets provide for added clarity.

Next is a Z schema specifying an operation to delete a number from *PhoneBook*:

Delete

$\Delta\ PhoneBook$
$name? : NAME$

$name? \in known$
$tel' = \{name?\} \lhd tel$

Z Schema 3.5 Delete schema for removing a number

A correct operation of Delete specifies that the name to be removed from the phone book must be known, followed by removing the name and phone associated with name? from the phone book. This is accomplished through the domain subtraction operator (⩤). Essentially the domain subtraction operator removes all the tuples from a set for which the first coordinate equals the set of values to be removed. In our case it is $\{name?\}$.

A precondition of an operation may sometimes be more complicated than what needs to be, or it may be insufficient to cover all the correct operations. In such cases a precondition of an operation may be formally calculated, which is what I show next. As illustration, I calculate the precondition of schema *AddName* above.

3.3.5 Calculate the precondition of each abstract operations on the state

In Z, preconditions are predicates in operations that apply only to before states and inputs. Preconditions may be calculated by existentially quantifying the after states and outputs, and then simplifying the resulting predicate (Potter et al., 1992). As indicated, we calculate the precondition of *AddName*, called *preAddName* below.

Having existentially quantified the after states and outputs, we arrive at:

```
_preAddName______________________________________
| PhoneBook
| name? : NAME; phone! : PHONE
|____________________________
| ∃ PhoneBook′ •
| name? ∉ known ∧
| tel′ = tel ∪ {name? ↦ phone?}
|_________________________________________________
```

Z Schema 3.6 preAddName precondition schema

Next, *PhoneBook′* is expanded:

$$\begin{aligned}&\exists\, known' : \mathbb{P}\, NAME \,;\, tel' : NAME \rightarrowtail PHONE \bullet\\ &\quad known' = dom\ tel' \land\\ &\quad name? \notin known' \land\\ &\quad tel' = tel \cup \{name? \mapsto phone?\}\end{aligned}$$

Having applied Z’s one-point rule, we notice that *name*? ∉ *known* and *tel′* is accordingly updated, we arrive at *dom tel ≠ NAME.* Essentially, it means that it is possible to add a name to the phone book.

Next, a table of all the partial operations is presented.

3.3.6 Table showing all the partial operations together with their inputs, outputs and preconditions

Table 3-1 Table showing partial operations, inputs/outputs, and preconditions

Operations	Inputs/Outputs	Preconditions
InitPhoneBook	–	–
AddName	$name? : NAME$ $phone? : PHONE$	$name? \notin known$
Find	$name? : NAME$ $phone! : PHONE$	$name? \in known$
Delete	$name? : NAME$	$name? \in known$
Known	$name? : NAME$ $result! : Report$	$name? \in known$
NotKnown	$name? : NAME$ $result! : Report$	$name? \notin known$

The schemas defined above all denote partial operations in the sense that they cater only for correct input. Consequently, error conditions denoting incorrect input should be defined.

3.3.7 Definition of all schemas that present error conditions

The behavior of the *AddName* operation is only defined for correct input (i.e., a name-date pair whose name is not already in the system). We would like to extend the specification to indicate what happens when the input is incorrect. The first step is to introduce a free type which will record the outcome of the operation:

$Report ::= OK \mid AlreadyKnown$

In a standard Z specification, we then introduce a separate schema which introduces an output variable and says that the operation was successful, *Success*

$result! : Report$

$result! = OK$

Z Schema 3.7 Success schema

Using conjunction of Z's schema calculus, we can construct an intermediate specification for the partial operation to add a name together with a report of success.

$AddName1 \triangleq AddName \wedge Success$

which defines an operation *AddName1* which behaves like *AddName* but also sets an output variable to *OK*. To extend the Z specification to account for incorrect input, we define a new schema for each possible error. In this case, there is only one possible error, namely that the name is already known to the system, and we therefore define:

AlreadyKnown

$\Xi\ PhoneBook$
$name? : NAME$
$result! : Report$

$name? \in known$
$result! = AlreadyKnown$

Z Schema 3.8 Known schema

Here we have used the Z convention for describing operations which do not change the state. Under this convention, $\Xi\ PhoneBook$ is understood to have the following definition:

$\Xi\ PhoneBook$

$\Delta\ PhoneBook$

$known' = known$
$tel' = tel$

Z Schema 3.9 PhoneBook schema

A robust version of the *Find* operation must be able to report if the input name is not known:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\mathit{NotKnown}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi\ \mathit{PhoneBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{result}! : \mathit{Report} \\
\hline
\mathit{name}? \notin \mathit{known} \\
\mathit{result}! = \mathit{not\_known} \\
\hline
\end{array}
$$

Z Schema 3.10 NotKnown schema

The robust operation either behaves as described by *Find* and reports success, or reports that the name is not known:

3.3.8 Use the Z schema calculus to make all the partial operations total

The three operations *Addname, Find*, and *Delete* have been built up in a structured fashion from smaller components. This avoids any duplication of effort, allowing us to factor out common aspects of the design, and results in a clearer, more comprehensible specification. Using Z's schema calculus, we can now define total operations for the three partial operations as follows:

$\mathit{RobustAddName} \triangleq \mathit{AddName1} \lor \mathit{AlreadyKnown}$

$= (\mathit{AddName} \land \mathit{Success}) \lor \mathit{AlreadyKnown}$

$\mathit{RobustFind} \triangleq (\mathit{Find} \land \mathit{Success}) \lor \mathit{NotKnown}$

$\mathit{RobustDelete} \triangleq (\mathit{Delete} \land \mathit{Success}) \lor \mathit{NotKnown}.$

The final total (robust) operation, *RobustAddName*, can now be constructed in Z as follows:

```
_RobustAddName____________________________________________
| Δ PhoneBook
| name? : NAME
| phone? : PHONE
| result! : Report
|_______________________________
| (name? ∉ known ∧
| tel′ = tel ∪ {name? ↦ phone?} ∧
| result! = OK)
| ∨
| (name? ∈ known ∧
| tel′ = tel ∧
| result! = AlreadyKnown)
|__________________________________________________________
```

3.4 Literature review on Formal Methods

The research presented by Knight et al. (1997) addresses the question of why FMs are not used more widely. For years, academics have suggested that applying Formal Methods to software development will aid industry in achieving its aims of improving software processes and quality (Souri et al., 2019). Formal Methods are still not often used by commercial software organisations (Souri et al., 2019). The study presented in this chapter aims to examine the disconnect between research and industry and determine what steps should be taken to optimize the advantages of Formal Methods in an Agile environment for business.

Among the challenges faced were the fact that a single specification language could only define a small portion of the system, and that key tools were either unavailable, incompatible with other development tools, or too sluggish.

3.4.1 FMs in the requirements elicitation phase

The requirements elicitation phase is regarded as the most crucial and the most challenging. The consequences of getting this critical phase wrong are adverse and can persist throughout the life of the software system (Pandey et al., 2013). An adequate requirements analysis function exposes and predicts error prone areas in the proposed system

(Pandey et al., 2013). Much of the research on Formal Methods has been around the correctness of the requirement specifications (Pandey et al., 2013). This is precisely because, while gathering the requirements, requirements' notation approaches and techniques can also be developed. These will enable the formal specification and the accurate validation of requirements (Pandey et al., 2013). The challenge in ASD therefore becomes continuous variability in requirements. This adds to the complexity of the project, particularly the developers who have to continuously change the code to comply with every requirement change.

Before the benefits of formal methods can be realised, they must overcome a number of relatively minor but significant practical obstacles. While industrial methodologies like Unified Model Language (UML) are rarely formalized, they are generally well-developed and understood. Formal Methods must meet this criterion in order to be used in industrial practice (Knight et al., 1997). The evaluation is based on the necessity for each software technology, including Formal Methods, to contribute to one overarching goal: cost-effective high-quality software development. Although Knight's study did not address hardware verification, I emphasised that the successful application of FMs to hardware design is a strong indicator that comparable success with software is achievable. FM approaches such as model checkers and the Z specification are reported to have flaws such as the state space explosion problem, which consumes a lot of memory and takes a long time to solve (Knight et al., 1997). In the opinion of the researcher, using a Scrum methodology, I will be able to break the software requirements into lean business cases, then User-Stories that can be easily interpreted into Z specification language.

3.4.2 The Software Product Line Engineering

The development phase includes a lot of anticipation, and that anticipation permeates every stage of the process until the final artefact is completed. The SPLE (Software Product Line Engineering) which informs the foundations, principles and techniques of developing software, can assist in reducing complexities by providing systems with well-defined commonalities and variability. In coming up with the software product lines, software developers come up with customizable and reusable methods. The SPLE introduces a commercial approach that assists in the improvement of quality and the time it takes to development software. In a report by medical systems provider Phillips Healthcare, there was a 50 per cent improvement of quality and the time it took to produce a software product in the organisations studied (Schaefer et al., 2011). The report also claims that the introduction of SPLE resulted in a reduction of development effort up to fourfold. (Schaefer et al., 2011) and also goes further to propose that sizeable investments must be made for re-architecting, recertification, and re-verification. Modelling formalisms in different development phases must achieve very specific results. In comparison to the current state of the art in SPLE, a model-centric development process for software product lines based on a single formal modelling framework can grow into a single-source technology with numerous important advantages. Figure 3.2 below shows the model-centric development process.

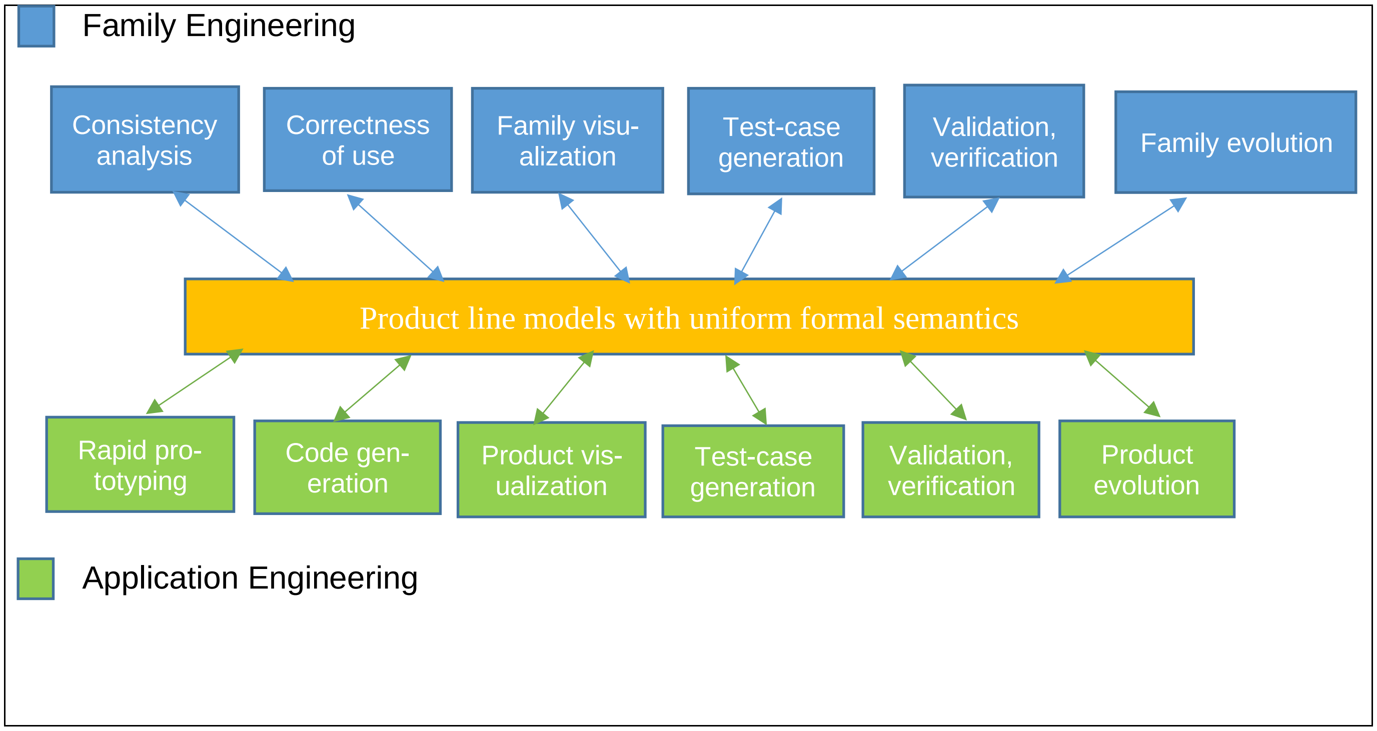


Figure 3.1 Executable models of the product line (Schaefer et al., 2011)

The product line and individual products are represented as executable models in Figure 3.2, allowing simulation and visualization tools to be utilised throughout the whole process of family and application engineering. With the upper phases of development reflecting family engineering and the bottom phases representing application engineering, the middle (orange) depicts product line models. This helps developers find and fix mistakes early in the development process and enables quick prototyping of products for interaction with stakeholders. The model is supplied data from tools for various kinds of dynamic and static analysis, automated test-case generation, model validation, and functional verification.

Formalism should also guide the selection of various development tools, particularly for requirements analysis and automation testing including functional verification (Schaefer et al., 2011). These tools will enable the formal verification at all the necessary phases, especially in the analysis phase. A proper requirements analysis can expose potential system errors and system behaviour at earlier stages of the project. Formalism offers simulation and visualization tool for the earlier revelation of the system's potential errors (Schaefer et al., 2011).

3.4.3 Specification formalism

The critics of FMs, according to Mbala et al. (2017), object to the steep learning curve involved in grasping the underlying discrete mathematics and formal logic required for effective application of the methodology. FMs, on the other hand, are critical for the control of quality parameters such as completeness, correctness, and consistency, as well as the verification of system requirements (Mbala et al., 2017), and they are based on (often discrete) mathematical notations and logic to express requirements specifications clearly and accurately. Mbala et al. (2017) developed a framework (figure 3.3) to help remove ambiguities and contradictions to some level. Such ambiguity is often present in natural language specifications.

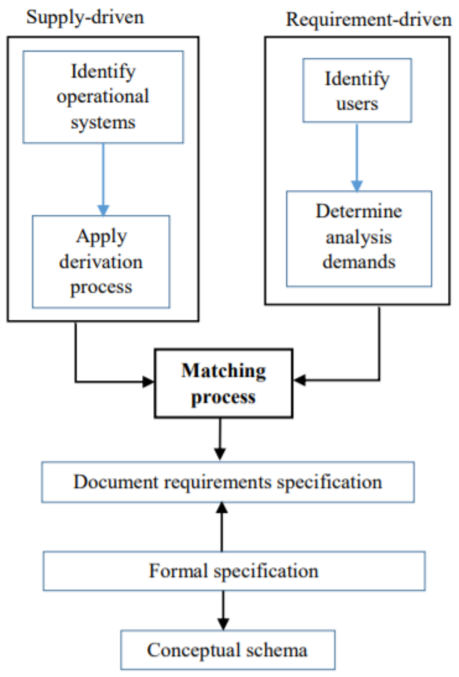


Figure 3.2 Framework facilitating the removing of vagueness or inconsistencies in system requirements

The matching of the two sets of needs generated through the supply-driven and requirements-driven procedures is the critical step in Figure 3.3 above. These needs sets could be heterogeneous and in various formats, for example, one could contain structured data (bottom-up data), while the other could contain unstructured data derived from incomplete and often inconsistent user requirements. Figure 3.3 then presents how these two streams of requirement are matched, then a single specification document is formulated. The completed requirements specification document will also include formal specifications and conceptual schema.

Because natural languages have a potential of being ambiguous, Formal Methods can be used in the elimination of requirements ambiguities (Dongmo et al., 2009). Dongmo et al. (2009) continue to draw another parallel by introducing another semi-formal verification method, Use Case Maps ‘UCMs’, which offers a simpler analysis that can be used to extract user scenarios of interaction between the system and the user. Dongmo et al. (2009) further suggest that FMs are not adopted by industry, simply because of the lack of step-by-step methodologies that embrace architectural and system boundaries.

3.4.4 Using formal verification to evaluate Human-Automation Interaction

Bolton et al. (2013) contrasted formal verification as mathematical technique that is used to appropriately prove that the Human Automation Interaction (HAI) does not demonstrate undesirable properties. Bolton et al. (2013) believes that the issues in the HAI may arise for many reasons, one of which is the automation failing as it is designed for specifically pre-determined scenarios. Automated Theorem Proving and Model Checking are the two main techniques used for formal verification in the industry (Bolton et al., 2013). In their work, Bolton et al. (2013) further argue that Model Checking is more limited than Automated Theorem Proving. In the theorem proving technique, automation is found to be reusable for continuous routine proof techniques. The authors assert that if system requirements are simplified and expressed in logical terms, the need for too much automation decreases. The major task in these techniques and the other formal verifications is that of model validation. In order to have insight into the proposed requirement's system, the model's validity must be verified. If the models themselves are invalid, the verification process is flawed and has limited chances of revealing potential software issues. Although both FMs and model checking programmes' effectiveness have been proven, there has been very little uptake in the market (Bolton et al., 2013).

These methods have been found useful in both hardware and software for their abilities in predicting failures arising from an interaction between humans and systems. Failures may go undetected during system tests and assessments because they occur under unusual and infrequent combinations of situations. Formal modelling tools and model verification technologies that explore the system's operational state space exhaustively may uncover human and sub-system interactions that result in hazardous operating scenarios. Bolton et al. (2013) further highlight two categories of failures that Formal Methods have to address:

1 active failure: are those that immediately lead to adverse consequences, and
2 latent failures: are those for which the damaging results may not become apparent until much later.

3.4.5 Incorporating FMs in testing

When introspectively analysing issues arising from the software system development, one can identify that a significant portion of them come as a result of properly specifying requirements. The root causes of defects found in system testing are as a result of requirements being unclear, imprecise, incomplete and ambiguous. If the above specifications' shortcomings are not addressed, the purpose and objective of the testing process becomes narrow.

Proper testing parameters cannot be defined and these result in a lot of rework once the system is taken to production and the end-users are not satisfied. Tretmans et al. (1999) explain that the development of test cases is a task of highly capable humans who analytically analyse the given specifications in order to come up with scenarios that will thoroughly test the expected functionality of the software system. This testing of the functional and non-functional aspects of the system requires a proper verification of the conformance to the properties specified. The Formal Methods can also be used in this analysis and checking of the correctness of the system. Using mathematical modelling of the system, formal verification can be used to prove properties and the functionality of the system.

Figure 3.4 presents how to feature formal verification in the testing phase of software development.

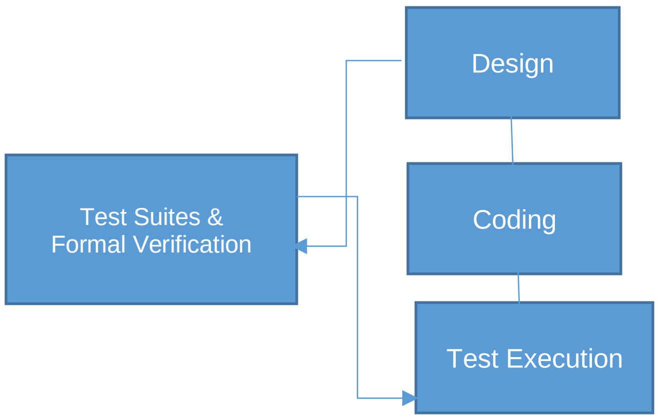


Figure 3.3 Formal verification in the testing phase of software development

In figure 3.4 Tretmans et al. (1999) presents how the formal verification can feature in a sequential software development methodology. Figure 3.4 shows the test plan consisting of test suites and formal verification scripts. The test plan is formulated from the system design and is executed in the final testing that comes after coding.

The rise of new technologies and the drive for shorter release cycles are raising the bar for software quality (Smartbear, 2018). Automation, continuous testing, and DevOps have pushed the software development lifecycle forward by adding speed and flexibility. Teams are being driven to streamline their testing and development processes in order to get more done in less time while keeping costs low in order to stay competitive. Every team aspires to achieve this mix of speed, quality and affordability, and it is also the most difficult obstacle they face. Quality assurance teams have had to make trade-offs between the three in the past. Deliver faster, but there's a danger that errors will make it to production. Ensure quality while taking a chance on meeting your deadline. When contemplating automation testing to fulfil the growing need for shorter delivery cycles and bug-free releases, it's critical to analyse whether the return on investment (ROI) is justified. Is there a way that I can incorporate FMs in this process? And can this incorporation help us quantify the ROI of FMs?
According to Smartbear (2018) manual testing, automated testing and Formal Methods aim to achieve the two main goals:

1. Reduction of Defect Leakage
2. Test Redundancy and Reusability

Software testing methods and techniques are continually evolving, and software testing research is a hot topic in software engineering. Testing is becoming more automated and integrated into Agile Software Development procedures, which include frequent builds (Huisman et al., 2020). Instead of marketing Formal Methods to industry as a "standalone" technique, these methods and processes are a logical location to gradually introduce them in industry by gradually integrating automated tool support in the testing and verification process. Another intriguing possibility is to try to integrate FMs with testing to speed up the entire verification process, for example, by utilizing Formal Methods to direct testing efforts to the 'dark corners' that such methods normally uncover (Huisman et al., 2020).

3.4.6 The myths of Formal Methods

It should be considered that although FMs are mathematical notations to describe in a precise way the properties which an information system must have (Spivey, 1998), FMs are not absolute. Hall (1990) was successful in putting together seven myths of Formal Methods that FMs' advocates are always faced with whenever I propose them. In the exercise to identify the usability of FMs within ASD, the work is focused on the below three myths.

1. Perfection can be guaranteed by implementing Formal Methods (They serve to identify errors early enough in the development process).
2. The FMs purpose is solely programme proving (They enable serious contemplation on the software system being developed).
3. The FMs are particularly for safety critical systems (They are applicable to almost all software systems).
4. They escalate development costs (They instead decrease costs).

With the above given myths, Hall (1990) clearly argues the practical benefits of Formal Methods which I will show in the case study.

3.4.7 Challenges in implementing FMs in ASD

The paper 'Formal Methods in Agile Development' concludes that Formal Methods are more challenging to implement in Agile methodologies (Lowe 2010). He continues to claim that developers are less keen to educate themselves to master FMs. However, he does identify a less chaotic Agile methodology that can be suitable for FMs. It is a process that is used for software systems enhancements where the external behaviour of the system does not change. This suggests that FMs can be used in the maintenance phase of the development when the software system is in production (Sharma et al., 2020). Once a system is released, the Formal Methods can be included in the requirements specification so that formal verification can be done for enhancements that will preserve the software's semantics. Figure 3.6 shows how the Formal Methods feature beyond the release of the system.

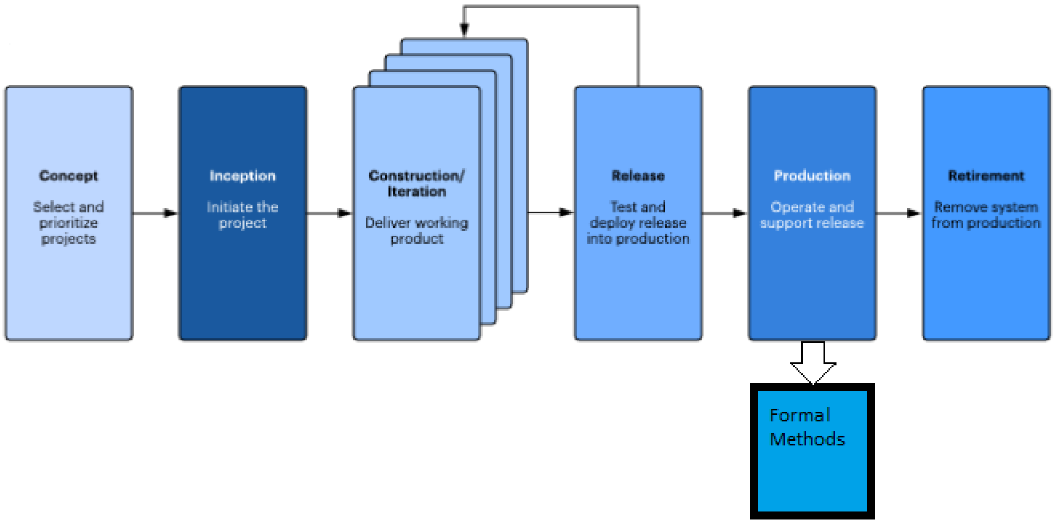

Concept
Select and prioritize projects
Inception
Initiate the project
Construction/ Iteration
Deliver working product
Release
Test and deploy release into production
Production
Operate and support release
Retirement
Remove system from production
Formal Methods

Figure 3.4 Agile Software Development process that includes Formal Methods after the developed system is in production (Sharma et al., 2020)

In any software development methodology, the release of software is accompanied by regression testing. In the regression testing, thorough testing of functionalities that existed before and after the new code is confirmed. The regression test packs are then filed and will be used whenever there are new changes to the system in production (Sharma et al., 2020). Figure 3.6 then shows the different phases in software development with the implication that Formal Methods can form part of the regression test packs that are for verifying the core functionality of the system. This is particularly because the core functionality of the software system rarely changes throughout its lifespan.

3.4.8 Are Formal Methods ready for Agile?

In all the efforts in attempting to use Formal Methods in an Agile Software Development, what then is to be gained? Particularly because of the many opposing differences that exist between the two approaches? In their work, Larsen et al. (2010) assess benefits that will come with combing FMs and ASD. They also assess the readiness of ASD to support FMs' techniques in order to have synergy in the processes. Larsen et al. (2010) identify the purpose of the Formal Methods as that of eliminating defects in complex computer systems. They also further describe FMs as a response to complexity. This response is used to analyse and model software systems as a mathematical entity. These mathematical analyses therefore enable every competent stakeholder to verify and refute aspects of the requirements specifications in all development phases. In their work Larsen et al. (2010) dispel the widely held misconceptions of regarding Formal Methods as a software development methodology on their own. Figure 3.6 from the previous sub-topic displays that FMs forms part of a methodology and is not a methodology.

Another misconception that Larsen et al. (2010) deal with is that FMs are only effective as a post-factor verification. In their arguments, they also advise against seeing Agile Software Development as a methodology that can only be implemented in a wholesale fashion. Each software development enterprise should adopt only the ASD characteristics that are suitable for their environment and their resources. Similarly, with any methodology and processes, only the applicable techniques are adopted based on the environment and sometimes the product being developed. The formalism of the requirements has to be intended at simplifying the specifications, otherwise it will be irrelevant including them in ASD which intends at rapidly completing a solution with ‘minimal documentation’. Introducing FMs must not be burdensome, and forms of static analysis and automatic verification can be used to ensure that key properties are preserved from one iteration to the next. The tools enabling FMs should also enable synergy in existing development methodology and enough research should be conducted in making this a reality.

3.5 Summary

In Chapter 3, I comprehensively discussed Formal Methods which is the highlight of the dissertation. I excavated past literature that is relevant to the work. The prevalent highlights were that the Formal Methods have mostly been used in the Traditional Software Development Methodology and they have been regarded as belonging to the analysis phase of the development methodologies (O'Regan, 2020). Software testing has also been identified as a phase to which FMs can belong (Huisman et al., 2020). However, in Agile Software Development Methodology, the lines between different phases are blurred and the principle of the methodology is to nimbly develop software without having separated phases (Larsen et al., 2010). Although there have been contemporary companies like Facebook that have adopted Formal Methods, the level of uptake remains low (Knight et al., 1997).

3.6 Conclusion

When examining this topic, it is frequently discovered that Formal Methods have long been regarded as the best technique to assist the software industry produce more dependable and trustworthy software (Huisman et al., 2020). However, many scholars agree that Formal Methods have not been well received by industry practitioners over time (Knight et al., 1997). Both the two claims are what has prompted this dissertation. With Agile Software Development being the most adopted and relevant methodology in the industry (Kim et al., 2021), merging it with the Formal Methods will enable a realization that two can complement each other and achieve efficiency and quality benefits.

In Chapter 4 I will pick an Agile Software Development based case study and formulate a framework that will see the inclusion of Formal Methods in an Agile methodology.

Chapter 4: Research Design

The research methodologies and methodology approach are presented in this chapter. First, the reader is informed about the choice of methodological approach and research design, providing information about the entire research process and its approach. Next, the reader is introduced to and given an explanation of the course of action that has been taken in the research paper. Lastly, I focus on the research study's quality and discuss its validity and reliability.

4.1 Introduction

The study design outlines the methodical steps taken to carry out the inquiry and serves as a manual for researchers as they interpret, gather, and analyse data (Saunders et al., 2018). The study must be thoroughly recorded in order to be successful. The purpose of the research design is to offer a suitable framework for a study. There are other interconnected decisions that must be taken but selecting a research strategy is an important stage in the research design process since it defines how pertinent data for a study will be obtained (Saunders et al., 2018). This research work was conducted in line with (Saunders et al., 2018) research onion (Figure 4.1) and all the selected options are briefly discussed below.

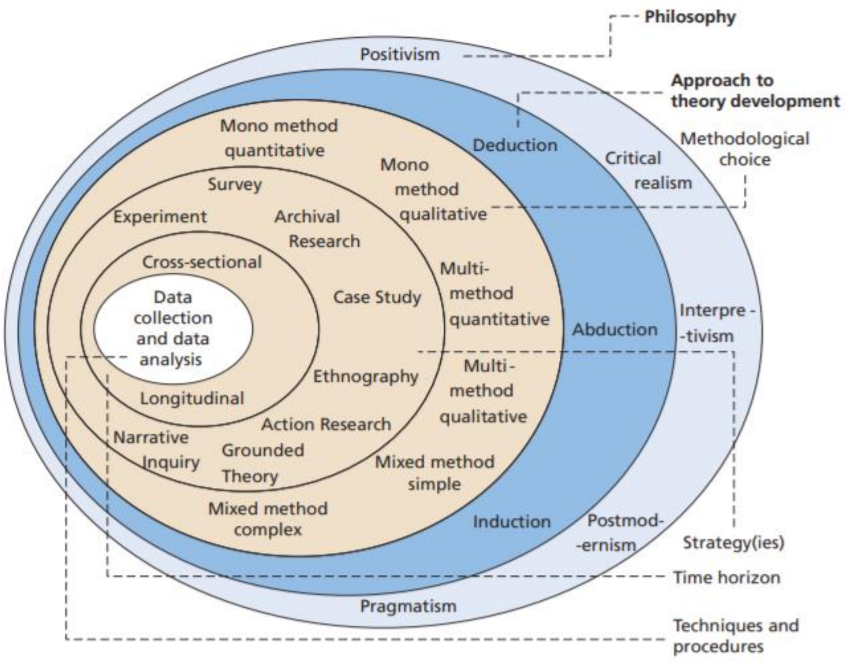


Figure 4.1 Universal research onion (*Saunders et al., 2018)*

The research philosophy had elements of interpretivism as well as positivism. The qualitative nature of Agile as a methodology involves interpreting natural language requirements expressed by users of the system. Formalizing aspects of Agile through the use of FMs gives the research also a positivist philosophy.

Turning to the second layer of the onion, the research approach was a mixed abductive and inductive approach – inductive in the sense that a framework was constructed, and deductive since the framework was validated by applying it to a case study (Arnold et al., 2020). The research strategy was that of a case study. In a similar vein, we looked at a Scrum case study to determine how FMs could be embedded.

The time horizon will be cross sectional since the research will be completed within a fixed time period, looking at the literature and FMs cases at a point in time (Arnold et al., 2020). Data collection will be through scholarly literature and the researcher's knowledge of the subject.

4.2 Philosophy - Positivism

The research philosophy enables the improvement of the comprehension and application of the theory to practice, and presentation of the research findings (Alharahsheh et al., 2020). This section explains what a paradigm is before delving into and debating the assumptions that underpin scientific and interpretive paradigms (Alharahsheh et al., 2020).

The positivist paradigm allows researchers to rely more heavily on statistics and generalization, which leads to the formation of universal laws and discoveries (Alharahsheh et al., 2020). Positivism is based on a scientist's philosophical stance when working with observable reality in society, which leads to the formation of generalizations. Using such, I use the Scrum guide which sets out the rules of the game and therefore provides the observable insight into Scrum. A tighter emphasis is placed on pure data and facts that are unaffected by human interpretation and bias in positivism, which is focused on the value of what is presented generally (Saunders et al., 2018).

4.3 Approach to theory development - Abductive and inductive (hybrid)

Inductive theorizing is theorizing that starts off with non-theoretical empirical phenomena, which should ideally result in a proposed or supported theory. In inductive theorizing, the researcher starts from empirical data and works towards developing a theory based on that data (Okoli, 2021). In this paper, I use the Scrum guide as the empirical framework in which Formal Methods can be used. As a result, I will formulate a theory in which FMs can assist Scrum in achieving quality and efficiency.

Abductive theorizing is theorizing that starts off with a rudimentary theory or theory-in-progress, which should ideally result in a proposed or supported theory (Okoli, 2021). Note that contemporary philosophers use the term “abduction” for a similar but distinct kind of reasoning normally called “inference to the best explanation”; that is, abductive reasoning now usually refers to considering a specific case and then attempting to infer the most likely rule that would explain that case (Douven, 2011). During the literature review chapter in this paper, the researcher discovered how limited the literature is on

Formal Methods being applied to Agile Software Development in general. Therefore, the starting point of the research was rudimental.

4.4 Methodological choice – Qualitative

This study employs a qualitative, exploratory case study to see where and how Formal Methods might be included into Scrum. The goal of an exploratory case study is to look at a phenomenon in the form of a causal relationship that hasn't received much attention. (Moi et al., 2021). The researcher's observation is that although FM's and Scrum have been widely researched individualistically, the two subjects have not been researched together. Because the construct of Formal Methods is generally an unexplored phenomenon in Agile Software Development, I decided to use exploratory research in this study because it serves as a prelude to qualitative research. The study's exploratory character aims to inform software engineering practice in terms of efficiency and quality, a concept that currently lacks a well-established theory. Furthermore, utilising an exploratory technique, I can better answer the study questions. Overall, the qualitative technique allowed us to gain a comprehensive and in-depth understanding of software practitioners' perspectives on Scrum, the framework that governs the end-to-end software development process.

4.5 Research strategy - Case study

The methodological approach used in this Scrum research was a case study. This is an empirical study of an ongoing event in a situation where it might be difficult to distinguish between phenomena and context (Cui et al., 2021). The case study was conducted at a banking institution where Scrum is used as a software development technique of choice. The reason it was conducted here, was that banking has become technology driven and their services are almost entirely technological. Banks have capacitated software development teams which adopt latest techniques and technologies for efficient and quality software output.

The process of implementing software, its potential impact on the transformation's future, and the strengths and weaknesses of the present software product delivery process were all worth looking into. The bank is the subject of the case study because it always seeks to enhance its procedures. Furthermore, the study will be generalizable so that the same techniques are applicable to other industries.

Case studies may be categorised into three categories: explanatory, exploratory, and descriptive (Cui et al., 2021). Since the case study has no predefined goal and a desire to get a comprehensive and in-depth understanding of the Agile process, it has been carried out utilising the exploratory technique.

4.6 Data collection – Observation

A qualitative research technique called observational research includes seeing and evaluating the target responder or subject in a real-world or natural setting (Rasch et al., 2020). The empirical study's initial phases and observations seek to add additional details to subsequent research processes and provide a comprehensive viewpoint (Rasch et al., 2020). Due to its full implementation of the Scrum approach according to the Agile Software Development Methodology in a typical setting, the observed case study in the research was selected. This is done in an effort to record as much information as possible about the Scrum environment and the development team's participation in order to determine how Formal Methods may be incorporated.

The observations took place at different Scrum events: the Sprint, Sprint Planning, daily Scrum, Sprint Review, and Sprint Retrospective session. All the observed events are thoroughly discussed in Chapter 2, table 2.2. The plan was to observe the Scrum Team during their everyday tasks, with the aim to analyse the everyday situations and gain understanding about how the process and software output could be improved with Formal Methods.

4.7 Trustworthiness

Trustworthiness refers to the reliability of the findings across time, from various viewpoints, and throughout the research process itself (Rasch et al., 2020). Due to the nature of qualitative research, it might be difficult to ensure reliability because the target's perspective or opinions may vary over time and the data mostly comes from case studies based on personal experiences. Thus, five essential concepts—credibility, transferability, dependability, confirmability, and authenticity—are examined in order to assure the research's reliability (Rasch et al., 2020). These ideas are based on the well-known qualitative research quality standards system, and by definition the study will be regarded as highly dependable if all five criteria are met.

4.8 Credibility

The credibility refers to whether or not the findings in the study are representative of the subjects' own experience (Rasch et al., 2020). Therefore, to ensure that this study is genuine, it was evaluated by peers and verified by a specialist in the subject. Peer review occurs often while the dissertation is enrolled, offering revisions at every level of the procedure along with comments, direction, and supervision from Prof. John Andrew (Andre) van der Poll and Prof. Hugo Lotriet.

4.9 Confirmability

This section explores whether the researcher's bias in any way influences the conclusions. Given that research biases may influence how sources are presented and used, it is a crucial part of the source dependability section (Rasch et al., 2020). As the investigation progresses and more is learned about the subject, there is a risk that the abductive aspect of the research will skew the case study that is undertaken later in the study. The Scrum guide, which serves as a framework for the implementation of the Scrum approach, has in some ways addressed it.

Unstructured observations have a significant drawback since the researchers decide what to observe and how to evaluate and process the data, making them extremely vulnerable to observer bias (Rasch et al., 2020). The observer's expertise and experience will have an impact on the empirical quality and findings. This is something that may raise worry for the dependability of the study and will have to be evaluated during the empirical collecting and analysis.

4.10 Authenticity

If the issue has been examined from a representative spectrum of opposing views and if the findings have the potential to change lives, the study is real (Rasch et al., 2020). This is mostly addressed by the real-life case study within the banking institute. Since the pre-study is an analysis from one organisation’s case study, it could lead to a problem formulation that is biased towards other industries. This is somewhat addressed by the study's iterative design and frequent re-evaluations of the questions and problems that were initially posed.

4.11 Validity

Validity is about accuracy if dependability is about consistency. The validity relates to how effectively the findings measure the target variable (Rasch et al., 2020). Lower validity is typically caused by bad study design and bad research methodology. If the study is to be regarded as genuine, it is vital that the choice of research technique adequately represents the research question and that the research question truly delivers the anticipated outcomes (Rasch et al., 2020). To improve the study's validity, steps might be done like triangulation. Nevertheless, there is no way to completely ensure veracity (Moi et al., 2021). Since the goal of interpretative research is to compile interpretations and explanations for a certain occurrence, high validity frequently results from these investigations.

4.12 Summary

The research design is painting a picture of how the research is conducted using the case study as the research strategy. I also presented the philosophies that guide us, the qualitative nature and how observations are used in the collection of data. All the choices I used were selected through the guidance by the research onion by Saunders et al. (2018). Another big consideration in this chapter are the quality controls that come with ensuring that the work is reliable, credible, transferable, dependable, conforms, is authentic and is valid.

4.13 Conclusion

As a guide for the researcher in Chapter 4 I presented the research design which will assist us in interpreting, collecting, and analysing data from the contributing chapters. The research design described how the investigation was conducted in a systematic manner (Saunders et al., 2018). Having a clear picture of how research is being conducted enhances the chances of its success. The research design provided an appropriate framework for this study.

In the next chapter, I present a case study that demonstrates how Scrum is practiced on a day-to-day basis.

Chapter 5: The Dlamini Bank Case Study

In Chapter 4 I discussed the research design. I outlined the methodology approach and the research techniques. First, I outlined the methodology I used and the study design which informed the reader about the entire research process. The methodology then introduced and described the technique used in the research report. Lastly, I focused on the research study's quality and discussed its validity and reliability.

In Chapter 5, I show an Agile Software Development case study. The ASD case study will emphasise the everyday functions of Scrum processes towards fulfilling a requirement specification of an ATM system. I will expand the banking requirement specification into more User-Stories and identify in which software development phases Formal Methods can be efficient and how will business enterprises benefit from embedding FMs in Agile Software Development. A brief methodology for embedding FMs in the Agile development process is presented.

5.1 Introduction

The case study selected is an appropriate method for addressing the balance between the efficient way of developing software and ensuring quality software output (Hilburn et al., 2020). The qualitative case study method that I am presenting is not aimed at analysing the case, but it is a good way to define the case and to explore a setting in order to understand the properties of the specific group and environment under study. It is well suited to software engineering research and provides a deeper understanding of the phenomena under study (Gustafsson, 2017). The case study that I was formulating in this report explores a real-life experience in the life of a Scrum Team.

5.2 The case study – Agile Software Development (Scrum)

Developers may handle difficult adaptive challenges with the Agile methodology while producing high-value solutions in a productive and innovative manner (Zayat et al., 2020). The Scrum framework is made up of Scrum Teams and all of the roles, events, artefacts and rules that go along with it. Each element of the framework has a distinct function and is necessary for Scrum to function and be used. In the continuation of a brief Scrum ATM cash withdrawal example, the case study will show a broadened Sprint scenario with more focus on the Product Backlog and Sprint Backlog (see figure 5.1).

Figure 5.1 provides an overview picture of a full Sprint. The Sprint is a piece of a project broken down into a two-week iteration. The product backlog becomes the broken-down piece of requirement that the Development Team will convert into a Sprint Backlog and finish within the two weeks' time box (Bhavsar et al., 2020).

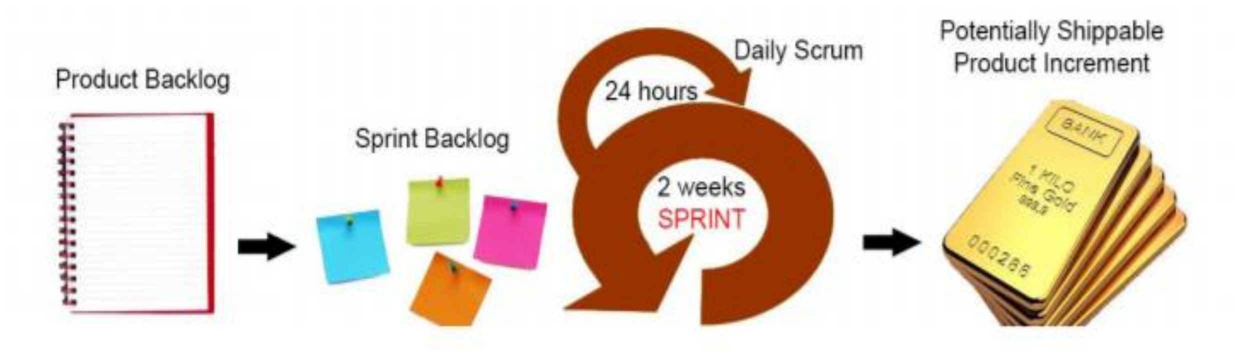


Figure 5.1 Scrum process flow (Bhavsar et al., 2020)

An example of a work item's Scrum process flow is shown in Figure 5.1 above. Work items are initially put into the Product Backlog. The Development Team adds work items to the Sprint Backlog with the intention of completing the Sprint's requirements, or DoD (Definition of Done) as it is known in Scrum. Completed work items are checked off during the Sprint Review and subsequently delivered as a product increment (Bhavsar et al., 2020).

Next, I define the case study to be used in this chapter.

5.3 Dlamini Bank case study

As a new banking institution, Dlamini Bank is rolling out ATM systems that will enable their client to have access to banking functionality in more convenient locations. The concept is given to Mduduzi, the Scrum Product Owner, for this software development project by the decision-makers (stakeholders) of the Bank. He has to start requirements engineering as one of his first jobs. He discusses the most crucial use cases with the architects, customers, and other stakeholders before noting them down (figure 5.2). The architects provide technical direction that the development team can follow, while customer representatives and other stakeholders continuously provide requirements' clarification.

After gathering the high-level use cases and requirements, he enters them into the Scrum Product Backlog (an understandable and general product backlog is shown in Figure 5.3) and starts a session with the architects and some senior engineers to estimate and prioritize the items. All of the items in the Scrum Product Backlog now have an initial rough estimate and a priority as a consequence of this session. The high-level requirements are then divided into smaller-grained User-Stories. He then schedules the first Sprint Planning meeting with the Scrum Team using this list. Figure 5.2 presents Dlamini Bank ATM's use cases which are goals that the ATM user is intended to achieve.

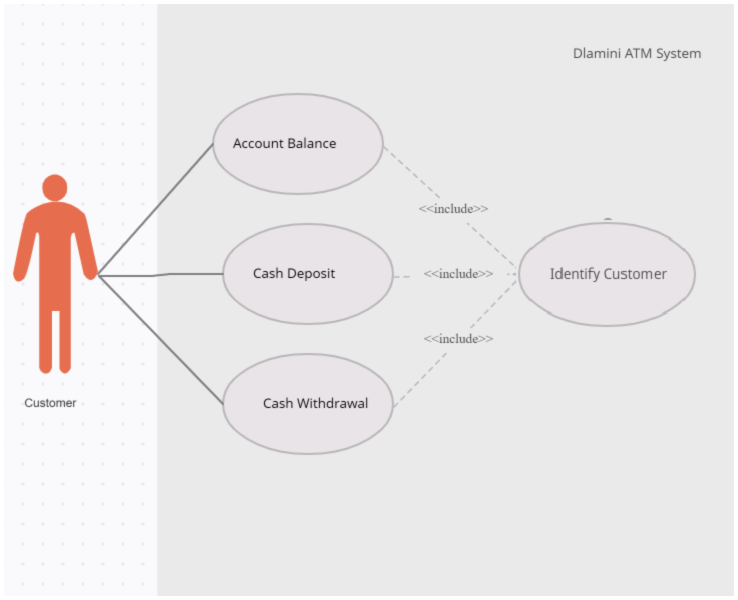


Figure 5.2 Dlamini Bank ATM system's initial use cases (Features)

Figure 5.2 above presents a Dlamini ATM system use case diagram. The diagram consists of an actor who represents the customer using the ATM. It also presents four goals that the customer intends to achieve. The ATM system will be a tool for the user to be able to check account balances, deposit cash and withdraw cash. All the mentioned three goals will be achieved after the 'Identify Customer' is achieved.

Next, I present how a full Sprint is run as shown in figure 5.1. I present a practical case study that shows all the tasks undertaken from Day 0 which represents Sprint Planning day, until day 28 which is typically an end of the Sprint which never takes longer than one month (Zayat et al., 2020).

5.4 Sprint 1 – Day 0 (S1.0)

The Scrum Master decides who will be on the team and calls a meeting, having invited these people.

A Sprint Planning meeting is usually indicated by Day 0. Mduduzi lists the Scrum Product Backlog items during the Sprint Planning meeting in order of highest importance to lowest (Table 5.1). At this stage, all the User-Stories have no status as none of them have yet been undertaken. The team examines each item in a Sprint Planning session to see whether they have the necessary capacity, expertise, and resources. The team also resolves any unresolved questions. Figure 5.3 represents how the development team gathers in front of the Sprint Backlog board for a 15-minute daily Scrum meeting.

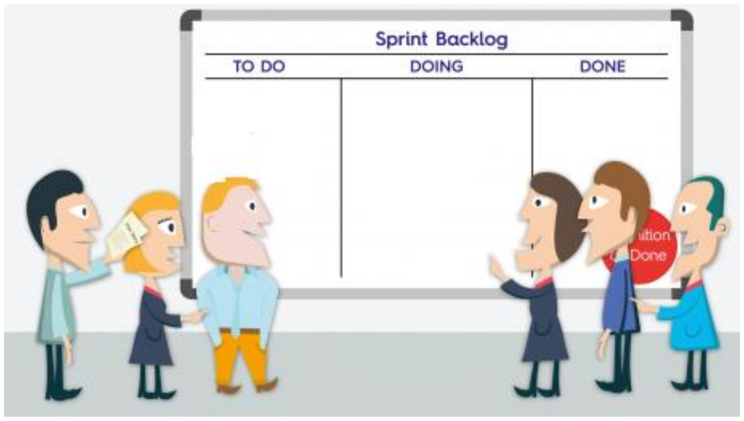


Figure 5.3 Illustration of a Development Team gathered around the Sprint Backlog board (https://watisscrum.nl/sprint-backlog/)

Since this is the beginning of a Sprint, the Sprint Backlog board is empty. Once the team agrees on taking up the User-Stories in Table 5.1, the User-Stories will show on the To do column of the board.

Embedding Formal Methods in Scrum, I formally specify the contents of the Sprint backlog board as follows:

I start by defining the basic types of the Sprint Backlog specification:

[*To do*, *Doing*, *Done*]

Next, I define the state space of the backlog board:

Sprint_Backlog

to do: *To do*
doing : *Doing*
done : *Done*

disjoint (*to do, doing*) ∧ disjoint (*to do, done*) ∧ disjoint (*doing, done*)

Z Schema 5.1 Sprint Backlog

The declaration part of the schema introduces 3 variables: *To do*, *Doing*, and *Done.* The variables represents the columns in Figure 5.3. The predicate presents the pairwise disjoint sets (since the same content cannot appear in multiple columns) where a Sprint Backlog item can move from *to do*, to *doing* (when it is in development), to *done* (when it is completed).

Following Z's Established Strategy, the next step is to define an initial state of the board and subsequently show that such a state can be realized (van der Poll and Kotze, 2005)

InitSprint_Backlog

Sprint_Backlog′

to do = ∅ ∧ *doing* = ∅ ∧ *done* = ∅

Z Schema 5.2 Initial Sprint Backlog

Z Schema 5.2 represents the initialisation of the Sprint Backlog where the three columns *To do*, *Doing*, *Done* are still empty sets.

Proof:

$\vdash Sprint\_Backlog' \bullet InitSprint\_Backlog$ (5.1)

(5.1) presents that a state can be realised such that it satisfies the requirements of *InitSprint_Backlog*. I then need to show:

$$\vdash \exists\ to\ do' : To\ do;\ \ doing' : \mathrm{Doing};\ done' : \mathrm{Done} \mid$$
$$to\ do' = \emptyset \land doing' = \emptyset \land done' = \emptyset \qquad (5.2)$$

The proof of (5.2) follows trivially since the empty set values are specified in schema *InitSprint_Backlog*. The proof indicates there is indeed an initial state from which the system may start. The proof follows trivially, since the empty set values are specified in schema *InitSprint_Backlog*.

Next, the team divides the use cases into ten (10) User-Stories 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10. The ten User-Stories can be tracked and managed as initially presented by the Product Owner in Table 5.1.

Table 5-1: Sprint Backlog User-Stories

Number	**User-Story**
1	*Insert Bank Card*
2	*Insert Deposit Account Number*
3	*Read Card Pin*
4	*Verify Customer And Show Personalized Welcome at Landing Page*
5	*Show Options to Deposit and Withdraw*
6	*Show Available Balance After Withdrawing Option*
7	*Enter Deposit Amount*
8	*Enter Withdrawal Amount*
9	*Push-out Cash and Show Available Balance*
10	*Confirm Transaction Success and Printout*

Given Table 5.1, I define a next basic type:

[*USER_STORIES*]

A state space of Table 5.1 is given in Schema 5.3.

```
_User_Stories___________________________________________
| stories: ℕ₁ ⇸ USER_STORIES
|________________________________________________________
```

Z Schema 5.3 State Space for User-Stories

Z Shema 5.3 presents the numbering of user stories as listed in table 5.1.
Next, I populate the user stories as indicated in Table 5.1 into Z Schema 5.3.

```
_AddUserStories_________________________________________
| Δ User_Stories
|_____________________________
| stories' = {1 ↦ "Insert Bank Card",
|            2 ↦ "Insert Deposit Account Number",
|            3 ↦ "Read Card Pin",
|            4 ↦ "Verify Customer And Show Personalized Welcome at Landing Page",
|            5 ↦ "Show Options to Deposit and Withdraw",
|            6 ↦ "Show Available Balance After Withdrawing Option",
|            7 ↦ "Enter Deposit Amount",
|            8 ↦ "Enter Withdrawal Amount",
|            9 ↦ "Push-out Cash And Show Available Balance",
|           10 ↦ "Confirm Transaction Success And Printout" }
|________________________________________________________
```

Z Schema 5.4 Adding User-Stories

The above Schema 5.4 *AddUserStories* accesses Schema 5.3 *User_Stories* and adds the 10 User Stories and maps them sequentially.

The formal specification of Table 5.1 could have followed either of two routes:

1. Initialise the component *stories* as an empty function, followed by a proof that such an empty set can be realised, followed by an operation like *AddUserStories,* or
2. Specify *AddUserStories* directly as I have done above.

Having considered the User-Stories in Table 5.1 and Schema 5.4, the team indicates they do not have the capacity to complete User-Stories 5 and 6 in the 1st Sprint. Consequently, Scrum *Product Owner* Mduduzi agrees to move them to the 2nd Sprint.

Table 5.2 illustrates how Sprints and User-Stories can be tracked and managed as initially presented by the Product Owner and change on the backlog board, having moved User-Stories 5 and 6 to the 2nd Sprint.

Table 5-2: Tracking prioritization of User Stories towards the satisfaction of customer requirements

User-Stories	***Sprint Ready***	***Priority***	***Status***	***Sprint***
1	*Yes*	*Medium*	*To do*	*1*
2	*Yes*	*Medium*	*To do*	*1*
3	*Yes*	*High*	*To do*	*1*
4	*Yes*	*Low*	*To do*	*1*
5	*No*	*High*	*To do*	*2*
6	*No*	*Medium*	*To do*	*2*
7	*Yes*	*High*	*To do*	*1*
8	*Yes*	*High*	*To do*	*1*
9	*Yes*	*Low*	*To do*	*1*
10	*Yes*	*Low*	*To do*	*1*

Next, I formally specify Table 5.2

Prioritization_User_Stories

$stories: \mathbb{N}_1 \rightarrowtail\!\!\!\!+ \; Sprint\_Ready \times Priority \times Status \times Sprint$

Z Schema 5.5 State Space User-Story Prioritization

Z Schema 5.5 shows the prioritization of the User Stories. In the schema, $\mathbb{N}_1$ represents set of strictly positive numbers showing the numbering of User Stories, where each one of the has attributes for *Sprint_Ready, Priority, Status, Sprint*. As indicated with the partial function symbol, the Sprint readiness, status, Sprint are determined for every User Story.

Next, populate the schema in accordance with the information in Table 5.2:

```
_Populate_Prioritization_User_Stories________________________
| Δ Prioritization_User_Stories
|______________________________
| (∀i : ℕ₁ •
|   stories(i).Status = "To do"
| ∧
|   (if i ∈ {5, 6} then stories (i).Sprint_Ready = "No"
|    else stories (i).Sprint_Ready = "Yes")
| ∧
|   (if i ∈ {1, 2, 6} then stories (i).Priority = "Medium"
|    elseif  i ∈ {5, 7, 8} then stories(i).Priority = "High"
|    else stories(i).Priority = "Low")
| ∧
|   (if i ∈ {5, 6} then stories (i).Sprint = "2"
|    else stories (i).Sprint = "1")
|______________________________________________________________
```

Z Schema 5.6 User-Story Prioritization

Z used not to have an "*if … then … else"* … construct, but it was added in the 2nd edition of the Z user manual (Spivey, 1998) to facilitate the user experience (readability, usability) of Z. In the above I further extended the syntax to include an "*elseif*" as indicated.

Considering the differences between Table 5.1 with underlying schema 5.1 and Table 5.2 with schema 5.2 I, note that the "*To do*" was transformed from an attribute in Table 5.2 (a component in Schema 5.1) to a mere value of an attribute "*Status*" in Table 5.2 (cf. a value in Schema 5.2). Such transformation was elicited further through the formal specification; it might not readily have been observed in the Agile specification and may be a source of ambiguity in subsequent system design.

Using Formal Methods, in the following section I verify the three (3) high level ATM banking system use cases (Figure 5.2) which were also devised into Agile's User-Stories (Table 5.2).

5.5 Feature 1 – Account balance

In reference to the Sprint Backlog board (Table 5.2), the following User-Stories fall under feature 1.

- User-Story 1: Bank Card Insertion
- User-Story 2: Card Pin Reading
- User-Story 3: Verify Customer and Show Personalized Welcome at Landing Page
- User-Story 5: Show Available Balance After Withdrawing Option (Before Withdrawal)
- User-Story 8: Show Available Balance After Transacting

5.5.1 User-Story objective

As a Dlamini Bank customer:

I want to be presented with my bank balance on inserting my bank card and pin number at an ATM,

So that I can immediately know how much to withdraw.

5.5.2 Acceptance criteria

1. Customer needs to have inserted a bank card and pin on the ATM.
2. Customer needs to have been validated as an existing customer.

The below is a state schema for the ATM banking system. In the schema, I show customers' accounts and balances. I am introducing the set of all accounts and balances as basic types of the specification:

[*ACCOUNT*, *BALANCE*].

$$\begin{array}{|l}
\multicolumn{1}{l}{\_ATM\_Banking\_\_\_\_\_\_\_\_\_\_} \\
known : \mathbb{P}\ ACCOUNT \\
atm : ACCOUNT \nrightarrow BALANCE \\
\hline
known = \operatorname{dom} atm \\
\hline
\end{array}$$

Z Schema 5.7 ATM Banking State Schema

Recall from Chapter 3 that Z Schema 5.7 consists of a central dividing line, in which variables are declared, and a part below the line which gives a relationship between the values of the variables. In this case I am describing the state space of a system, and the two variables represent important observations which I can make of the state:

- *known* is the set of accounts in the banking system;
- *atm* provides functions which, when applied to certain accounts, returns their balances.

One possible state of the system has three people in the set known, with their balance recorded by the balance function:

$$\begin{aligned} &known = \{Khulekani, Bheki, Phumlani\} \\ &Balance = \{Khulekani \mapsto R800, \\ &\qquad Bheki \mapsto R2000, \\ &\qquad Phumlani \mapsto R400\} \end{aligned}$$

Note that I use people′s names as the domain element rather than account numbers as indicated in the state space (ATM_banking) of this system. This is done to avoid complexities of account numbers.

5.6 Feature 2 – Cash deposit

With reference to the Sprint Backlog board (Table 5.2), the following User-Stories fall under feature 2.

- User-Story 2: Insert Deposit Account Number
- User-Story 4: Verify Customer and Show Personalized Welcome at Landing Page
- User-Story 7: Enter Deposit Amount
- User-Story 10: Confirm Transaction Success

5.6.1 User-Story Objective

As a Dlamini Bank customer:

I want to deposit cash into my bank account at an ATM

So that I do not have to wait for bank’s branch working hours.

5.6.2 Acceptance criteria

1. Customer needs to enter a valid account to deposit cash.
2. System needs to validate the existence of the account number.
3. System needs to give the customer an option to enter the amount to be deposited.

Z Schema 5.8 presents a cash deposit formal specification which describes the intended system behaviour. Having the system formally specified has helped eliminate the obscurity that lies between the system requirements defined purely with natural language and the actual functionality of the specified system.

CashDeposit

$\Delta\ ATM\_Banking$
$account? : ACCOUNT$
$deposit? : BALANCE$
$receipt! : RECEIPT$

$account? \in known \Rightarrow$
$(\exists\ balance' : BALANCE \bullet balance' = atm(account?) + deposit? \wedge$
$atm' = atm \oplus \{account? \mapsto balance'\})$
$receipt! = deposit?$

Z Schema 5.8 Cash deposit

The declaration *Δ ATM Banking* alerts us to the fact that the schema is describing a possible state change: it introduces variables known as components in Z: *known*, *atm*, *balance*, *known!*, *atm!,* and *balance!*. The first three are observations of the state before the change, and the last three with output operations (!) are observations of the state after the change. It must be true both before and after the operation since every pair of variables is implicitly restricted to meet the invariant. The declarations of the three operation inputs follow. The names of inputs often terminate with a question mark. The new balance following the deposit is specified accordingly.

As I had done in the ATM_Banking schema, below I also present how the Cash Deposit schema picked up specification issues that would have been missed by User-Stories and other natural language requirements.

5.7 Feature 3 – Cash withdrawal

In reference to the Sprint Backlog board (Table 5.2), the following User-Stories fall under feature 3.

- User-Story 1: Bank Card Insertion
- User-Story 3: Card Pin Reading
- User-Story 4: Verify Customer and Show Personalized Welcome at Landing Page
- User-Story 5: Show Options to Deposit & Withdraw
- User-Story 6: Show Available Balance After Withdrawing Option
- User-Story 8: Enter Withdrawal Amount
- User-Story 9: Push-out Cash and Show Available Balance
- User-Story 10: Confirm Transaction Success and Printout

5.7.1 User-Story objective

As a Dlamini Bank customer,
I want to withdraw cash from my bank account through an ATM
So that I can have physical access to my banked money in more locations and at any time.

5.7.2 Acceptance criteria

1. Customer needs to have inserted a bank card and pin on the ATM.
2. System checks to see if the requested amount exceeds the balance.
3. If so, the system displays the balance and asks the user to enter a new amount.
4. If amount entered is less than the account balance, cash is dispensed, and the new balance is displayed.

Z Schema 5.9 presents a cash withdrawal formal specification which describes the intended system behaviour.

CashWithdrawal

$\Delta\ ATM\_Banking$

$pin? : PIN$

$withdrawal? : BALANCE$

$receipt! : RECEIPT$

$withdrawal? \leq atm\ (account?)$

$pin? \in known \Rightarrow$

$(\exists\ balance' : BALANCE \bullet balance' = atm(account?) - withdrawal? \land$

$atm' = atm \oplus \{account? \mapsto balance'\} \land$

$receipt! = balance')$

Z Schema 5.9 Cash withdrawal

The declaration *Δ ATM Banking* alerts us to the fact that the schema is describing a possible state change: it introduces variables: *pin, balance, receipt, pin?, withdrawal?, and receipt!*. The schema represents a withdrawal scenario, where a known user is presented with their account's available balance after being verified. This then proceeds them with a function to withdraw from the available balance and finally be presented with a receipt showing their transaction and the remaining balance.

In table 5.1 above, the User-Stories are broken down into tasks and are allocated into Sprints. These allocations are also based on the prioritization of the tasks and are a piece-meal towards the achievement of a working software. The Sprint Ready and Sprint columns show that except for User-Stories 5 and 6, the rest of the User-Stories must be done in the allocated 28 days.

Sprint 1 – Day 1 (S1.1)

The next Sprint meeting is labelled as Day 1. The team gathers for their daily Scrum meeting in the morning (figure 5.3). Everyone summarizes what has been accomplished so far, updates the estimated number of hours left on the cards of the Sprint task board, discusses what developers are going to work on that day, and mentions any impediments preventing them from continuing their work. (refer to Section 2.4.5: A day in Scrum Environment).

One of the team members shares with the group that one of the software tools he is using requires a new licence, which presents a problem. To see whether other team members are experiencing the same issue, Scrum Master Thembi asks them if they want her to address it after the meeting. Everyone resumes working on their tasks after a fifteen-minute session. Figure 5.4 represents how the development team gathers in front of the Sprint Backlog board for a 15-minute daily Scrum meeting. Note that the content of the Sprint Backlog board in Figure 5.4 is abstract and generic. An example of a backlog board with specific content appears in Table 5.2.

Figure 5.4 Illustration of a Development Team gathered around the Sprint Backlog board (https://watisscrum.nl/sprint-backlog/)

In figure 5.4 the team and the Scrum Master keep progress of all the User-Stories allocated for the duration of the Sprint. In the daily Scrum meeting, the individual members of the development team announce what User-Stories they are currently doing and if there are any impediments hindering their progress. In the beginning of the Sprint (Day 1), the Done column is empty until a User-Story is completed. After the meeting Thembi updates the Sprint Backlog board by moving the User-Stories' artefacts between To do, Doing and Done. Then she follows up and removes all the impediments that affect the team's progress following the Sprint 1 – Day 1 meeting.

Sprint 1 – Day 2 (S1.2)

In the morning of Day 2 the whole team meet again for their daily Scrum meeting. After the meeting, the team disperses and attends to the tasks from the Sprint Backlog that they had committed to. In the afternoon one of the Scrum Team members is unsure whether both the ATM withdrawal and deposit functions must print out a customer's balance. Thembi then calls Mduduzi the Scrum Product Owner and discusses these options. Mduduzi then clarifies that only the withdrawal must print out the balance. After that the team member knows how to proceed; he can continue with the development. Table 5.3 shows how the development team documents the progress for the allocated User-Stories in the Sprint Backlog board.

Table 5-3 User-Stories details as illustrated in Sprint Backlog board

To do	***Doing***	***Done***
5. Show Options to Deposit and Withdraw	*3. Card Pin Reading*	*1. Bank Card Insertion*
6. Show Available Balance After Withdrawing Option	*4. Verify Customer and Show Personalized Welcome at Landing Page*	*2.Insert Deposit Account Number*
	7. Enter Deposit Amount	
	8. Enter Withdrawal Amount	
	9. Push-out Cash and Show Available Balance	
	10. Confirm Transaction Success and Printout	

In Table 5.3, the Scrum Master and development team document and keep track of their progress towards completing all the items allocated for the Sprint. In the Sprint Backlog board, individual members of the team move the User-Stories they are working on from the To do column to Doing column. Once, they have completed the User-Story, they move it from Doing to Done. This is performed continuously until the Sprint is finished. In this case study, for instance, this is repeated until day 28 of the Sprint.

```
_Sprint_Backlog_User_Stories____________________________
| Δ Sprint_Backlog_User_Stories
|-----------------------------------------
| (∀i : [1 .. 10] •
|   if i ∈ {5, 6} then stories (i).Status = "To do"
|   elseif  i ∈ {3, 4, 7, 8, 9, 10} then stories(i).Status = "Doing"
|   else stories(i). Status = "Done")
|__________________________________________________________
```

Z Schema 5.10 Sprint_Backlog_User_Stories Statuses

Z Schema 5.10 shows the Sprint Backlog User-Stories statuses. The possible statuses on the backlog (Figure 5.4) are ‘To do’, ‘Doing’, and ‘Done’. In the Schema, I show using a condition statement that User-Stories 5 and 6 are still on the ‘To do’ status, while all of User-Stories 3,4,7,8,9 and10 are in the ‘Doing’ status. The other User-Stories are in ‘Done’ status.

Sprint 1 – Day 28 (S1.28)

Thembi has invited the team to the Sprint Review Meeting on this last day of the first Sprint. To show Mduduzi, the team has planned a demonstration. Mduduzi then sits in front of the computer to determine whether the created user stories fulfil his requirements and whether the features are properly documented. At the end of the Review Session, he concludes:

- *User-Stories 1,2,3,4,8, and 9 are finished as expected.*
- *User-Stories 5 and 6 were put on hold for Sprint 2.*
- *User-Stories 7 and 10 couldn't be finished in time because of the number of defects open.*

Table 5.4 shows the open defects that have resulted in User-Story 7 and 10 not to be completed.

Table 5-4 Open Defects List

Defect ID	***Description***	***Status***	***User-Story***
#15	*User is able to enter deposit amount manually, instead of the ATM determining deposit amount through the physical cash put into the machine.*	*Open*	*7*
#16	*ATM printouts the account balance after the deposit.*	*Open*	*10*

Table 5.4 shows the 2 defects which were raised in relation to User-Stories 7 and 10. Thembi, the Scrum Master, will follow up and make sure these defects are corrected and both User-Stories 7 and 10 are completed in the next Sprint. The next Sprint will also carry over User-Stories 5 and 6 which the team did not have the capacity to undertake. Table 5.4 shows the Sprint Backlog board at the end of Sprint 1, i.e., at the end of the 28 days.

Below, I show a Z Schema for the two open defects.

```
_AddOpenDefects______________________________________________
| Δ Open_Defects
|_____________________________________
| DefectID' = {15 ↦ "User is able to enter deposit amount manually,
|                    instead of the ATM determining deposit amount through the physical
|                    cash put into the machine.",
|              16 ↦ "ATM printouts the account balance after the deposit."}
|_____________________________________________________________
```

Z Schema 5.11 AddOpenDefects

Z Schema 5.11 presents the 2 defects that remain open in schema Open_Defects. These 2 defects are mapped using their DefectID and description as also presented in table 5.4.

Next, in Table 5.5, I present a Sprint Backlog board as at the end of the Sprint period.

Table 5-5 User-Stories details as illustrated in Sprint Backlog board (At the end of Sprint 1)

To do	**Doing**	**Done**
5. Show Options to Deposit and Withdraw	*7. Enter Deposit Amount*	*1. Bank Card Insertion*
6. Show Available Balance After Withdrawing Option	*10. Confirm Transaction Success And Printout*	*2.Insert Deposit Account Number*
		3. Card Pin Reading
		4. Verify Customer And Show Personalized Welcome at Landing Page
		8. Enter Withdrawal Amount
		9. Push-out Cash And Show Available Balance

In table 5.5, six (6) User-Stories are completed (Done column). Going into Sprint 2, the Sprint Backlog board is cleared, and the incomplete User-Stories are once again moved into the Sprint Planning process and will appear in the To do column.

In the afternoon the team gets together for the Sprint 1 Retrospective Meeting and discusses what went well during the Sprint and what could be improved. For each of the User-Stories undertaken during the Sprint, every member who participated gives their input on what improvements can be started, what must not be done going forward, and what ideas must be continued in the next Sprints. Important feedback is that there were many defects which were as a result of the unclear software requirements, and this may be eliminated by verifying the next Sprint's requirements through Formal Methods.

Next, I created a Z Schema for Sprint Backlog at the end of the Sprint period.

```
_Sprint_Backlog_User_Stories______________________
| Δ Sprint_Backlog_User_Stories
|--------------------------------
| (∀i : [1 .. 10] •
|   (if i ∈ {5, 6} then stories (i).Status = “To do”
|    elseif  i ∈ {7,10} then stories(i).Status = “Doing”
|    else stories(i). Status = “Done”)
|__________________________________________________
```

Z Schema 5.12 Sprint_Backlog_User_Stories at the end of the Sprint

Z Schema 5.12 shows the Sprint Backlog board User-Stories statuses at the end of the first Sprint. Using condition statements, the Schema affirms that User-Stories 5 and 6 remain on the ‘To do’ status, while User-Stories 7 and 10 are still in the ‘Doing’ status as the results of the defects raised against the stories. Lastly, the Schema shows the rest of the User-Stories are ‘Done’ as they have fulfilled the customer requirements.

Sprint 2 – Day 1 (S2.1)

Based on his most recent stakeholder meetings, Mduduzi, the Scrum Product Owner, adds new items to the Scrum Product Backlog. The stakeholders claim that the font style and colour scheme used on the ATM screens do not match the bank's branding and identity. The team is then invited to the Sprint 2 Planning Meeting by Mduduzi. Under the direction of Thembi, the Scrum Master, the team discusses and agrees on User-Stories. The team also allocate time to verify the User-Stories using Formal Methods in order to address the defects that come with the ambiguity of relying of just natural language descriptions.

5.8 Value Proposition of embedding FMs in a Scrum Sprint

By introducing Formal Methods into this Chapter’s hypothetical Agile case study, Table 5.6 below presents some of the benefits we realized.

Table 5-6 Value Propositions for imbedding Formal Methods in Scrum

Concept	***Advantages***
1. State space as captured by Z Schema 5.1 Sprint Backlog	The backlog board is defined by three columns, namely, *To Do*, *Doing*, and *Done*. Formalizing the board revealed that the three components of the Sprint Backlog are pairwise disjoint.
2. Proof of Initial Sprint Backlog	The proof shows how an initial state of the system may be realized, an aspect which Scrum developers may not necessarily pay attention to.
3. Z Schema 5.3 State Space for User Stories	This schema shows that the use of FMs makes it explicit that user stories are numbered sequentially starting from 1. This is an important consideration given that days in a Sprint are numbered from 0.
4. Z Schema 5.5 State Space User-story Prioritization	The schema presents a Cartesian product which fixes an ordering among the columns of the table. Attributes of a record in relational databases are not necessarily ordered, but the columns in the prioritization of the user stories (Table 5.2) appear to be ordered. The Z specification makes this explicit through the Cartesian product as a type.
5. Extending notation of conditional predicates – *if*/*else*/*elseif* statements	We have extended the predicate notation of Z by adding conditional statements in the form of *if*/*else*/*elseif* statements as these usually occur in procedural and executable software development languages.
6. Identification of boundary conditions – Z Schema 5.9 Cash withdrawal	Boundary conditions not necessarily identified during a Scrum sprint may become explicit through formally specifying conditions.

Concept	***Advantages***
	The test WRT the amount requested (*withdrawal?* ≤ *atm*(*account?*)) indicates that the amount requested may indeed equal the amount available, so that afterwards the balance may be zero. These types of conditions can usually be missed in the brevity of natural language's user-stories and results in defects.
7. Notation for a specific day within a Sprint was developed.	The need to identify specific days from 0 to 28 (4 weeks) within specific sprints led to a pseudo FMs advantage. A notation S*m*.*n* for sprint *m*, day *n* was developed. For example, S*1*.*1* denotes Sprint *1*, day *1*.

5.9 Framework for embedding FMs in Scrum

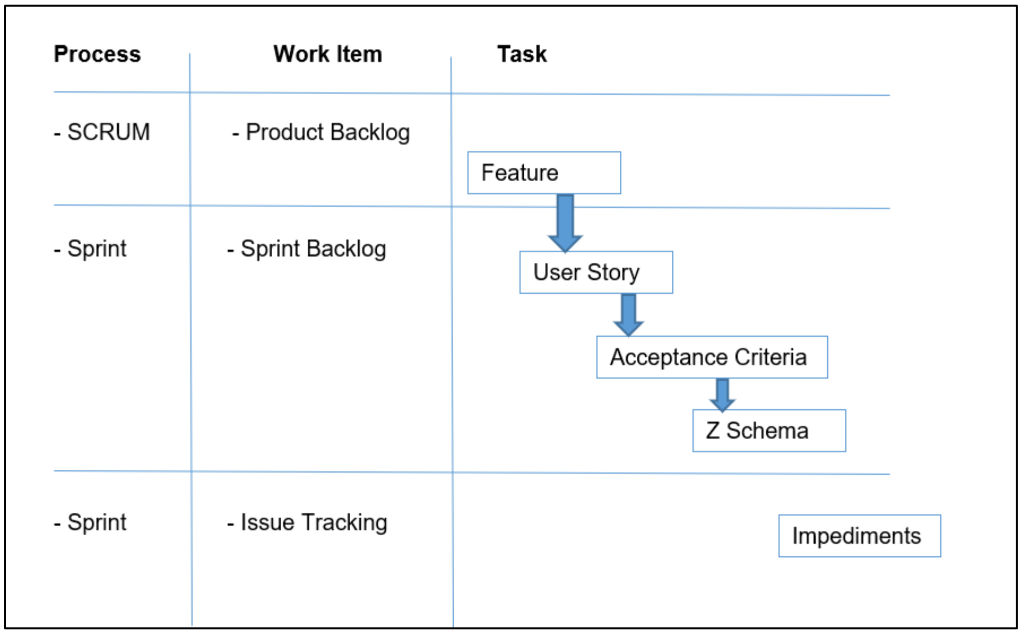
In this section I present a framework that can be replicated in order to have Formal Methods effectively usable in an Agile Software Development. By drawing closer to the subject matter, I elected to use Z Schemas within Scrum. The framework presented follows the steps I used when showing the Dlamini Bank case study's development of their ATM systems.

It should be noted that Scrum itself is a framework, not a process. This implies that many decisions made within the Scrum framework are left up to the team to determine rather than being specified in a specific methodology. In this section, I present a Scrum framework that a team wanting to harness the benefits of Formal Methods can use.

5.9.1 Diagrammatic summary of the Scrum framework

In Table 5.7 below, I show a diagrammatic summary of the Scrum framework:

Table 5-7 Diagrammatic summary of the Scrum framework



Process	Work Item	Task
- SCRUM	- Product Backlog	Feature
- Sprint	- Sprint Backlog	User Story → Acceptance Criteria → Z Schema
- Sprint	- Issue Tracking	Impediments

In Table 5.7, I present what work item belongs to what process and what task belongs to what work item. The table shows that the high-level Scrum process consists of Product Backlog where the business decides on enablement features that will enhance customer experiences. The Sprint process then consist of a Sprint Backlog which sets out the goals of what ought to be achieved within the determined period of between one to four weeks. The Sprint Backlog tasks include the formulation of User-Stories, Acceptance Criterion, and Z Schemas, all of which must be in fulfilment of Product Backlog features.

Lastly, the Sprint process then includes the issue tracking working item which is a task of removing impediments in order to achieve the set-out goals.

5.9.2 Workflow for a new backlog item

A Process Flow Chart is a visual diagram which shows the processes and relationships between the major components in a system.

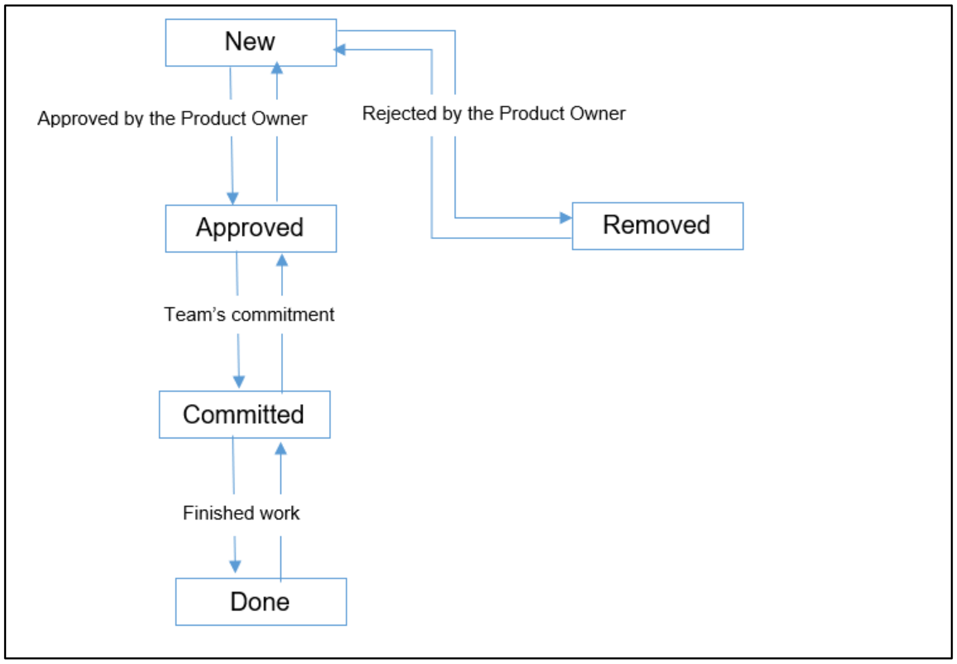


Figure 5.5 Workflow for a New Backlog Item

The above workflow presents the process followed by Product Backlog items. A new item is either removed or approved by the Product Owner. The Product Owner then obtains commitment from the Scrum Team during the Sprint Planning session. After the Scrum Team commits, the process is set in motion with the development for the Sprint period until a deliverable is considered completed or done.

5.9.3 Framework for Z Schema within Scrum Sprint

I finally present a framework for incorporating Formal Methods (Z Schemas) within Agile Software Development's Scrum.

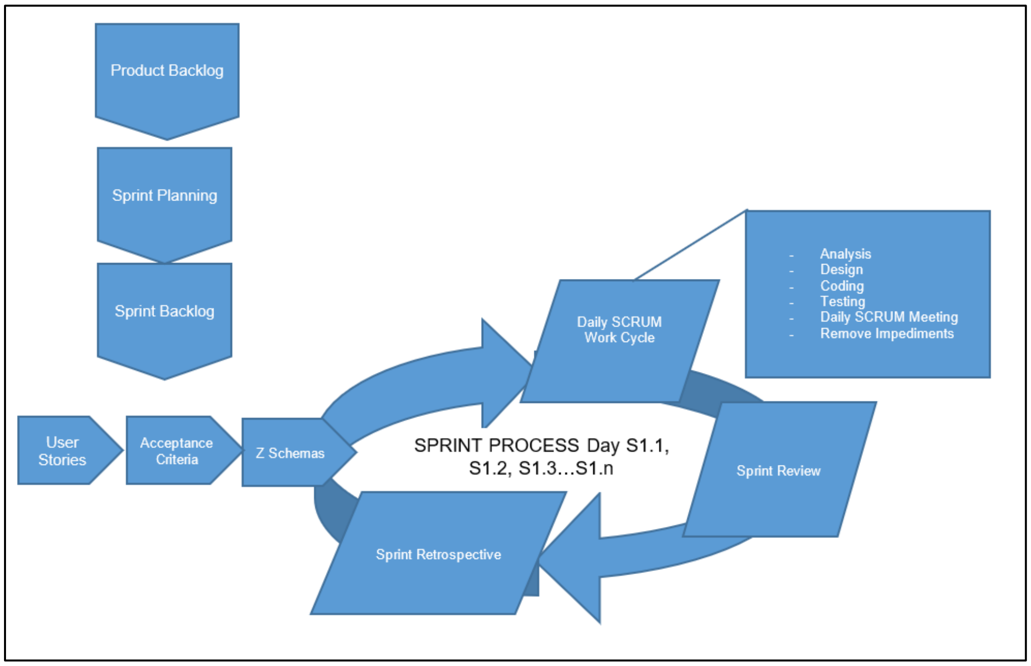


Figure 5.6 Framework for Z Schema within Scrum Sprint

In the Figure 5.6 Framework I show a Product Backlog presented by the Product Owner transitioned into a Sprint Backlog after the Scrum Team holds a Sprint Planning meeting where they prioritize the features presented by the Product Owner into a Sprint Backlog. In the Framework, I then show the three main tasks to kick-start the Sprint:

1. Creating the User-Stories
2. Formulating the Acceptance Criterion
3. Developing the Z Schemas

After the above feature refinement tasks, the daily Scrum Work Cycle follows the time-boxed evolution of the development. This evolution is continuously reviewed during the Sprint Review sessions, which are to confirm which User-Stories can be set to complete and what to do with the incomplete ones.

The Scrum Team conducts a Spint Retrospective session at the conclusion of the Sprint. A regular meeting called the "Sprint Retrospective" is held at the conclusion of a Sprint to examine what worked well during the previous cycle and what may be improved for the subsequent Sprint. The Scrum framework for creating, delivering, and managing complex projects must include it.

5.10 Summary

In Chapter 5, I presented a case study of a Scrum software development environment, where Dlamini Bank is rolling out ATM systems that will enable their client to have access to banking functionality in more convenient locations. In this case study, I show how the ATM rollout requirement is initiated and introduced to the development team. Using high-level features, the development team was able to devise User-Stories that the team iteratively develop working software in a piece-meal method. The development team then followed the Sprint’s time-box in ensuring that the allocated time and scope are delivered within the Sprint.

In presenting the major research objective, I created a Framework for Z Schema within Scrum Sprint, where FMs were embedded in the process throughout, culminating in a methodology for embedding FMs in Scrum, aimed at addressing possible shortcomings in Scrum.

5.11 Conclusion

After presenting a Scrum case study in Chapter 5, I do it again in Chapter 6 to demonstrate that Formal Methods may be used in Scrum across a variety of sectors. Due to the need to present a new case study, I want to make sure you have a solid knowledge of the Scrum Framework and why it's a framework rather than a method. Furthermore, it will be clear that the team is self-organising since for each Sprint the team as a whole decides its own fate.

In the next chapter, I present a university eVoting case study where Formal Methods are implemented within the Scrum framework.

Chapter 6: The University eVoting Case Study

In Chapter 5, I showed a Scrum case study where I used Formal Methods within a Sprint. The case study emphasised the everyday functions of Scrum processes towards fulfilling a requirement specification of an ATM system. I also expanded the banking requirement specification into more User-Stories and identified in which software development phases Formal Methods can be efficient and how business enterprises will benefit from embedding FMs in Agile Software Development. At the Scrum process level, I formalized the Sprint Backlog board, adding the User-Stories and the prioritization of User-Stories. By breaking down User-Stories into categories using the use-case technique I was able to show how Formal Methods can be useful in Agile Software Development.

Using the same framework in Chapter 6 of embedding Formal Methods in Agile Software Development as presented in Chapter 5, I use a university eVoting case study to confirm that the Formal Methods can be usable in other industries' Agile Software Development practices.

6.1 Introduction

Technology has now become a critical component in the management, organisation, and completion of the voting process. The election process should be defined as the actions that include the creation of the electoral roll, student identification, voting, vote counting, and results reporting. The process of registering eligible students to vote and assigning them to geographical campuses and residences begins with the registration procedure.

Electoral commissions throughout the world are currently looking for voter management systems to handle the electoral roll, students' identity, the act of voting, vote counting, and results reporting for student body elections. The above technological scenarios will be used to procure this voter management system in order to elicit system requirements, with the goal of closing the semantic gap between legal written documentation and the execution of a voter management system.

Computerizing voting procedures entails the use of computer technology in operations such as voter registration, voting, and counting votes. Although the initial cost of implementing electronic voting systems would be substantial, the long-term benefits would be a significant reduction in election costs.

Compliance with election legislation, the unique terminology, and the requirement to have at the very least, a legal stakeholder who understands and analyses the law are all key constraints in the procurement of such a system. Finally, it is emphasized the need of having a requirements specification that includes all of the essential procedures for voting, as well as all of the various scenarios from which a software design can be created.

6.2 The case study – Using Formal Methods in Agile Software Development

The University is intending on becoming the first South African university to conduct an election using electronic voter management system. The voter management system is intending to use students' information such as their student numbers, ID numbers and student emails for authentication as guided by the university council rules for a legitimate election. An introduction of the voter verification process feature is regarded as a major step in the procurement of the end-to-end eVoting system that will reduce the university costs for running student leadership elections, which are usually expensive and cumbersome to run. Thus, a voter verification process for voter management is required for credible and fair student elections even in other universities that would want to use a similar system.

The idea is then given to the Scrum Product Owner to devise high-level requirements for the voter management process system. The Product Owner then constructs UML use cases to simplify the idea and the need for this voter management process system. The Product Owner finally reverts back to the university council and presents them with the constructed high-level requirements in order to get confirmation that indeed this is what they require. On confirmation by the stakeholders, the Product Owner hands over these requirements in a form of an intuitive and generic product backlog to the Scrum Master and this informs the initiation of a Sprint.

Figure 6.1 below presents the high-level eVoting use cases devised by the Product Owner. Use Cases: Voter Registration, Registration Confirmation, Voting, Student Identification.

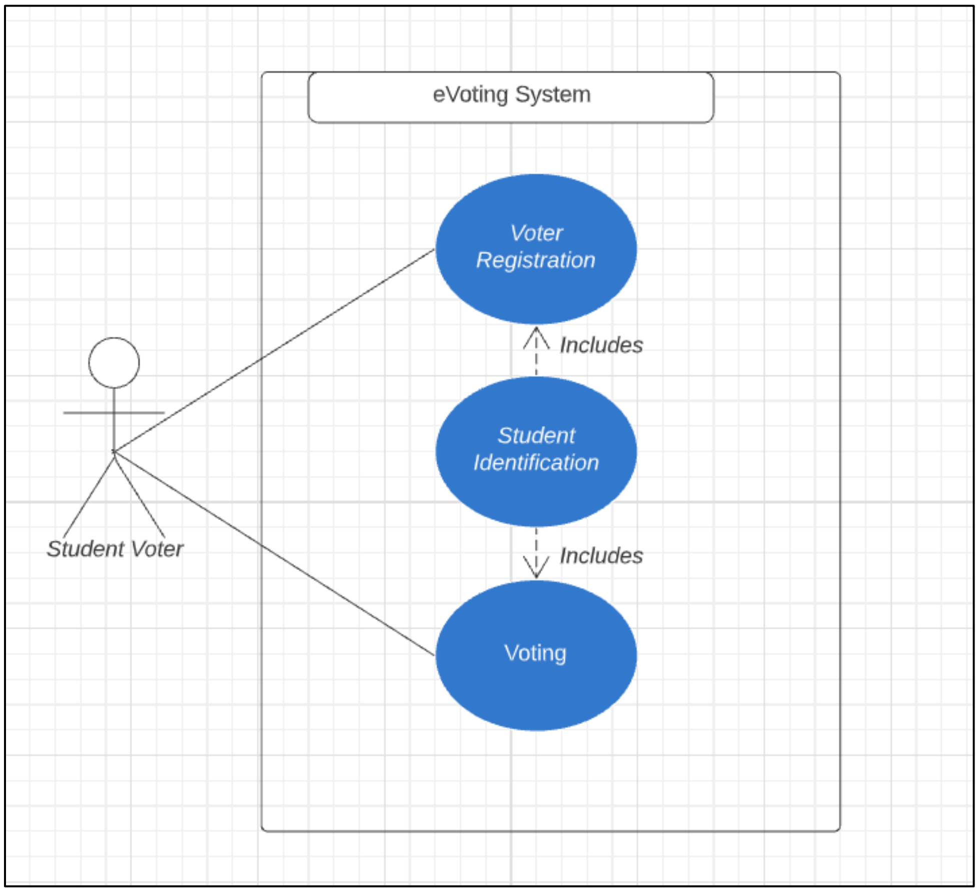


Figure 6.1 Use case diagram for the eVoting system

Figure 6.1 represents the student process for voting electronically. It shows that the identification of the student is central so that they can be able to register for voting and ultimately vote for their representatives of choice.

6.3 Sprint 1 – Day 0 (S1.0)

The Sprint Planning session kicks off a Sprint by introducing the product backlog to the development team, which is led by the Scrum Master and Product Owner. Before beginning this Sprint Planning meeting, the Scrum Master and Scrum Product Owner should assess the team's capability, consider the project's overall timeframe, and be prepared to act on prior Sprint insights. The development team will analyse this backlog during the Sprint Planning meeting to see what needs to be done next to keep the project on schedule. The Scrum Team estimates the time or effort it will take to finish each item once they have the product backlog of items.

The Scrum Master can better manage the project's budget and schedule with the use of this information. Once the items have been estimated, the team can determine how many of these User-Stories and in which combinations would fit into the next Sprint based on the team's capabilities. Table 6.1 below presents the User-Stories as devised by the product owner.

Table 6-1 Product Backlog as presented by the Product Owner

User-Story Number	***Use Case***	***User-Story***
	Voter Registration	
1		*Insert Student Number*
2		*Verify Student Registration*
3		*Register Student to Vote*
4		*Confirm Registration*
	Voting	
5		*Insert Student Number*
6		*Validate Student Voting Status*
7		*Vote*
8		*Consolidate Results*
9		*Announce Results*

Table 6.1 shows User-Stories presented in the product backlog. These are subject for discussions during the Sprint Planning by the Scrum Team, Scrum Master and Product Owner.

Next, I populate the User-Stories as indicated in Table 6.1 in a Z schema.

```
_AddUserStories_________________________________________
| Δ User_Stories
|________________________________
| stories′ = {1 ↦ "Insert Student Number",
|             2 ↦ "Verify Student Registration",
|             3 ↦ "Register Student to Vote",
|             4 ↦ "Confirm Registration",
|             5 ↦ "Insert Student Number",
|             6 ↦ "Validate Student Voting Status",
|             7 ↦ "Vote",
|             8 ↦ "Consolidate Results",
|             9 ↦ "Announce Results"}
|_______________________________________________________
```

Z Schema 6.1 AddUserStories

In schema AddUserStories I formalize the adding of the nine User-Stories into the product backlog.

In table 6.2, I present the Sprint 1 Backlog as decided on by the Scrum Team during Sprint Planning session.

Table 6-2 First Sprint Backlog Prioritization

User-Stories	Sprint Ready	Priority	Status	Sprint
1	Yes	Medium	To do	1
2	Yes	High	To do	1
3	Yes	High	To do	1
4	Yes	Medium	To do	1
5	Yes	Medium	To do	1
6	Yes	High	To do	1

User-Stories	***Sprint Ready***	***Priority***	***Status***	***Sprint***
7	*Yes*	*High*	*To do*	*1*
8	*No*	*High*	*To do*	*2*
9	*No*	*Low*	*To do*	*2*

The Scrum Team quantify the backlog items and their capacity during the Sprint Planning, where they also prioritize the User-Stories according to the effort required. Because none of the User-Stories have yet been undertaken, all their statuses are still ‘To do’. The team then agrees on which User-Stories are ready and are achievable within the first Sprint. The team decides that they do not have capacity to finish User-Stories 8 and 9. Using the above Sprint Backlog table 6.2, I then formalize the backlog into the Z Schema 6.2:

```
_Populate_Prioritization_User_Stories________________________
Δ Prioritization_User_Stories
_____________________________________________________________
(∀i : ℕ₁ •
  stories(i).Status = “To do”
∧
  (if i ∈ {8, 9} then stories (i).Sprint_Ready = “No”
   else stories (i).Sprint_Ready = “Yes”)
∧
  (if i ∈ {1, 4, 5} then stories (i).Priority = “Medium”
   elseif  i ∈ {2, 3, 6, 7, 8} then stories(i).Priority = “High”
   else stories(i).Priority = “Low”)
∧
  (if i ∈ {8, 9} then stories (i).Sprint = “2”
   else stories (i).Sprint = “1”)
_____________________________________________________________
```

Z Schema 6.2 Populate_Prioritization_User_Stories

Schema 6.2 shows the prioritization of User-Stories where all of them are still in the ‘To do’ status at the beginning of the Sprint. I then specified that User-Stories 8 and 9 are not ready for the Sprint, and the rest are ready. I also specified the User-Stories by priority where they are categorised as low, medium and high.

The basic types are:

[To do, Doing, Done]

In the Z Schema 6.3 next, I define the state space of the backlog board.

```
_Sprint_Backlog_____________________________________________
| to do: To do
| doing : Doing
| done : Done
|___________________________________
| disjoint (to do, doing) ∧ disjoint (to do, done) ∧ disjoint (doing, done)
|___________________________________________________________
```

Z Schema 6.3 Sprint_Backlog

The declaration part of the schema introduces 3 variables: To do, Doing, and Done. The variables represents the columns in Figure 6.2. The predicate presents the pairwise disjoint sets (since the same content cannot appear in multiple columns) where a Sprint Backlog item can move from to do, to doing (when it is in development), to done (when it is completed).

Figure 5.3 represents how the development team gathers in front of the Sprint Backlog board for a 15-minute daily Scrum meeting.

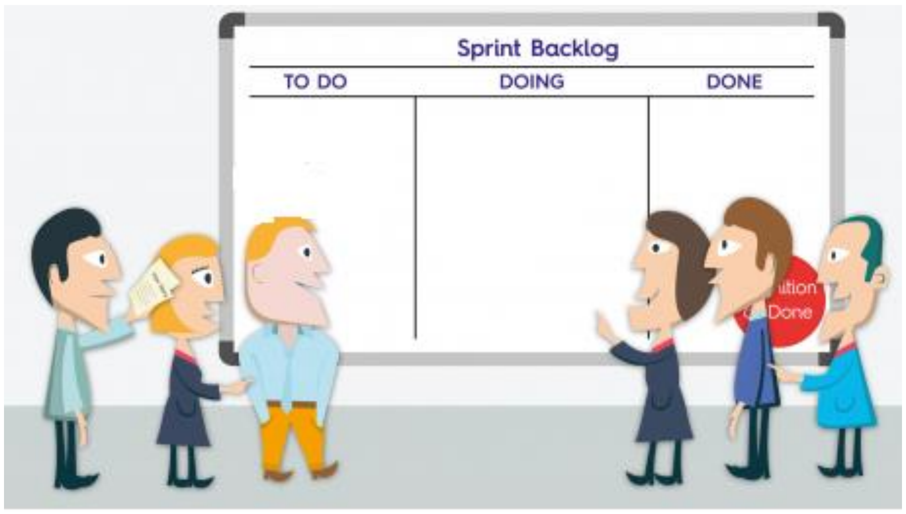


Figure 6.2 Illustration of a Development Team gathered around the Sprint Backlog board (https://watisscrum.nl/sprint-backlog/)

As indicated in Figure 6.2 and Z Schema 6.3, the board consists of three components, to do, doing, and done with types indicated in Z Schema 6.3. Since the same content cannot appear in multiple columns, the components are pairwise disjoint.

Following Z's established strategy, the next step is to define an initial state of the board and subsequently show that such a state can be realised.

$$\begin{array}{|l}
\textit{InitSprint\_Backlog} \\
\hline
\textit{Sprint\_Backlog}' \\
\hline
\textit{to do} = \emptyset \wedge \textit{doing} = \emptyset \wedge \textit{done} = \emptyset \\
\hline
\end{array}$$

Z Schema 6.4 InitSprint_Backlog

A proof obligation of the existence of an initial state arises.

Proof:

$\vdash \textit{Sprint\_Backlog}' \bullet \textit{InitSprint\_Backlog}$ (6.1)

Hence, I need to show:

$\vdash \exists\ \textit{to do}' : \textit{To do};\ \textit{doing}' : \text{Doing};\ \textit{done}' : \text{Done}\ |$

$\qquad \textit{to do}' = \emptyset \wedge \textit{doing}' = \emptyset \wedge \textit{done}' = \emptyset$ (6.2)

The proof of (6.2) follows trivially, since the empty set values are specified in schema *InitSprint_Backlog*.

6.4 Feature 1 - Voter Registration

In reference to the Sprint Backlog board (Table 6.2), the following User-Stories fall under feature 1.

- User-Story 1: Insert Student Number
- User-Story 2: Verify Student Registration
- User-Story 3: Register Student to Vote
- User-Story 4: Confirm Registration

6.4.1 User-Story Objective:

As a student at the University
I want to register to vote
So that I am eligible to vote

6.4.2 Acceptance Criteria

1. The voter needs to be a registered student

[*STUDENT*, *REGISTRATION*]

$$
\begin{array}{|l}
\hline
\textit{Vote} \\
known : \mathbb{P}\ STUDENT \\
vote : STUDENT \rightarrow\!\!\!\!\!+\ REGISTRATION \\
\hline
known = \mathrm{dom}\ vote \\
\hline
\end{array}
$$

Z Schema 6.5 Vote schema

Schema *Vote* describes the state space of a system, and the two variables represent important observations which I can make of the state:

- *known* is the set of students in the registered for the academic year;
- *vote* denotes a function which allows registered students to electronically vote for student leadership of their choice.

6.5 Feature 2 – Voting

- User-Story 1: Insert Student Number
- User-Story 2: Validate Student Voting Status
- User-Story 3: Vote
- User-Story 4: Consolidate Results
- User-Story 5: Announce Results

6.5.1 User-Story Objective

As a student at the University
I want to vote
So that the student body can be led by leaders of my choice

Schema 6.5 presents an *eVoting* formal specification which describes the intended system behaviour.

eVoting

$$\begin{array}{|l}
\Delta\ Vote \\
student? : STUDENT \\
evoting? : REGISTRATION \\
results! : RESULTS \\
\hline
student? \in known \\
vote' = vote \oplus \{student? \mapsto evoting?\} \\
results! = \text{“}Student\text{”} + student? + \text{“}has\ voted.\text{”} \\
\hline
\end{array}$$

Z Schema 6.6 eVoting System

The declaration *Δ Vote* alerts us to the fact that the schema is describing a state change: it introduces variables: *student, registration, results, student?, evoting?, and results!*. The schema represents an electronic voting scenario where a registered student is automatically eligible to vote and see the voting results.

6.6 Sprint 1 – Day 1 (S1.1)

The eVoting system development work begins on Day 1 of the project. Every morning, the complete team comes for their daily Scrum meeting. Everyone gives a quick recap of what has been completed so far, updates the expected number of hours remaining on the Sprint Task board cards, outlines what they plan to do for the day, and highlights any roadblocks to completing their work. One of the team members informs the rest of the group that he is having trouble obtaining a new licence for one of the software applications he uses. The Scrum Master verifies that other members of the team are experiencing the same problem and undertakes to address it following the meeting. Everyone returns to their duties after 15 minutes. Figure 6.3 depicts how the development team meets for a 15-minute daily Scrum meeting in front of the Sprint Backlog board. The Sprint Backlog board in Figure 6.3 contains abstract and general content.

Figure 6.3 Illustration of a Development Team gathered around the Sprint Backlog board (https://watisscrum.nl/sprint-backlog/)

As indicated in Chapter 5 as well, the Scrum Master keeps track of all of the User-Stories that have been assigned for the length of the Sprint. Individual members of the development team announce what User-Stories they are presently working on and any impediments to their progress during the daily Scrum meeting. The Done column is empty at the start of the Sprint (Day 1) until a User-Story is completed. The Scrum Master updates the Sprint Backlog board after the meeting by shifting the User-Story artefacts from To do, Doing to Done. The Scrum Master then follows up and removes all the impediments to the team's success.

6.7 Sprint 1 – Day 2 till final Sprint day (S1.2, n)

The entire team meets again during Day 2 for their daily Scrum meeting. The team disperses after the meeting and gets to work on the tasks from the Sprint Backlog that they had committed to. Until the last day of the Sprint (represented by n), the same routine continues. Each day includes a daily Scrum meeting, where progress for each User-Story is discussed and the development team and Scrum Master work towards removing impediments until all possible User-Stories are on the Done column (Table 6.3 below).

Table 6-3 User-Stories details as illustrated in Sprint Backlog board

To do	**Doing**	**Done**
8. Consolidate Results	*5. Insert Student Number*	*1. Insert Student Number*
9. Announce Results	*6. Validate Student Voting Status*	*2. Verify Student Registration*
	7. Vote	*3. Register Student to Vote*
		4. Confirm Registration

The Scrum Master and development team document and track their progress towards achieving all the Sprint's items in Table 6.3. Individual team members transfer User-Stories they're working on from the To do column to the Doing column on the Sprint Backlog board. They move the User-Story from Doing to Done once it has been completed. This is repeated continuously until the Sprint is completed; thereafter Sprint 2 begins. In the beginning of Sprint 2, the Scrum Product Owner adds new items to the Scrum Product Backlog based on the stakeholder meetings. This then allows for the repetition of the Sprint 1 process until all User-Stories are in the Done column of the Sprint Backlog board and the eVoting system development is completed.

```
_Sprint_Backlog_User_Stories____________________________
| Δ Sprint_Backlog_User_Stories
|_______________________________
| (∀i : [1 .. 10] •
|   (if i ∈ {8, 9} then stories (i).Status = “To do”
|    elseif  i ∈ {5, 6, 7} then stories(i).Status = “Doing”
|    else stories(i). Status = “Done”)
|________________________________________________________
```

Z Schema 6.7 Sprint_Backlog_User_Stories

6.8 Summary

I gave a case study of a Scrum software development environment in Chapter 6, where the university is implementing an eVoting system that will allow students to vote online from the comfort of their own homes. I illustrate how the eVoting system rollout need is launched and presented to the development team in this case study. The development team was able to create User-Stories using high-level use-cases, allowing them to iteratively produce functioning software in a piecemeal manner. The development team then adheres to the Sprint's time-box to ensure that the Sprint's specified time and scope are met.

6.9 Conclusion

Computerized voting systems would eliminate the need for ballot boxes, queue management, and paper ballots by simulating these functions. This would result in significant cost savings in terms of printing. The use of electronic voting systems to automate the verification process would help to impose necessary rules in order to check whether or not a person has already cast a vote, eliminating the need for permanent ink.

Having shown the possibility of using Formal Methods within the Agile Software Development Methodology, in Chapter 7 below I am concluding the research work.

Chapter 7: Conclusion

In Chapter 6 I presented an idea of how a Scrum Team can feature Z Specifications in establishing clearer software requirements. I was able to model how I can use predicate calculus and the Z schema calculus in verifying User-Stories. I used the ASD case study in Chapter 5 in presenting that the use of Formal Methods can be embedded in a Scrum environment. By also presenting the advantages and disadvantages of Formal Methods, I was able to identify to what extent they can be usable in order to realise their ultimate goal of quality software output.

In this chapter, I am summarizing and concluding the research work. I revisit ideas I had when proposing the research work and compare them with what I have achieved in this dissertation with respect to research objectives met.

7.1 Introduction

Having arrived at this point of this dissertation, I consider what has been achieved. In Chapter 1 I introduced the proposed study and provided the context as to why it would be beneficial for Formal Methods to be normalized and embraced in Agile Software Development. I introduced the problem statement, which I believed would be soluble through undertaking this research work.

In Chapter 2, I conducted a literature review on some of the widely used software development methodologies. I collected and reviewed scholarly work relevant to the research work for both Agile and traditional methodologies. As a central feature of the research, in Chapter 3 I introduced Formal Methods. I presented a Z Specification example for a preliminary overview of the FM subject and provided scholarly literature in justifying the need for this software verification technique to be included in Agile Software Methodology.

I then presented the research design in Chapter 4, where I disclosed how the FM/ASD relationship would be investigated and how I would collect and analyse data. This was done in a case study as an appropriate method for conducting the research, as presented in Chapter 5. The case study was based on a day-to-day Scrum practice which is measured in Sprints. In this case study I showed how the planning is done, up until a shippable product is delivered. In the Scrum case study, I identified areas of improvement that Formal Methods can help the quality of the software product delivered. In Chapter 6, I then used the same case study to embed the FMs into a Scrum practice and explicitly showed how the final product would then be improved.

7.2 Revisiting the problem statement

When I proposed the research, I highlighted that I would be focusing on the limitations of the compatibility of Agile Software Development Methodology and Formal Methods in addressing the lack of quality software output problem. I identified that Agile Software Development (ASD) facilitates rapid development of software. However, this rapidity often leads to faulty software systems, particularly the security critical systems and Formal Methods (FM) can facilitate the development of provably correct software systems, or at least highly dependable systems.

The task through this research was to find a way to normalize this relationship between ASD and FMs. In Chapter 6, I used the day-to-day practice of Scrum in showing how FMs can feature and how success can still be achieved to keep up with the increasing demand for quality and efficiency in software systems business. Specifically, I indicated through tracing formal specifications of selected Agile constructs how Formal Methods may be embedded in Agile.

7.3 Achievement of the research objectives

The research objective for the study was achieved and presented in the day-to-day practice of Scrum blending with Formal Methods for companies to keep up with the increasing demand for quality and efficiency in the software systems business. I consider these next.

7.3.1 The advantages and disadvantages of Agile and FM software development.

7.3.1.1 Advantages

Table 7-1 The main advantages of using FMs in ASDM

Conceptual Advantage	Discussion
Higher Value Product	The blending of ASD and FMs will help to deliver the product of higher value.
Measurable Correctness	The use of FMs in Scrum provides a measure of the correctness of a system, as opposed to the current process quality measures.
Early Detection of Defects	Formal Methods can be applied to the earliest design artefacts, thereby leading to earlier detection and elimination of design defects.
Clarity Questions Come Early	The use of Formal Methods forces the Scrum development team to ask questions during planning that would otherwise be postponed until coding. While Scrum allows the team to have neat visibility of the product being developed.
Effective and Efficient Test-Cases	From formal specification, we can systematically derive effective test cases directly from the product backlog. It's a cost-effective way to generate test cases.
Fast and Effective Delivery	Using FMs in ASD can help teams carry out project deliveries in a fast and effective way.
Improved Design and Problem Understanding	Using Formal Methods leads to an improved design and a good understanding of the problem domain in the Sprint Backlog. FMs provides confidence that the system under development is correct especially if proof is tracked User-Stories.
Devising Complex Projects into Piecemeal Constructs	Large and complex projects can be separated into practically manageable parts while quality is prioritized. In this regard the use of Z's schema calculus allows for a piecemeal construction of a specification.
Improved Process and Quality	Process and quality improvements are analysed during the Sprint Review. ASD works well for dynamic and fast-moving project improvement, while the use of FMs

Conceptual Advantage	Discussion
	allows for rigorous design and ultimately highly dependable software.
Stakeholder Feedback Comprehension	ASD takes and comprehends feedback given by customers and stakeholders.

7.3.1.2 Disadvantages

Table 7-2 The main disadvantages of using FMs in ASDM

Conceptual Disadvantage	Discussion
Requires Experience to Succeed	Only experienced team members can be successful in using the FMs in Scrum. That said, it arguably holds for most software development approaches.
They may be too Complex to Succeed	Most important language constructs and software system components lack formal semantic definitions or are too complex to be useful.
FMs is Ideal for Upfront and Predictable Requirements	In ASD, requirements might be different from what the user states, and will usually vary with time. This poses a challenge with traditional FMs as they usually assume that the user requirements are final. Software system normally takes inputs from external environment. These inputs may not be predictable. This obvious ignored issue usually creates the problem of developing `correct' specifications and deciding what behaviour is correct.

7.3.2 Identify what business enterprises would achieve by merging Formal Methods into Agile Software Development Methodology

Table 7-3 Potential benefits for business enterprises

Business Enterprise	Discussion
Efficiently Developed Reliable Software Output	The ultimate achievement for business enterprises is the efficiently developed reliable software output that is

Business Enterprise	Discussion
	result of merging Formal Methods into Agile Software Development Methodology.
Harnessing Positive Agile Philosophies	Scrum emphasise teams, working software, customer collaboration, and responding to change. These aspects are important for the prosperity of enterprises in general and not limited to software development.
Better Business Engagement Leading to Greater Customer Satisfaction	The Agile philosophy creates much better business engagement and leads to greater customer satisfaction. This is an important benefit that can create more positive and enduring working relationships.
Ensuring the Building of the Right Product that will Deliver the Desired Value and Benefits	It is common in more traditional projects to deliver a "successful" project and find that the product is not what was expected, needed or hoped for. In combining Agile Software Development and Formal Methods, the emphasis is placed on building the right product that will deliver the desired value and benefits.
Delivering Measurably Within Timescales, Fixed Budget, and Cost	The Scrum provides of fixed timescales which enables a fixed budget. The scope of the product and its features are variable, rather than the cost. As we are developing complete slices of functionality, we can measure the real cost of development as it proceeds, which will give us a more accurate view of the cost of future development activities and therefore this allows better planning at the enterprise level.

7.3.3 Determine for which Agile development phases it may be appropriate to implement FMs

The changing face of the software requirements is the attribute of ASD and therefore for development teams to realise the FMs' benefits they have to be usable at every level where the requirements could change. A list of clear requirements is what the development teams need to create the right product. Software requirements translate the expectations and needs of the users to functionalities and features that can be implemented. They can be clearer even from the beginning of a project, but sometimes they are hidden, implied and can even occur as unexpected guests in the middle of the development night.

I observed that Formal Methods can be implemented at every level of Scrum. However, the research has found that FMs are more effective in the Plan and Design steps. The User-Stories agreed in the first step of each iteration are the basis for the extraction of the functional and non-functional requirements that will be developed, and this is where FMs can be effectively embedded.

7.3.4 Develop a framework for embedding FMs in an Agile methodology

In Chapter 5, we presented a framework for embedding FMs into Agile methodology. I expand on the framework as the major contribution to the study in point 7.5 below.

7.4 Contribution of the study

The main contribution to this study was presented in chapters 5 and 6. Ideas on how a Scrum Team can feature Z Specifications in establishing clearer software requirements were presented. I was able to model how I can use predicate calculus and Z's schema calculus in verifying User-Stories. I used the ASD case study in Chapter 5 in presenting that the use of Formal Methods can be embedded in a Scrum environment. By also presenting the advantages and disadvantages of Formal Methods, I was able to identify what extent can be usable in order to realise their ultimate goal of quality software output.

In enhancing the usability of Formal Methods, I have extended the Z notation schemas and included conditional statements in the form of *if/else/elseif* statements as these usually occur in procedural and executable software development languages. The *if/else/elseif* statement executes a block of code or script if a specified condition is true. If the condition is false, another block of code or script can be executed. Standard Z already has the *if/else* specification construct, and I extended it accordingly by adding the *elseif*.

Formal Methods have been shown to facilitate the production of highly dependable software, yet it is hard to achieve the necessary competency level by a software engineer. Agile on the other hand hastens the software development process, yet may lead to challenges, especially with respect to mission-critical software development. In this study, I was able to contribute and show a practical process to overcome these challenges.

7.5 Contribution towards a framework for embedding FMs in Scrum

As a significant contribution into the research, in Chapter 5 I presented a framework that can be replicated in order to have Formal Methods effectively usable in an Agile Software Development. By drawing closer to the subject matter, I elected to use Z Schemas within Scrum.

The framework presents the process followed by Product Backlog items where a new development item is approved by the Product Owner. The Product Owner then obtains commitment from the Scrum Team during the Sprint Planning session. After the Scrum Team commits, the process is set in motion with the development for the Sprint period until a deliverable is considered completed or done.

The framework, consists of three main tasks to kick-start the Sprint:

1. Creating the User-Stories
2. Formulating the Acceptance Criterion
3. Developing the Z Schemas

After the above tasks, the daily Scrum Work Cycle follows the time-boxed evolution of the Sprint (one to four weeks). This evolution is continuously reviewed during the Sprint Review sessions, which are to confirm which User-Stories can be set to complete and what to do with the incomplete ones.

7.6 Future work

I summarized the results of the study into a framework which could be used as a starting point for further theoretical and empirical studies on this topic. The framework for embedding Formal Methods into Agile Software Development will be evaluated empirically in the future. Researchers could use different research methodological instruments to validate the results of future studies among practitioners in industry by developing measurement scales for the success of the proposed framework.

The above opportunity for future work will allow the implementation and testing of the proposed framework. It should provide an opportunity to identify Scrum Teams, training them in the use of discrete mathematics and Z so that they can implement the process proposed in Chapter 6.

7.7 Summary

In concluding the research work, I revisited what I had proposed to achieve in Chapter 1. I revisited the problem statement which was the limitation of Agile Software Methodology to rigorously verify software specifications in order to achieve qualitative software output.

In another part of the conclusion, I looked at whether the initial research objectives were achieved. I identified both the advantages and disadvantages of embedding Formal Methods into Agile Software Development. I also presented what would business enterprises achieve by having this proposed theory of embedding FMs in ASD. Lastly, I presented the appropriate phase or level of Scrum process when FMs would be more effective and the idea of how a Scrum Team can feature Z Specifications in establishing clearer software requirements.

Embedding FMs in the Agile software development methodology appears to be feasible, yet the usual objections to the use of FMs for software development may well arise. I hope this work will assist in addressing these challenges in the future.

References

Akbar, M.A., Sang, J., Khan, A.A., Shafiq, M., Hussain, S., Hu, H., Elahi, M. and Xiang, H., 2017. Improving the quality of software development process by introducing a new methodology–AZ-model. *IEEE Access*, *6*, pp. 4811-4823.

Zefeiti, S.M.B.A. and Mohamad, N.A., 2015, Methodological considerations in studying transformational leadership and its outcomes. *International Journal of Engineering Business Management*, *7*, p.10.

Alcazar, E.G. and Monzon, A., 2000, June. A process framework for requirements analysis and specification. In *Proceedings Fourth International Conference on Requirements Engineering. ICRE 2000.(Cat. No. 98TB100219)* (pp. 27-35). IEEE.

Alharahsheh, H.H. and Pius, A., 2020. A review of key paradigms: Positivism VS interpretivism. *Global Academic Journal of Humanities and Social Sciences*, *2*(3), pp.39-43.

Alshamrani, A. and Bahattab, A., 2015. A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model. *International Journal of Computer Science Issues (IJCSI)*, *12*(1), p.106.

Andry, J.F., Riwanto, R.E., Wijaya, R.L., Prawoto, A.A. and Prayogo, T., 2019. Development Point of Sales Using SCRUM Framework. *Journal of Systems Integration (1804-2724)*, *10*(1).

Araki, K. and Chang, H.M., 2002. Formal methods in Japan: current state, problems and challenges. In *Proceedings of the Third VDM Workshop, VDM*.

Arnold, M. and Bashir, T., 2020. Agile Methodologies as a Management Tool for Physical Systems Engineering Development.

Ayodele Adeola Adesina-Ojo 2013, *Towards the formalisation of object-oriented methodologies*, University of South Africa.

Azorín, J.M. and Cameron, R., 2010. The application of mixed methods in organisational research: A literature review. *Electronic Journal of Business Research Methods*, *8*(2), pp. 95-105.

Baham, C., 2019. Implementing scrum wholesale in the classroom. *Journal of Information Systems Education*, *30*(3), p.141.

Bannink, S., 2014, January. Challenges in the Transition from Waterfall to Scrum–a Casestudy at Portbase. In *20th Twente Student Conference on Information Technology* (Vol. 182).

Beedle, M., Bennekum, A.V., Cockburn, A., Cunningham, W., Fowler, M., Highsmith, J. and Thomas, D., 2010. Principles behind the agile manifesto. *Retrieved on November 11*, p.2010.

Bhavsar, K., Shah, V. and Gopalan, S., 2020. Scrum: An agile process reengineering in software engineering. *International Journal of Innovative Technology and Exploring Engineering*, *9*(3), pp. 840-848.

Bjørner, D., 1998, April. Formal Methods in the 21'st Century: An Assessment of Today—Predictions for the Future. In *Proc. ICSE* (Vol. 98).

Bolton, M.L., Bass, E.J. and Siminiceanu, R.I., 2013. Using formal verification to evaluate human-automation interaction: A review. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(3), pp. 88-503.

Bolton, M.L., Bass, E.J. and Siminiceanu, R.I., 2008, July. Using formal methods to predict human error and system failures. In *Proc. 2nd Int. Conf. appl. human factors ergonom*.

Bowen, J.P., 1996. Formal specification and documentation using Z: A case study approach (Vol. 66). London: International Thomson Computer Press.

Butler, M., 2001. Introductory Notes on Specification with Z. *Department of Electronics and Computer Science, University of Southampton*.

Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J. and Rodriguez, D., 2015, April. Moving fast with software verification. In *NASA Formal Methods Symposium* (pp. 3-11). Springer, Cham.

Carr, M. and Verner, J., 1997. Prototyping and software development approaches. *Department of Information Systems, City University of Hong Kong, Hong Kong*, pp.319-338.

Comte, A., 2000. The positive philosophy (in three volumes). Paul, Trench, Trubner.

Cosmas, N.I., Christiana, A.F., Jeremiah, O.O. and Ikechukwu, A.C., 2018. Transitions in System Analysis and Design Methodology. *Am. J. Inf. Sci. Technol*, *2*(2), pp.50-56.

Cui, Y., Zada, I., Shahzad, S., Nazir, S., Khan, S.U., Hussain, N. and Asshad, M., 2021. Analysis of service-oriented architecture and scrum software development approach for IIoT. *Scientific Programming*, *2021*.

De Lucia, A. and Qusef, A., 2010. Requirements engineering in agile software development. *Journal of Emerging Technologies in Web Intelligence*, *2*(3), pp.212-220.

De Vries, H. and Van der Poll, H.M., 2016. The influence of Lean thinking on organisational structure and behaviour in the discrete manufacturing industry. *Journal of Contemporary Management*, *13*(1), pp.55-89.

Dongmo, C., 2016. *Formalising non-functional requirements embedded in user requirements notation (URN) models* (Doctoral dissertation).

Dongmo, C. and van der Poll, J.A., 2009. Use of Case Maps as an Aid in the Construction of a Formal Specification.

Dudovskiy, J. 2018, 06/08/2018-last update*, Research Methodology*. Available: https://research-methodology.net/research-methodology/research-approach/deductive-approach-2/ [2018, 18/09/2018].

Dwivedi, A.K. and Rath, S.K., 2012, December. Model to specify real time system using Z and Alloy languages: A comparative approach. In *International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012)* (pp. 1-6). IET.

Eden, A.H., 2001, November. Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering* (pp. 256-263).

Eleftherakis, G. and Cowling, A.J., 2003, November. An agile formal development methodology. In *Proceedings of the 1st South-East European Workshop on Formal Methods* (pp. 36-47).

Enderton, H.B., 1977. *Elements of set theory*. Academic press.

Erich, F., Amrit, C. and Daneva, M., 2014. Report: DevOps literature review. *University of Twente, Tech. Rep*.

Franch, X., Gómez, C., Jedlitschka, A., López, L., Martínez-Fernández, S., Oriol, M. and Partanen, J., 2018, June. Data-driven elicitation, assessment and documentation of quality requirements in agile software development. In *International Conference on Advanced Information Systems Engineering* (pp. 587-602). Springer, Cham.

Fulara, J. and Jakubczyk, K., 2010, January. Practically applicable formal methods. In *International Conference on Current Trends in Theory and Practice of Computer Science* (pp. 407-418). Springer, Berlin, Heidelberg.

Ganesh, N. and Thangasamy, S., 2011. Issues identified in the software process due to barriers found during eliciting requirements on agile software projects: Insights from India. *International Journal of Computer Applications*, *16*(5), pp.7-12.

Garg, A., 2009. Agile software development. *DRDO Science Spectrum*, *1*, pp.55-59.

Gleirscher, M., Foster, S. and Woodcock, J., 2019. New opportunities for integrated formal methods. *ACM Computing Surveys (CSUR)*, *52*(6), pp.1-36.

Gruhn, V. and Striemer, R., 2018. *The Essence of Software Engineering*. Springer Nature.

Gruner, S., 2010. FM+ AM'09: workshop on formal methods and agile methods. *Innovations in Systems and Software Engineering*, *6*(1), pp.135-136.

Gustavsson, T., 2019, May. Voices from the teams-impacts on autonomy in large-scale agile software development settings. In *International Conference on Agile Software Development* (pp. 29-36). Springer, Cham.

Hall, A., 1990. Seven myths of formal methods. *IEEE software*, *7*(5), pp.11-19.

Hatcher, C.A., 2019. *A Conceptual Framework for Flight Test Management and Execution Utilizing Agile Development and Project Management Concepts*. 812th Test Support Squadron, 812 TSS/ENTI Edwards United States.

Hehner, E.C., 2017. *A practical theory of programming*. Springer Science & Business Media.

Herrmannsdörfer, M., Konrad, S. and Berenbach, B., 2008. Tabular notations for state machine-based specifications. *Crosstalk*, *21*(3), pp.18-23.

Highsmith, J., 2003. Agile software development-why it is hot. *Extreme Programming Perspectives, M. Marchesi, et al., Editors*, pp.9-16.

Hinkelmann, K. and Witschel, H.F., 2013. How to choose a research methodology. *University of Applied Sciences, Northwestern Switzerland, School of Business, Available Online: http://knut. hinkelmann. ch/lectures/project2013/p1_5_how-to-choose-a-researchmethodology. pdf [Accessed 1 April 2017]*.

Holbeche, L., 2018. *The agile organization: How to build an engaged, innovative and resilient business*. Kogan Page Publishers.

Huisman, M., Gurov, D. and Malkis, A., 2020. Formal methods: from academia to industrial practice. A travel guide. *arXiv preprint arXiv:2002.07279*.

Hussain, A., Mkpojiogu, E.O., Ishak, N. and Mokhtar, N., 2019. A Study on the Perceived Mobile Experience of Myeg Users. *Int. J. Interact. Mob. Technol.*, *13*(11), pp.4-23.

Ismail, N.N.S. and Abdullah, H., 2017, March. Implementing Rapid Application Development (RAD) methodology in developing Online Laboratory and Room Booking System (ELABAS). In *e-Proceedings iCompEx17 Academic Paper*.

Kassab, M., Lee, J., Mazzara, M., Succi, G. and Tumyrkin, R., 2016. Software Quality-Traditional vs. Agile: an Empirical Investigation. *arXiv preprint arXiv:1610.08312*.

Khalifa, M. and Verner, J.M., 2000. Drivers for software development method usage. *IEEE Transactions on Engineering Management*, *47*(3), pp.360-369.

Kim, G., Humble, J., Debois, P., Willis, J. and Forsgren, N., 2021. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution.

Knight, J.C., DeJong, C.L., Gibble, M.S. and Nakano, L.G., 1997. Why are formal methods not used more widely? In *Fourth NASA formal methods workshop*.

Krishnan, M.S., 2015. Software development risk aspects and success frequency on spiral and agile model. *International Journal of Innovative Research in Computer and Communication Engineering (An ISO 3297: 2007 Certified Organization)*, *3*(1), pp.301-310.

Lall, M., van der Poll, J.A. and Venter, L.M., 2012, November. Towards a formal definition of availability of web services. In *The International Conference on Computing, Networking and Digital Technologies (ICCNDT 2012)* (pp. 154-165).

Löwe, M., 2010. Formal Methods in Agile Development. *Special issue of Electronic Communications of the EASST: Graph and Model Transformation*, *30*, pp.1-6.

Lano, K., 1995, May. Reactive system specification and refinement. In *Colloquium on Trees in Algebra and Programming* (pp. 696-710). Springer, Berlin.

Larsen, P.G., Fitzgerald, J.S. and Wolff, S., 2011. Are formal methods ready for agility? a reality check. In *FM+ AM 2010: Second International Workshop on Formal Methods and Agile Methods,* Newcastle University.

Madni, A.M., Sievers, M.W., Humann, J., Ordoukhanian, E., Boehm, B. and Lucero, S., 2018. Formal methods in resilient systems design: application to multi-UAV system-of-systems control. In *Disciplinary Convergence in Systems Engineering Research* (pp. 407-418). Springer, Cham.

Mafuwane, B.M., 2011. *The contribution of instructional leadership to learner performance* (Doctoral dissertation, University of Pretoria).

Manandhar, N. 2008, *Agile Software Development*, Natick, Massachusetts.

Masombuka, K.T., 2020. *A framework for a successful collaboration culture in software development and operations (DevOps) environments* (Doctoral dissertation).

Mbala, I.N. and van der Poll, J.A., 2017, December. Towards Specification Formalisms for Data Warehousing Requirements Elicitation Techniques. In *The 3rd International Conference on Computing Technology and Information Management (IC-CTIM 2017)* (Vol. 1, pp. 45-58).

McCormick, M., 2012. Waterfall vs. Agile methodology. *MPCS, N/A*, *3*.

Meseguer, P., 1992. Towards a conceptual framework for expert system validation. *AI Communications*, *5*(3), pp.119-135.

Milićević, J.M., Filipović, F., Jezdović, I., Naumović, T. and Radenković, M., 2019. Scrum agile framework in e-business project management: an approach to teaching scrum. *European Project Management Journal*, *9*(1), pp.52-60.

Moi, L. and Cabiddu, F., 2021. Leading digital transformation through an Agile Marketing Capability: the case of Spotahome. *Journal of Management and Governance*, *25*(4), pp.1145-1177.

Moyo, B., 2021. *The contingent use of systems development methodologies in South Africa* (Doctoral dissertation, North-West University (South Africa)).

Mushashu, E.T. and Mtebe, J.S., 2019, May. Investigating Software Development Methodologies and Practices in Software Industry in Tanzania. In *2019 IST-Africa Week Conference (IST-Africa)* (pp. 1-11). IEEE.

Nemathaga, A.P. and van der Poll, J.A., 2019. Adoption of formal methods in the commercial world. In *Eight international conference on advances in computing, communication and information technology (CCIT 2019)* (pp. 75-84).

Novikov, M. and Heuser, N., 2006. "Agile Software Development",1, pp. 1-1-6.

O'Regan, G., 2020. *Mathematics in computing*. Springer International Publishing.

Pandey, S.K. and Batra, M., 2013. Formal methods in requirements phase of SDLC. *International Journal of Computer Applications*, *70*(13), pp.7-14.

Popli, R. and Chauhan, N., 2013. Agile software development. *International Journal of Computer Science and Communication*, *4*(2), pp.153-156.

Potter, B., Sinclair, J. and Till, D., 1992. *An introduction to formal specification and Z*. Prentice-Hall, Inc..

Rasch, L. and Thun, V., 2020. The Road to Become Agile: A case study of agile transformations in the retail market, including an organization development approach.

Roche, J., 2013. Adopting DevOps practices in quality assurance. *Communications of the ACM*, *56*(11), pp.38-43.

Rodriguez-Calero, I.B., Coulentianos, M.J., Daly, S.R., Burridge, J. and Sienko, K.H., 2020. Prototyping strategies for stakeholder engagement during front-end design: Design practitioners' approaches in the medical device industry. *Design Studies*, *71*, p.100977.

Royce, W.W., 1970, August. Managing the development of large software systems Dr. Winston W. Rovce Introduction. In *Ieee Wescon* (pp. 328-338).

Rush, D.E. and Connolly, A.J., 2020. An agile framework for teaching with scrum in the IT project management classroom. *Journal of Information Systems Education*, *31*(3), pp.196-207.

Sabale, R.G. and Dani, A.R., 2012. Comparative study of prototype model for software engineering with system development life cycle. *IOSR Journal of Engineering*, *2*(7), pp.21-24.

Saunders, M., Lewis, P. and Thornhill, A., 2018. *Research methods for business students*. Pearson education.

Schaefer, I. and Hähnle, R., 2011. Formal methods in software product line engineering. *Computer*, *44*(02), pp.82-85.

Sharma, A. and Bawa, R.K., 2020. Identification and integration of security activities for secure agile development. *International Journal of Information Technology*, pp.1-14.

Sirjani, M., Lee, E.A. and Khamespanah, E., 2020, July. Model checking software in cyberphysical systems. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)* (pp. 1017-1026). IEEE.

Smartbear 2018, *6 Ways to Measure the ROI of Automated Testing*, Smartbear, Galway, Ireland.

Solms, F. and Loubser, D., 2010. URDAD as a semi-formal approach to analysis and design. *Innovations in Systems and Software Engineering*, *6*(1), pp.155-162.

Souri, A., Rahmani, A.M., Navimipour, N.J. and Rezaei, R., 2019. A symbolic model checking approach in formal verification of distributed systems. *Human-centric Computing and Information Sciences*, *9*(1), pp.1-27.

Spivey, J.M. and Abrial, J.R., 1998. *The Z notation* (Vol. 29). Hemel Hempstead: Prentice Hall.

Stocks, P.A., 1993. *Applying formal methods to software testing* (Doctoral dissertation, University of Queensland).

Stocks, P. and Carrington, D., 1996. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, *22*(11), pp.777-793.

Szalvay, V., 2004. An introduction to agile software development. *Danube Technologies*. Inc., Bellevue,(November), pp.1-9.

Tanvir, S., Safdar, M., Tufail, H. and Qamar, U., 2018. Merging prototyping with agile software development methodology. In *International Conference on Engineering, Computing & Information Technology (ICECIT 2018)* (pp. 50-54).

Tomayko, J.E., 2017, September. Engineering of unstable requirements using agile methods. In *International Conference on Time-Constrained Requirements Engineering*.

Tonchia, S., 2018. Project strategy management. In *Industrial Project Management* (pp. 81-92). Springer, Berlin.

Torp, C.E., 2003. Method of software validation. *Nordtest Report TR*, *535*.

Tretmans, G.J. and Belinfante, A., 1999. *Automatic testing with formal methods* (pp. 2011-2012). Centre for Telematics and Information Technology, University of Twente.

Turetken, O., Stojanov, I. and Trienekens, J.J., 2017. Assessing the adoption level of scaled agile development: A maturity model for Scaled Agile Framework. *Journal of Software: Evolution and process*, *29*(6), p.e1796.

Turk, D., France, R. and Rumpe, B., 2002, May. Limitations of agile software processes. In *Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002)* (pp. 43-46).

Tutorials Point 2018*, SDLC* [Homepage of Tutorials Point], [Online]. Available: https://www.tutorialspoint.com/index.htm [2018, 04/09/2018].

Van der Poll, J.A., 2010. Formal methods in software development: A road less travelled. *South African Computer Journal*, *2010*(45), pp.40-52.

Van der Poll, J.A. and Kotzé, P., 2005. Enhancing the established strategy for constructing a Z specification: Reviewed article. *South African Computer Journal*, *2005*(35), pp.118-131.

Vijayasarathy, L.R. and Butler, C.W., 2016. Choice of software development methodologies: Do organizational, project, and team characteristics matter?. *IEEE software*, *33*(5), pp.86-94.

Vlaanderen, K., Jansen, S., Brinkkemper, S. and Jaspers, E., 2011. The agile requirements refinery: Applying SCRUM principles to software product management. *Information and software technology*, *53*(1), pp.58-70.

Wang, Y., Li, J., Hongbo, S., Li, Y., Akhtar, F. and Imran, A., 2019. A survey on VV&A of large-scale simulations. *International Journal of Crowd Science*, *3*(1), pp.63-86.

Woodcock, J. and Davies, J., 1996. Using Z: Specification, Refinement, and Proof.

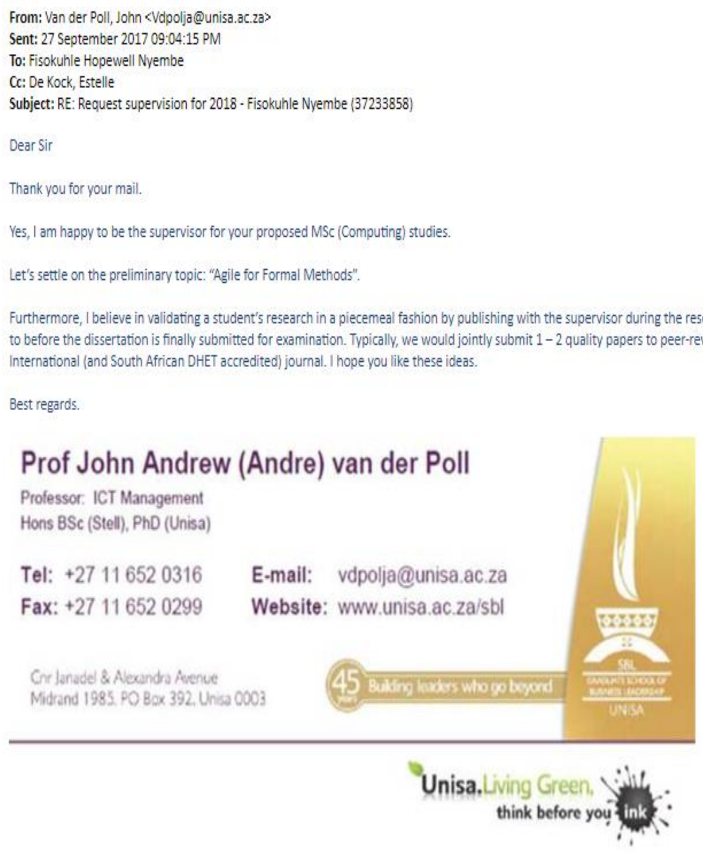
Zayat W, Senvar O. Framework study for agile software development via scrum and Kanban. International journal of innovation and technology management. 2020 Jun 24;17(04):2030002.

Zainal, Z.A.I.D.A.H., 2008. The relationship between reading comprehension and strategies of readers: A case study of UTM students. *Research in English Language Teaching*, *1*(3), pp.95-118.

Zhang, X., Hu, T., Dai, H. and Li, X., 2010. Software development methodologies, trends, and implications. *Information Technology Journal*, *9*(8), pp.1747-1753.

Appendix A: Student/Supervisor Agreement

Supervisor: Prof. Van der Poll, John



From: Van der Poll, John <Vdpolja@unisa.ac.za>
Sent: 27 September 2017 09:04:15 PM
To: Fisokuhle Hopewell Nyembe
Cc: De Kock, Estelle
Subject: RE: Request supervision for 2018 - Fisokuhle Nyembe (37233858)

Dear Sir

Thank you for your mail.

Yes, I am happy to be the supervisor for your proposed MSc (Computing) studies.

Let's settle on the preliminary topic: "Agile for Formal Methods".

Furthermore, I believe in validating a student's research in a piecemeal fashion by publishing with the supervisor during the research project. This way any change in direction, omissions, etc. may be identified and attended to before the dissertation is finally submitted for examination. Typically, we would jointly submit 1 – 2 quality papers to peer-reviewed conferences, with the final dissertation to be summarised and submitted to an International (and South African DHET accredited) journal. I hope you like these ideas.

Best regards.

Prof John Andrew (Andre) van der Poll
Professor: ICT Management
Hons BSc (Stell), PhD (Unisa)

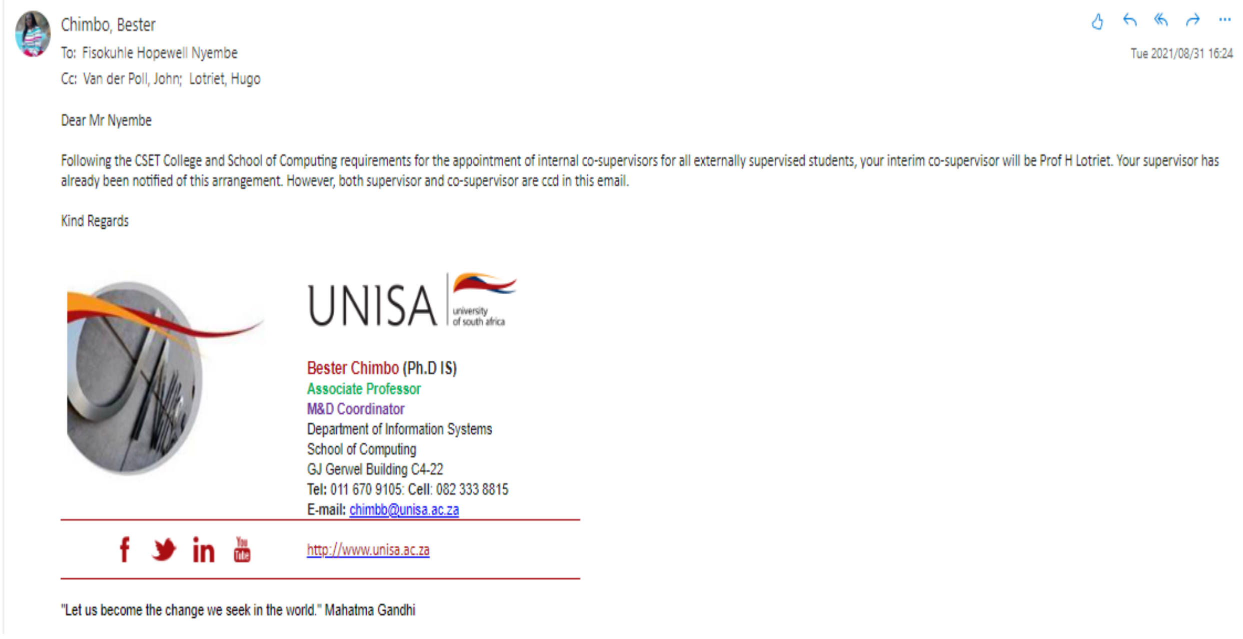
Tel: +27 11 652 0316 **E-mail:** vdpolja@unisa.ac.za
Fax: +27 11 652 0299 **Website:** www.unisa.ac.za/sbl

Cnr Janadel & Alexandra Avenue
Midrand 1985, PO Box 392, Unisa 0003

45 Building leaders who go beyond

Unisa.Living Green, think before you ink

Co-supervisor: Prof. Lotriet, Hugo



Chimbo, Bester
To: Fisokuhle Hopewell Nyembe
Cc: Van der Poll, John; Lotriet, Hugo

Tue 2021/08/31 16:24

Dear Mr Nyembe

Following the CSET College and School of Computing requirements for the appointment of internal co-supervisors for all externally supervised students, your interim co-supervisor will be Prof H Lotriet. Your supervisor has already been notified of this arrangement. However, both supervisor and co-supervisor are ccd in this email.

Kind Regards

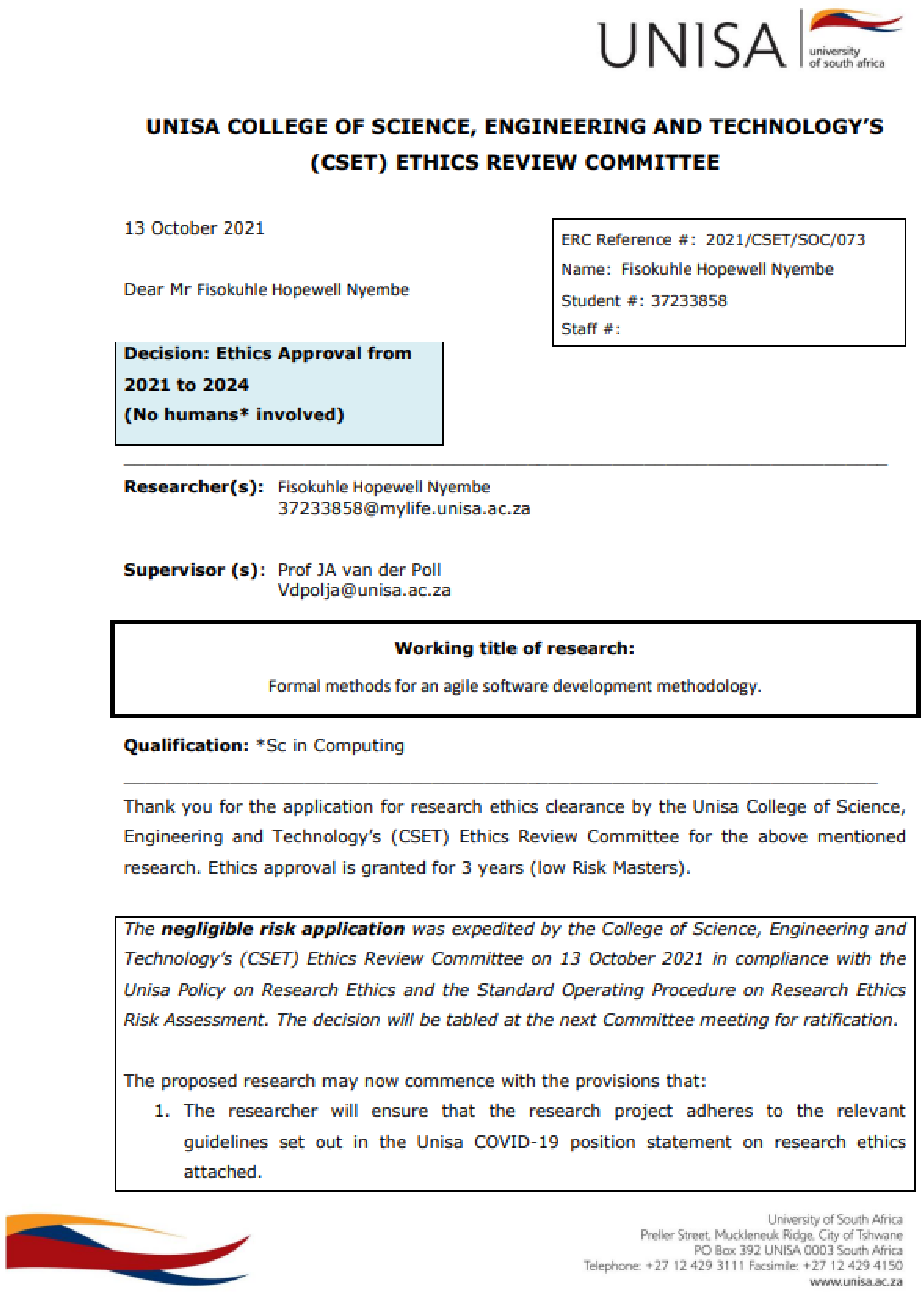
UNISA university of south africa

Bester Chimbo (Ph.D IS)
Associate Professor
M&D Coordinator
Department of Information Systems
School of Computing
GJ Gerwel Building C4-22
Tel: 011 670 9105; **Cell:** 082 333 8815
E-mail: chimbb@unisa.ac.za

http://www.unisa.ac.za

"Let us become the change we seek in the world." Mahatma Gandhi

Appendix B: Ethical Clearance Certificate



UNISA university of south africa

UNISA COLLEGE OF SCIENCE, ENGINEERING AND TECHNOLOGY'S (CSET) ETHICS REVIEW COMMITTEE

13 October 2021

Dear Mr Fisokuhle Hopewell Nyembe

ERC Reference #: 2021/CSET/SOC/073
Name: Fisokuhle Hopewell Nyembe
Student #: 37233858
Staff #:

Decision: Ethics Approval from 2021 to 2024 (No humans* involved)

Researcher(s): Fisokuhle Hopewell Nyembe
37233858@mylife.unisa.ac.za

Supervisor (s): Prof JA van der Poll
Vdpolja@unisa.ac.za

Working title of research:

Formal methods for an agile software development methodology.

Qualification: *Sc in Computing

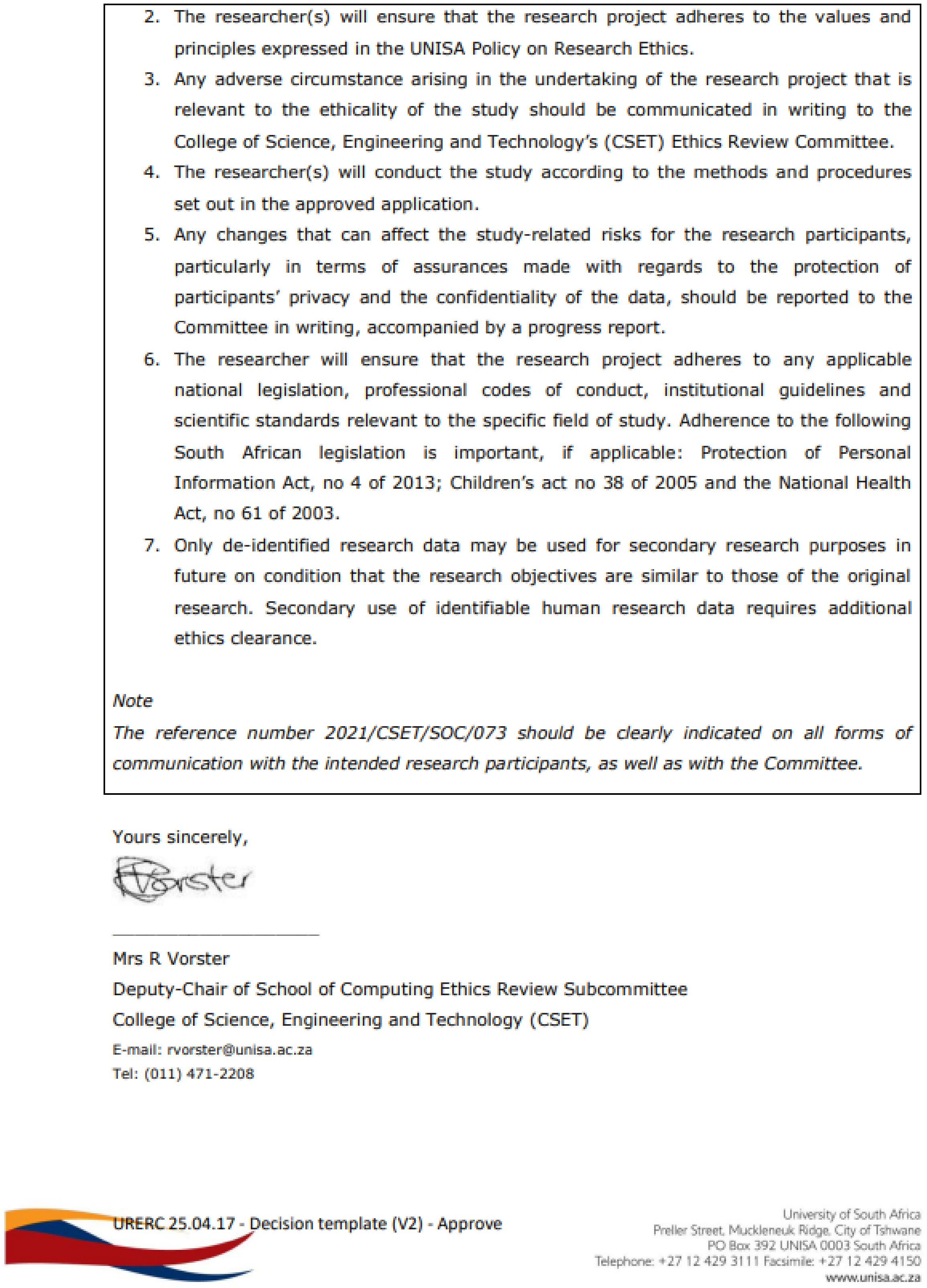
Thank you for the application for research ethics clearance by the Unisa College of Science, Engineering and Technology's (CSET) Ethics Review Committee for the above mentioned research. Ethics approval is granted for 3 years (low Risk Masters).

*The **negligible risk application** was expedited by the College of Science, Engineering and Technology's (CSET) Ethics Review Committee on 13 October 2021 in compliance with the Unisa Policy on Research Ethics and the Standard Operating Procedure on Research Ethics Risk Assessment. The decision will be tabled at the next Committee meeting for ratification.*

The proposed research may now commence with the provisions that:

1. The researcher will ensure that the research project adheres to the relevant guidelines set out in the Unisa COVID-19 position statement on research ethics attached.

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za



2. The researcher(s) will ensure that the research project adheres to the values and principles expressed in the UNISA Policy on Research Ethics.
3. Any adverse circumstance arising in the undertaking of the research project that is relevant to the ethicality of the study should be communicated in writing to the College of Science, Engineering and Technology's (CSET) Ethics Review Committee.
4. The researcher(s) will conduct the study according to the methods and procedures set out in the approved application.
5. Any changes that can affect the study-related risks for the research participants, particularly in terms of assurances made with regards to the protection of participants' privacy and the confidentiality of the data, should be reported to the Committee in writing, accompanied by a progress report.
6. The researcher will ensure that the research project adheres to any applicable national legislation, professional codes of conduct, institutional guidelines and scientific standards relevant to the specific field of study. Adherence to the following South African legislation is important, if applicable: Protection of Personal Information Act, no 4 of 2013; Children's act no 38 of 2005 and the National Health Act, no 61 of 2003.
7. Only de-identified research data may be used for secondary research purposes in future on condition that the research objectives are similar to those of the original research. Secondary use of identifiable human research data requires additional ethics clearance.

Note

The reference number 2021/CSET/SOC/073 should be clearly indicated on all forms of communication with the intended research participants, as well as with the Committee.

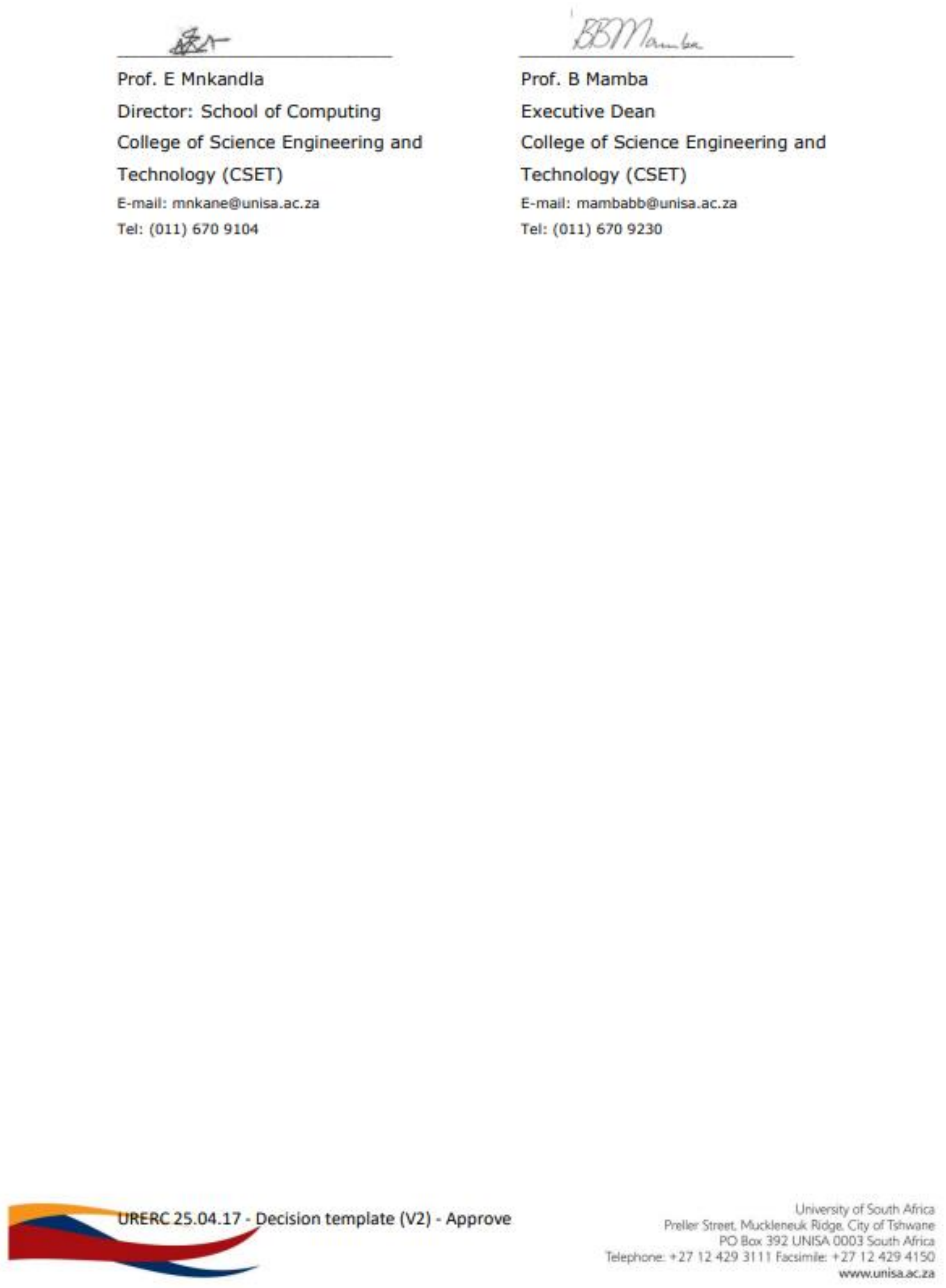
Yours sincerely,

[illegible]

Mrs R Vorster
Deputy-Chair of School of Computing Ethics Review Subcommittee
College of Science, Engineering and Technology (CSET)
E-mail: rvorster@unisa.ac.za
Tel: (011) 471-2208

URERC 25.04.17 - Decision template (V2) - Approve

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za



[illegible]

Prof. E Mnkandla
Director: School of Computing
College of Science Engineering and Technology (CSET)
E-mail: mnkane@unisa.ac.za
Tel: (011) 670 9104

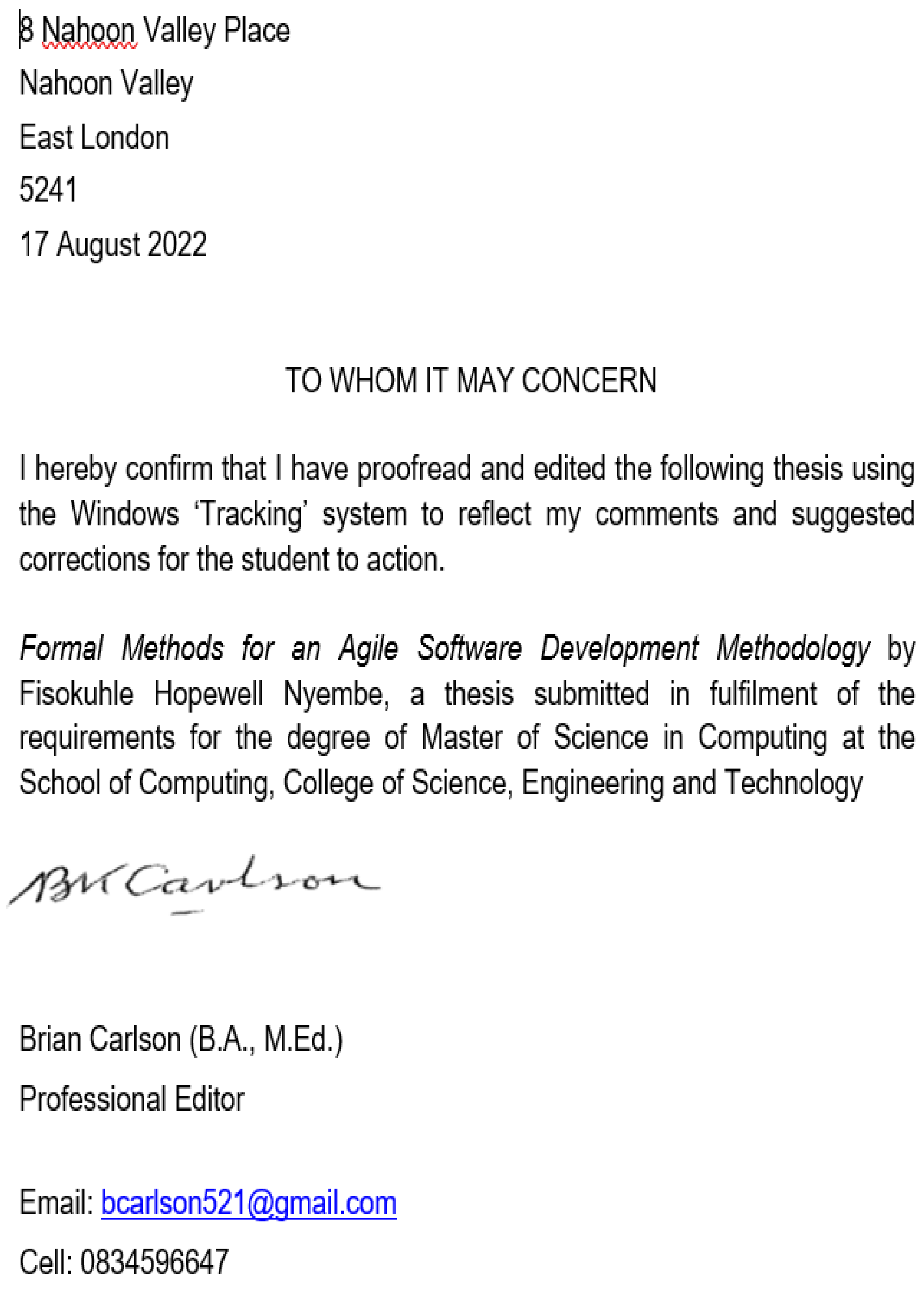
BBMamba

Prof. B Mamba
Executive Dean
College of Science Engineering and Technology (CSET)
E-mail: mambabb@unisa.ac.za
Tel: (011) 670 9230

URERC 25.04.17 - Decision template (V2) - Approve

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za

Appendix C: Language Editor Certificate



8 Nahoon Valley Place

Nahoon Valley

East London

5241

17 August 2022

TO WHOM IT MAY CONCERN

I hereby confirm that I have proofread and edited the following thesis using the Windows 'Tracking' system to reflect my comments and suggested corrections for the student to action.

Formal Methods for an Agile Software Development Methodology by Fisokuhle Hopewell Nyembe, a thesis submitted in fulfilment of the requirements for the degree of Master of Science in Computing at the School of Computing, College of Science, Engineering and Technology

[illegible]

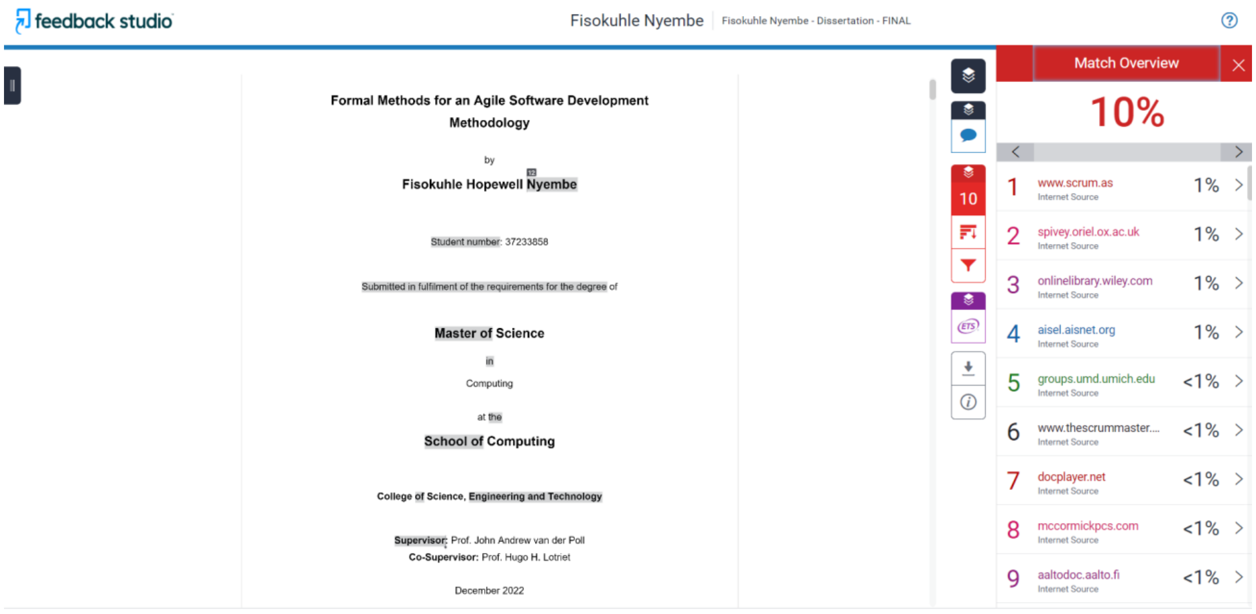
Brian Carlson (B.A., M.Ed.)

Professional Editor

Email: bcarlson521@gmail.com

Cell: 0834596647

Appendix D: Turnitin Report (1st page)



feedback studio | Fisokuhle Nyembe | Fisokuhle Nyembe - Dissertation - FINAL

Formal Methods for an Agile Software Development Methodology

by

Fisokuhle Hopewell Nyembe

Student number: 37233858

Submitted in fulfilment of the requirements for the degree of

Master of Science

in

Computing

at the

School of Computing

College of Science, Engineering and Technology

Supervisor: Prof. John Andrew van der Poll
Co-Supervisor: Prof. Hugo H. Lotriet

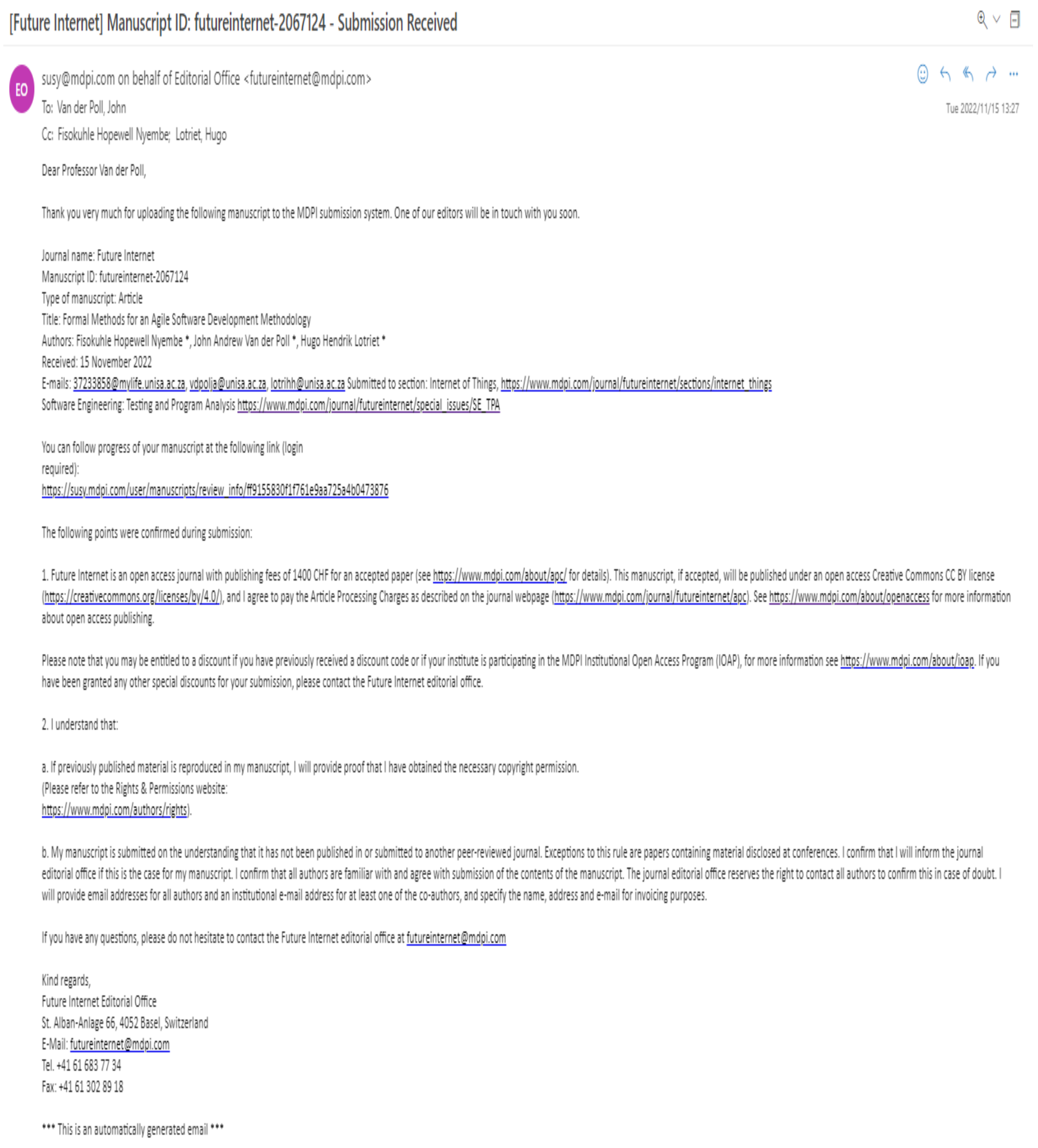
December 2022

Match Overview

10%

1	www.scrum.as Internet Source	1%
2	spivey.oriel.ox.ac.uk Internet Source	1%
3	onlinelibrary.wiley.com Internet Source	1%
4	aisel.aisnet.org Internet Source	1%
5	groups.umd.umich.edu Internet Source	<1%
6	www.thescrummaster.... Internet Source	<1%
7	docplayer.net Internet Source	<1%
8	mccormickpcs.com Internet Source	<1%
9	aaltodoc.aalto.fi Internet Source	<1%

Appendix E: Journal Article Submission



[Future Internet] Manuscript ID: futureinternet-2067124 - Submission Received

susy@mdpi.com on behalf of Editorial Office <futureinternet@mdpi.com>

To: Van der Poll, John

Cc: Fisokuhle Hopewell Nyembe; Lotriet, Hugo

Tue 2022/11/15 13:27

Dear Professor Van der Poll,

Thank you very much for uploading the following manuscript to the MDPI submission system. One of our editors will be in touch with you soon.

Journal name: Future Internet
Manuscript ID: futureinternet-2067124
Type of manuscript: Article
Title: Formal Methods for an Agile Software Development Methodology
Authors: Fisokuhle Hopewell Nyembe *, John Andrew Van der Poll *, Hugo Hendrik Lotriet *
Received: 15 November 2022
E-mails: 37233858@mylife.unisa.ac.za, vdpolja@unisa.ac.za, lotrihh@unisa.ac.za Submitted to section: Internet of Things, https://www.mdpi.com/journal/futureinternet/sections/internet_things
Software Engineering: Testing and Program Analysis https://www.mdpi.com/journal/futureinternet/special_issues/SE_TPA

You can follow progress of your manuscript at the following link (login
required):
https://susy.mdpi.com/user/manuscripts/review_info/ff9155830f1f761e9aa725a4b0473876

The following points were confirmed during submission:

1. Future Internet is an open access journal with publishing fees of 1400 CHF for an accepted paper (see https://www.mdpi.com/about/apc/ for details). This manuscript, if accepted, will be published under an open access Creative Commons CC BY license (https://creativecommons.org/licenses/by/4.0/), and I agree to pay the Article Processing Charges as described on the journal webpage (https://www.mdpi.com/journal/futureinternet/apc). See https://www.mdpi.com/about/openaccess for more information about open access publishing.

Please note that you may be entitled to a discount if you have previously received a discount code or if your institute is participating in the MDPI Institutional Open Access Program (IOAP), for more information see https://www.mdpi.com/about/ioap. If you have been granted any other special discounts for your submission, please contact the Future Internet editorial office.

2. I understand that:

a. If previously published material is reproduced in my manuscript, I will provide proof that I have obtained the necessary copyright permission.
(Please refer to the Rights & Permissions website:
https://www.mdpi.com/authors/rights).

b. My manuscript is submitted on the understanding that it has not been published in or submitted to another peer-reviewed journal. Exceptions to this rule are papers containing material disclosed at conferences. I confirm that I will inform the journal editorial office if this is the case for my manuscript. I confirm that all authors are familiar with and agree with submission of the contents of the manuscript. The journal editorial office reserves the right to contact all authors to confirm this in case of doubt. I will provide email addresses for all authors and an institutional e-mail address for at least one of the co-authors, and specify the name, address and e-mail for invoicing purposes.

If you have any questions, please do not hesitate to contact the Future Internet editorial office at futureinternet@mdpi.com

Kind regards,
Future Internet Editorial Office
St. Alban-Anlage 66, 4052 Basel, Switzerland
E-Mail: futureinternet@mdpi.com
Tel. +41 61 683 77 34
Fax: +41 61 302 89 18

*** This is an automatically generated email ***