University of Huddersfield Repository

Morris, Lee

Understanding Software Sustainability in the field of Research Software Engineering

Original Citation

Morris, Lee (2021) Understanding Software Sustainability in the field of Research Software Engineering. Masters thesis, University of Huddersfield.

This version is available at http://eprints.hud.ac.uk/id/eprint/35783/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

Understanding Software Sustainability in the field of Research Software Engineering

Lee Morris

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of MSc by Research

July 2021

Copyright Statement

I. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.

II. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

III. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

Abstract

Software sustainability is an increasing issue in software engineering and we aimed to see how it is understood in research software engineering. By interviewing research software engineers (RSEs) we aimed to investigate how the research software engineering community understands and measures software sustainability and if they adopt sustainable practices in their own work. We discovered that current practices fall well short of the defined software engineering principles and practices and there's a fundamental gap between basic software engineering practice and RSE activity. This gap needs to be closed in a drive towards achieving sustainable software. We suggest the creation of a set of practices to be used by RSEs to guide them through the process of testing the sustainability of software and make them more recognisable. For future work, we suggest exploring the possibility of raising awareness of the Karlskrona Manifesto for Sustainability Design and provide some education and training in it for RSEs.

Acknowledgements

After over two and a half years since I began this journey, it's finally over! I've been doing my master's degree for longer than I've been in my current job. For what started as a part-time thing while I looked for work and tried to stop drinking, it's finally come to an end. I'd like to thank Universal Credit for stopping my dole money while I enrolled on this course. I've now been working for over two and a half years and been sober for over three.

I'd like to thank to my parents Steve and Jane, grandparents Brian & Anne, Val and my late grandad Keith. Special mention to Josh. Thanks to Colin for rescuing the whole thing.

Here's to Huddersfield Town to winning the league, England winning the World Cup and Boris Johnson losing the next election.

Lee Morris
July 2021

Contents

Copyright Statement 2
Abstract 3
Acknowledgements 4
List of Figures 7
Section 1: Introduction 9
Problem statement and motivation 9
Section 2: Background 11
Software Quality 11
Software Testing 13
Static Analysis 17
Dynamic Analysis 20
Summary 21
Section 3: Literature Review 23
Software Sustainability 23
Scientific Software Development 30
Scientific Software Testing 33
Summary 37
Study Design 38
Planning 38
Interviewees 39
Data Collection 39
Data Analysis 40
Threats to Validity 40
Section 5: Results 41
Background 41
Software Engineering Education and Training 43
Software Development Lifecycles 50
Software Testing 56
Software Quality 59
Software Sustainability 65
Sustainability Design 73
Summary 80
Section 6: Discussion 82
Background 82

Software Engineering Education and Training 83
Software Development Lifecycles 84
Software Testing 85
Software Quality 86
Software Sustainability 88
Sustainability Design 91
Section 7: Summary and Conclusions 93
Future work and directions 94
References 95
Appendix 1 106
Background 106
Software Engineering Education and Training 106
Software Development Lifecycles 107
Software Testing 107
Software Quality 108
Sustainability Design 110

List of Figures

Figure 1 – Training in software requirements 43
Figure 2 – Training in software design 44
Figure 3 – Training in software architecture 44
Figure 4 – Training in software coding 45
Figure 5 – Training in software testing 45
Figure 6 – Training in software maintenance 46
Figure 7 – Training in software configuration management 46
Figure 8 – Training in software engineering management 47
Figure 9 – Training in software engineering process 47
Figure 10 – Training in software engineering models and methods 48
Figure 11 – Training in software quality 48
Figure 12 – Training in software engineering professional practice 49
Figure 13 – Training in software engineering economics 49
Figure 14 – Time spent on requirements elicitation 51
Figure 15 – Time spent on documenting requirements 52
Figure 16 – Time spent on software architecture 52
Figure 17 – Time spent on software design 53
Figure 18 – Time spent on coding 53
Figure 19 – Time spent on software testing 54
Figure 20 – Time spent on usability testing 55
Figure 21 – Time spent on project management 56
Figure 22 – Importance of functional suitability in development of code 61
Figure 23 – Importance of efficiency in development of code 61
Figure 24 – Importance of compatibility in development of code 62
Figure 25 – Importance of usability in development of code 62
Figure 26 – Importance of reliability in development of code 63
Figure 27 – Importance of security of development of code 63
Figure 28 – Importance of maintainability in development of code 64
Figure 29 – Importance of portability in development of code 64
Figure 30 – Extensibility is the foundation of sustainable software? 69
Figure 31 – Interoperability is the foundation of sustainable software? 70
Figure 32 – Maintainability is the foundation of sustainable software? 70
Figure 33 – Portability is the foundation of sustainable software? 71
Figure 34 – Reusability is the foundation of sustainable software? 71
Figure 35 – Scalability is the foundation of sustainable software? 72
Figure 36 – Usability is the foundation of sustainable software? 72
Figure 37 – Ecosystems are under stress and declining, and this is affecting human conditions and futures? 73
Figure 38 – Sustainability is a reasonable approach to addressing this decline? 74
Figure 39 – Sustainability is systemic? 74
Figure 40 – Sustainability has multiple dimensions? 75
Figure 41 – Sustainability transcends multiple disciplines? 75
Figure 42 – Sustainability is a concern independent of the purpose of the system? 76
Figure 43 – Sustainability applies to both a system and its wider contexts? 76
Figure 44 – Sustainability requires action of multiple levels? 77

Figure 45 – System visibility is a necessary precondition and enabler for sustainability design? 77
Figure 46 – Sustainability requires long-term thinking? 78
Figure 47 – Sustainability is the responsibility of everyone? 78
Figure 48 – Acting as a sustainable practitioner means both reducing my footprint and increasing my handprint? 79
Figure 49 – I am currently integrating sustainable practices into my work? 79
Figure 50 – Are you aware of the Karlskrona Manifesto for Sustainability Design? 80

Section 1: Introduction

Problem statement and motivation

The sustainability of software systems has become increasingly important as technology grows and an ever-growing population becomes more reliant on it. Condori-Fernandez and Lago (2018) recognise that society has developed an increased dependency on software-intensive systems, thus becoming more distributed, heterogenous, decentralised and interdependent whilst they are operating in often unpredictable dynamic environments.

When a research area matures, it can be followed by a sharp increase in the number of results and reports that come to light and it is an important task to summarise them and provide overviews. For example, many research fields have specific methodologies for secondary studies and one example of this is their use in evidence based medicine (Petersen, 2008). This wasn't generally true in software engineering but the movement towards a more evidence based software engineering has led to a new and increased focus on methods, such as empirical and systematic research methods. Budgen et al. (2007) suggested a method utilising structured abstracts to report results.

Modern societies have become dependent on complex software and software systems, this dependency having increased over the previous two decades. This underlies many day-to-day aspects of life such as transportation, education, finance, retail, healthcare, governance and communication amongst other things (Deek et al., 2005). Scientific and engineering research is now heavily reliant on software and it's been suggested that it should be recognised as a first-class, experimental scientific instrument (Goble, 2014) due it's importance in advances in research in computational science and engineering.

However, software as a research instrument has not yet reached a level of maturity, especially when compared to the conventional tools that are used for empirical and theoretical science (Milewicz and Rodeghero, 2019). End-user developers typically develop research software, though their understanding is limited as is their application of basic concepts, principles and techniques of software engineering. This is coupled with a "code-first" approach to development, partly driven by the perceived complexity and uncertainty of the problem (Hannay et al., 2009). Leading to research software having suboptimal design, occidental complexity, code smells, technical debt and the increased risk of software entropy. This approach isn't without consequence, and could lead the way to stagnation, decay and the

long-term decline of investment for essential research software. It is widely recognised that a resilient ecosystem of software is required for the future of scientific and engineering enterprise (Monteith et al., 2007).

Research Software Engineering (RSE) attempts to create software that is well-designed, reliable and efficient in order to solve the problems that are posed by research. However, evidence to show that research software is well designed, maintainable, understandable and extensible is rather scarce. A “code-first” approach has its consequences, leading to a range of rotten symptoms, such as software rigidity, frailty, immobility and viscosity. This results in costs due to high maintenance and evolution, which are the base of software decay and death in all software investment.

The definition of sustainability is something that’s been discussed at length and the conclusions are that there’s many ways to characterise it (Somerville, 2004). The general definition of sustainability is the “capacity to endure” (Oxford English Dictionary, 2003). Further to this, to increase understanding, it has been suggested to reflect on four points when discussing sustainability. Tainter (2006) suggested the four points were; What should be sustained? For whom? For how long? At what cost? There’s also a widely accepted characterisation proposed by Brundtland (1987), which emphasises its focus on ‘meeting the needs of the present without compromising the ability of future generations to meet their own needs’. This empathises the opinion that development has effects that lie outside the system to be developed. It is important to highlight that the word ‘need’ is vital to this definition and includes a dimension of time, present and future (Venters et al., 2017).

The main aim of this study is to survey and explore the knowledge of Research Software Engineers (RSEs) in relation to software sustainability, software quality, software testing, software engineering principles and practices, software development lifecycles and sustainability design.

This dissertation is structured as follows; Section 2 provides background to software quality, software testing, static analysis and dynamic analysis. Section 3 looks at software sustainability, scientific software development and scientific software testing, providing a literature review on the three topics. Section 4 deals with the methodology and study design. Section 5 focuses on the results of the study. Section 6 is a discussion and analysis of the results. Section 7 sees a summary and conclusion of the study which recommends for the next steps towards a more comprehensive and consistent perspective on sustainability.

Section 2: Background

This section looks at the background on software quality, software testing, static analysis and dynamic analysis in order to provide some insight into those four areas from their inception up to the present day. We also take a look at the present state of these four areas in software engineering and what could come in the near future.

Software Quality

Software quality has been defined as *"the degree to which a system meets specified requirements or user needs and expectations"* (Thoppil, 2018). Another definition is "*a field of study and practice that describes the desirable attributes of software products*" (ASQ, 2020).

Software quality dates back to the 1950s, when it was in its infancy. It was very much a one-man affair at this time, with that one person being the designer, developer and tester all at once (Huynh, 2020). This individual was usually a scientist and as the same person was involved in each step, quality assurance was very efficient, very focused and very user-centric, with the fastest possible feedback cycle (Hannula, 2016). During these times, work continued on the software until the program "just ran". Quality simply meant resolving all the bugs that were encountered during the development of the software.

By the 1970s, this approach had progressed and quality would be achieved when all requirements had been met, such as all cases described by the end user, or a technical, functional or design status. It was around this time that a strong link between quality and testing was established. Quality started to be measured against the success of a test and the tests would explore all features in order to reach the quality that had been targeted (Huynh, 2020).

Testing teams began to increase in size, as did the professions, and this led to the birth of the "software developer" (Hannula, 2016). The software developers would receive concepts from the designers and then give the testers programs to confirm and validate. This was a vertical relationship where each interviewee would do their job and then hand over their findings to the next interviewee. Although this worked, it wasn't a very efficient way of working, leading led to lengthy delays, delayed testing, costly corrections and aversion to change (Hunyh, 2020).

Software quality models can also be traced back to the 1970s with the introduction of the McCall Model (1977) and the Boehm Model (1978).

By the 1980s, a number of developments happened.

- Static and dynamic analysis was introduced and began to be employed regularly, which assessed the inner quality of the source code.
- Early testing was also utilised, with the idea that bugs that are detected early cost much less to fix than those detected much later on.
- Automated testing also became more used. This was a benefit as many low-level tests can be automated when they are described properly.
- As hardware became more powerful, the computing power allowed it to run testing campaigns on several different versions of a program on several different targets. This meant that a function's behaviour had to be the same across multiple contexts and also be tested in those contexts.
- There was an increased focus on agility, which made propositions to cancel the 'tunnel effect', connect the stakeholders and keep them engaged.
- Quality also became part of the process, with a continuous quality by integrated approach where all development lifecycle components are accessible and the quality expands from its code and test-centric definition to include more elements from the project.

Currently, there are two main approaches to software quality and they are defect management and the software quality attributes approach.

The software defect management approach is based on counting and managing defects. Software defects are regarded as any failure to address end-user requirements (ASQ, 2020). There are many defects that occur in software engineering and this can include coding errors, design errors and process timing errors. If requirements have been missed or misunderstood, a software defect is quite likely to occur.

In ISO/IEC 25010:2011, there is a product quality model where the software quality attributes approach is set out. The model sets out a hierarchy of eight quality characteristics; performance efficiency, functional suitability, usability, compatibility, security, reliability, portability and maintainability. These characteristics are then broken down and composed of a number of sub-characteristics. These attributes facilitate the measurement of a performance of software product by software testing professionals. Software architects are enabled by high

scores in Software Quality Attributes, guaranteeing that a software application will perform as the specifications provided by the client (Codoid, 2019).

Software Testing

Software testing is an investigative activity used to identify and eliminate defects, aiming to ensure the quality of software by performing analysis of a product in order to identify and eliminate continual and persistent errors (Mori, 2020).

The start point of software testing is rather vague but it can be traced back to as early as 1950, though was very different to current practice. There was no distinction between debugging and testing, the focus was simply on fixing bugs. The developers would write the code and if they faced any errors they would analyse and debug the issues. The objective was simple: get the application to work without crashing the system. There were no testers or a recognised concept of testing at this time. Software testing was defined as "*what programmers did to find bugs in their programs"* (Lewis, 2017).

In 1988, Gelperin and Hetzel (1988) classified the phases and goals of software testing into stages;

- -1956 - The Debugging-Oriented Period
- 1957-1978 - The Demonstration-Oriented Period
- 1979-1982 - The Destruction-Oriented Period
- 1983-1987- The Evaluation-Oriented Period
- 1988-present - The Prevention-Oriented Period.

Alan Turing (1950) wrote the first article discussing software testing in 1950, addressing the question "How would we know that a program exhibits intelligence?". He felt that the test consists of an interrogator who has been given the task of interrogating a human player and a computer player and then determining which is which. If the interrogator is unable to reliably distinguish between the machine from the human, then the machine has passed the test.

The Demonstration-Oriented Period

By 1957, the distinction between debugging and testing had been recognised when Charles L. Baker spoke about them while reviewing Dan McCraken's book *Digital Computer Programming* (Baker, 1957). McCracken's book is recognised as the first general textbook on programming, where he explored a number of programming techniques including program checkouts. This comprised of two separate goals;

1. Debugging - making sure the program runs.
2. Testing - making sure the program solves the problem.

By 1958, the first software test team had been put together by Gerald M. Weinberg, who was working as manager of Operating Systems Development for Project Mercury, recognised as the first human spaceflight program of the United States (Fingent, 2018).

During this period, the meaning of both “debugging” and “testing” included efforts to detect, locate, identify and correct faults (Gelperin and Hetzel, 1988). The distinction between debugging and testing was reliant on the definition of success. A destruction model emerged from this practice and it regrouped these tasks so that all fault detection was included in “testing” and all fault location, identification and correction was included in “debugging”. Despite the meanings of “debug” and “test” being realigned, there was still a desire to “make sure the program runs”. Gelperin and Hetzel (1988) have acknowledged that what was once called “debugging” is now referred to as “sanity testing”.

At this time, testers would generally follow a happy path and as time went on, employers would specifically request testing skills for prospective jobs and software test engineering started to become a career (Bulldiser, 2016). By 1972, the first software testing conference was hosted at the University of North Carolina by Bill Hetzel, leading to a series of academic testing workshops, the first of which was hosted in 1978 in Fort Lauderdale (Gelperin and Hetzel, 1988).

The Destruction-Oriented Period

Glenford J. Myers published the software testing book, *The Art of Software Testing*, in 1979 and it was the first of its kind, defining testing as *“the process of executing a program with the intent of finding errors”*. It also discussed the method of sad path testing, where the use of test data with a high probability of causing problem failures increases the likelihood of finding problems in the program. Around this time, the term “testing” started to become a broader term, moving beyond checkout and relating to things such as software analysis and review techniques (Bulldiser, 2016).

The Evaluation-Oriented Period

By 1983, the Institute for Computer Sciences and Technology of the National Bureau of Standards had produced their *Guideline for Lifecycle Validation, Verification and Testing of Computer Software.* This described testing as a methodology that includes analysis, review

and testing activities. During the same year, a national standard on test documentation (ANSI/IEEE STD 829) was published. This was followed in 1987 by a national standard on unit testing (ANSI/IEEE 1008).

The Prevention-Oriented Period

By the late 1980s, the methods had developed once again and some of these practices are still in place today. The focus of software testing shifted to preventing program failures before they actually occur. This was set out in The Systematic Test and Evaluation Process (STEP) methodology, where testing activities are performed parallel to development activities. It becomes normal for testing activities to include planning, analysis, design, implementation, execution and maintenance. It became clear that detecting defects in the design phase is far less expensive to deal with than if defects are detected at the implementation phase. This saw the introduction of test driven development (TDD), where tests are created first and then shared with programmers leading to less bugs turning up in code.

Some academics seem to agree that we are still in the Prevention-Oriented Period although quite a lot has changed in software testing since the late 1980s. The start of this period coincided with the increased usage of the PC, which allowed programmers to write for a single hardware platform for the first time (Tozzi, 2016).

This continued into the 1990s, though the PCs weren't identical and meant that the hardware and software of each machine could vary widely. The dilemma at this point was that programmers were faced with increased pressure to release software that performed well on any type of computer that was advertised as PC-compatible (Tozzi, 2016). Exploratory testing also became common in the 1990s, where the tester explored and understood the software in an attempt to locate more bugs (Ullah, 2019).

Ullah (2019) has suggested that the Prevention-Orientation era ended in 2000 following the rise of new concepts of testing such as test-driven development and behavioural-driven development. They also recognise that the advent of automation in 2004 was a major revolution in testing. *Test-driven development* emerged in 1999 when Kent Beck advocated a "test-first development" in his book, *Extreme Programming Explained* (Beck, 1999) and then published, *Test-Driven Development by Example,* in 2002, where he explained that test-driven development was the idea of testing the program before it was written (Beck, 2002).

It has been said that Beck 'rediscovered' the process as a prototype of TDD can be traced back to the 1960s and the mainframe era. During this time, code would be entered onto punch cards and the programmers' time with the machine was limited so they'd have to maximise the time they were given. One of the processes was to note down the expected output before entering the punch cards into the computer and when the mainframe outputted the results, the results could be checked immediately by comparing the actual output to the expected output which had been documented earlier (Barber, 2012).

The modern TDD is quite similar where the tests are created before the code is delivered and the critical driver of development is passing the tests (Garnage, 2017). TDD is a process for developing software, with the goal being to keep code quality high and generally has four steps; write a failing test, write the code so the test passes, refactor and repeat (Barber, 2012).

Behavioural-driven development (BDD) emerged in the early 2000s and was first introduced by Dan North in 2006. While using TDD he kept encountering confusion and misunderstandings and decided to present TDD in a way that got straight to the good stuff while avoiding the pitfalls (North, 2006). North's response was to introduce BDD, evolving out of established agile practices, designing it to make them more accessible and effective for teams that were new to agile software delivery (North, 2006). In TDD, the thing is built right whereas in BDD the right thing is built (Trofimov, 2021).

BDD sees tests written in non-technical language to ensure that everyone understands them and then forms an approach for building a shared understanding on what type of software to build by discussing examples (Garnage, 2017). BDD usually follows a three step iterative process; firstly, take a small upcoming change to the system and talk about examples of the new functionality to explore and agree on the details of the expected result. This is known as a User Story. Secondly, document the examples in a way that can be automated. Thirdly, implement the behaviour that has been documented in each User Story. These practices are known as *discovery, formulation* and *automation*, with *discovery* dealing with what the system could do, *formulation* dealing with what the system should do and *automation* dealing with what the system actually does.

These processes are now common within software engineering but Ullah (2019) has also recognised that the industry is moving towards testing using artificial intelligence tools and cross-browser testing, using tools such as SauceLabs and Browserstack. While the use of

artificial intelligence in software engineering is still in its infancy, it is on the rise (Functionize, 2021).

Currently, autonomous and intelligent agents, known as "bots", are being used to automate activities such as modelling, application discovery, test generation and failure detection and this has been on the rise since 2014 (King, 2020). It is a combination of machine learning techniques that are being used to implement the "bots", including decision tree learning, neural networks and reinforcement learning. A number of AI-driven testing approaches have emerged during the past decade including; differential testing, visual testing, declarative testing and self-healing automation (King, 2020).

Edwards (2020) has suggested that artificial intelligence is "software testing's new best friend" and predicts that artificial intelligence will transform software testing for the better. Whereas Merrill (2019) has recognised that the surface area for testing software has never been so broad and machine-based intelligence will be the most vital solution to overcome increasing testing challenges. He predicts that testing tools will change and the tester, in its current guise, will become extinct. Edwards (2020) has a slightly different view and predicts that the role of the tester will become more strategic with test engineers guiding the testing process, playing a key role in deciding what makes software good for the user.

Static Analysis

Static analysis is a method of debugging completed by examining the code before the execution of a program and has been defined as "*a program analysis approach to find software flaws without executing the program code*" (Ahmed et. al, 2020). Static analysis involves no dynamic execution of the software under test and has the ability to detect possible defects at an early stage, before the program is executed (Ghahrai, 2017). The process involves a set of methods that are used to analyse the source code or object code, determining how the software functions and also establish criteria that checks code correctness.

This process helps to provide an understanding of the code structure, helping to ensure that the code is up to industry standards by comparing it against a set of coding rules or standards (Asahara, 2020). It is used in software engineering by software developers and can be helpful in finding issues such as programming errors, security vulnerabilities, undefined values and coding standard violations. Static analysis views things from the inside out (Asahara, 2020).

Static analysis is quite simple, as long as the process is automated, and takes place before software testing in the early development of the code. Usually, once the code is written, a static code analyser is run to check the code. The code is then checked against defined coding rules from standards or custom predefined rules (Gillis, 2018). The analyser will identify whether the code complies with the predefined rules. This can cause some problems with false positives being flagged during the process and it means that human involvement is important as they must go through and dismiss or deal with them accordingly. Once the process is complete, the developers begin the process of fixing any mistakes that have been identified and once these mistakes are dealt with, the code can be tested.

Static analysis relies on having code testing tools available and if they weren't available, humans would have to carry out the whole process manually, reviewing the code and working out how it will behave. This highlights the importance of automated tools to static analysis. There are a number of those tools out there and readily available, all with different uses and aims.

Upadhyay (2020) identified four main static analysis methods used in software engineering; control analysis, fault/failure analysis, data analysis and interface analysis. *Control analysis* checks the sequence of control transfers by inspecting the controls used in calling structure, control flow analysis and state transition analysis. *Fault/failure analysis* looks at faults and failures and analyses them. Then it uses input-output transformation to identify what conditions have caused the failure. *Data analysis* makes sure that proper operation is applied to data objects, ensuring that the defined data is correctly used. This method checks the definition and context of variables. Finally, *interface analysis* uses interactive simulations to check the code.

Hicken (2018) identified two main types of static analysis, pattern-based static analysis and flow analysis. *Pattern-based static analysis* is utilised to search for patterns in the code and report them as possible errors. It is rare for this method to suffer from the issue of false positives. *Flow analysis* is quite different as it checks for problematic constructions against a set of rules, while decision paths are simulated to search deeper into the application and root out defects that are difficult to discover. This method is good for discovering real defects but false positives are often generated using this method.

There are also additional types of static analysis out there, such as metrics analysis tools and coverage analysis tools. *Metrics analysis tools* simply measure the characteristics of the code.

Coverage analysis tools track unit and application tests to gain more understanding of how effectively the code has been tested. These two methods provide significant visibility into the safety, security and reliability of an application when used together as part of an automated development testing process (Hicken, 2018).

Gomes et al. (2009) think that one of the backbone pillars behind software development is the use of analytical methods to review source code in order to correct implementation bugs. It is acknowledged that in the beginning, there was no conscience on how necessary and effective a review may be in software engineering. However, by the 1970s, this had changed and it had been recognised that formal review and inspections were important for both product quality and productivity. They soon became standard practice in software engineering and adopted by development projects (Kan, 2003). It soon became clear that the approach to software engineering saw that more reliable and efficient programs were down to removing defects in the early stages of development.

Static analysis tools are often used to support software developers, helping them detect and resolve the vulnerabilities of software and security. Static analysis research has introduced increasingly complex analysis methods and tools that support a growing number of programming languages, libraries and coding concepts in recent years. This means that results are returned faster and with better precision (Nachtigall et al., 2019).

Static analysis is recognised as a very useful tool within software engineering with a number of benefits. It has the ability to find code weaknesses at the exact location and can be operated by trained developers who are fully understanding of the code (Jackson, 2009). It scans the entirety of the code and finds vulnerabilities even if they exist in the distant corners of the application (Parker, 2020). It also scans the entire code base and can provide mitigation recommendations, which helps to reduce the research time (Jackson, 2009). Also, automated tools are less prone to human error and can be done in an offline development environment (Gillis, 2018). Most importantly, static analysis aids the discovery of defects early in the software development process, meaning that the cost of fixing them is often reduced.

For all its benefits, there are some drawbacks to using it however, with the main one being the identification of false positives or negatives. This is the main disadvantage of using static analysis as it creates further work for developers, who deem them as low priority and eventually stop trying to fix them (Parker, 2020). It can also be time consuming if conducted

manually, the tools don't support all programming languages, and add a false sense of security that everything is being addressed (Jackson, 2009).

Static analysis tools have potential benefits that are beyond question, but their usability is often criticised and this prevents developers from using static analysis to its full potential (Nachtigall et al., 2019). Usability issues have plagued the last decade, with one example being ill-explained warning messages, highlighting the need for understanding how to design analysis tools that can satisfy the developers and the gap between the academically perceived potential of static analysis and its use in practice, needs to be overcome.

In conclusion, static analysis is precise and sound and although the results may be weaker than what is considered desirable, they are guaranteed to generalise future executions (Ernst, 2003).

Dynamic Analysis

Dynamic analysis analyses the properties of a program that is running, executed while the program is in operation (DuPaul, 2013). This is in contrast with static analysis, which examines the text of a program in order to derive properties that hold for all executions. Dynamic analysis derives properties that hold for one or more executions by examination of the running program, usually through program instrumentation (Larus and Bell, 1994). Dynamic analysis looks at things from the outside in (Asahara, 2020), and examines it in its running state, trying to manipulate it in order to discover vulnerabilities in the security (DuPaul, 2013).

It is a technique of debugging that examines an application during or after the program has been run, aiming to understand the behaviour of software systems by exploiting execution data. It cannot be performed against a certain set of rules because the source code has the ability to run with any number of different inputs. The testing simulates attacks on an application, then it analyses how the reaction of the application (Asahara, 2020). This is done to discover where the application is vulnerable. Along with static analysis, dynamic analysis is a method of analysing source codes for possible vulnerabilities. Whereas static analysis sees the code being analysed before anything is run, dynamic analysis sees the code analysed while it's being run.

Although dynamic analysis is unable to prove that a program satisfies a particular property, it has the ability to detect violations of properties as well as providing useful information to programmers about the behaviour of their programs (Ball, 1999). Dynamic analysis is precise

and the analysis is able to examine the actual, exact run-time behaviour of a program (Ernst, 2003). The most common practice of dynamic analysis is the execution of Unit Tests against the code to identify any errors in the code (Ghahrai, 2017). Other practices include integration testing, system testing, acceptance testing and regression testing.

Dynamic analysis has its benefits, with one of the main one benefits being the ability to detect vulnerabilities that are either too subtle or too complex to be identified by static analysis. Static analysis tends to miss sophisticated memory handling errors such as indexing beyond array bounds and memory leaks. Dynamic analysis tends to find these errors. It also has the ability to analyse applications even if there's no access permitted to the actual code.

Dynamic analysis also has the ability to identify defects in a run time environment, as opposed to static analysis that is unable to do this. There is also less of a problem with generating false positives. It can also be run against any application and does not require access to the actual code to perform dynamic analysis, saving both time and resources.

However, there are some drawbacks with the use of dynamic analysis and similar to static analysis, automated tools can provide a false sense of security that everything is being addressed (Ghahrai, 2017). Although dynamic analysis produces less false positives and false negatives, it still produces some and these need to be dealt with manually. Automated tools are generally only as good as the rules they are using to scan with (Jackson, 2009) so dynamic analysis relies on a strong set of standards and rules. And perhaps most importantly, dynamic analysis identifies defects after the program has been run, usually during testing, but these coding errors may not surface during the testing. This means that there are defects that dynamic analysis might miss that could be picked up by static analysis (Bellairs, 2020).

In conclusion, despite the disadvantages mentioned, dynamic analysis is efficient and precise, it doesn't require costly analyses and despite requiring selection of test suites, it provides results that are highly detailed regarding the test suites (Ernst, 2003).

Summary

Software quality dates back to at least the 1950s but progressed throughout the latter parts of the 21st century, helped by the introduction of static analysis, dynamic analysis and the increased use of software testing. There are currently two main approaches to software quality, which are defect management and the software quality attributes approach.

Software testing dates back to at least 1950, with the start point being rather vague. The original concept is very different to the concept that is used in current times. There have been a number of different recognised periods of software testing; The Debugging-Oriented Period, The Demonstration-Oriented Period, The Destruction-Oriented Period, The Evaluation-Oriented Period and The Prevention-Oriented Period. It has been suggested that the Prevention-Oriented Period ended in 2000 following the rise of new concepts of testing which include test-driven development and behavioural-driven development. However, the rise of artificial intelligence is predicted to transform software testing for the better, seeing the role of the tester changing, becoming more strategic and guiding the process. Some even suggest it could become the new best friend of software testing.

Static analysis is the simple concept of examining the code before the execution of a program. It helps to provide an understanding of the code structure and helps to ensure it is up to industry standards by comparing it against a set of coding rules or standards. Many methods have been identified such as *data analysis, control analysis, fault/failure analysis, interface analysis, flow analysis* and *pattern-based analysis*. It has been suggested that one of the backbone pillars behind software development is the use of analytical methods to review source code to correct implementation bugs (Gomes et al., 2009). Static analysis is precise and sound and although the results may be weaker than is desired, they are guaranteed to generalise future executions (Ernst, 2003).

Dynamic analysis is the concept of analysing the properties of a running program with the aim of understanding the behaviour of software systems by exploring execution data. It is a technique of debugging that examines an application during or after the running of a program. The aim is to understand the behaviour of software systems by exploiting execution data. A common dynamic analysis practice is executing Unit Tests and also makes use of system testing, integration testing, acceptance testing and regression testing. Dynamic analysis is efficient and precise, it doesn't require costly analyses and despite requiring selection of test suites, it returns results that are highly detailed regarding the test suites (Ernst, 2003).

Section 3: Literature Review

Defining Sustainability

"The quality of being able to continue over a period of time" (Cambridge Dictionary).

Academics have attempted to define sustainability and their understanding of the word, meaning that vast amounts of literature contains such discussion.

With regards to software sustainability, the earliest discussions of sustainability in software engineering dates back to the 1968 NATO Software Engineering Conference (Becker et al., 2015), when software maintenance and evolution were discussed. There has been much discussion of how to define sustainability, with Becker at al. (2015), defining it as "the capacity to endure".

It is important to find out what previous papers have said about software sustainability and this will be explored in the next section as we take a look at previous research of software sustainability in the field of software engineering.

Software Sustainability

Software Sustainability was discussed as early as 2003 by Seacord et al. (2003) but was referred to as "software sustainment" and "software maintenance". It was noted that the two terms are often used interchangeably, with "maintenance" referring to software activities aimed at correcting software defects and "sustainment" being used as a more general term to refer to all manners of software evolution. Seacord et al. (2003) propose a Sustainment Process Model and also discuss the definition of software sustainability, deciding that the definition depends on more than just the actual code and also depends upon the sustainment organisation, the sustainment team, the customers and the operational domain that the software operates in. Sustainability is also affected by other software artefacts which include the architecture, design documentation and test scripts. A sustainability measure is a combined measure of all four areas and in conclusion, they recognise that measuring software sustainability provides a basis for measured improvement.

It was noted by Hong and Voss (2008) that reservations about the quality and sustainability of digital objects have regularly been cited as barriers to the uptake of e-Research infrastructure and tools. They are not surprised that researchers hesitate to invest in uptake

of digital artefacts as it's unclear whether and for how long they can be relied upon. They conclude that to be properly sustained, it is important that digital objects in the e-Research environment must evolve and to continue this evolution in usage, community input is necessary.

Amsel et al. (2011) defined sustainability as "*meeting the needs of the present without compromising the ability of future generations to meet their own needs*". While acknowledging that sustainable software engineering aims to create reliable, long-lasting software to meet user needs while reducing the environmental impacts, they identified three separate research efforts to investigate the area.

They investigated to what extent the users thought about the environmental impact of their software usage. Then, they created GreenTracker, a tool that measured the energy consumption of software to raise awareness about the environmental impact of software usage. They also explored the indirect environmental impact of software to try and understand how software affects sustainability beyond its own power consumption. They also noted that the relationship between environmental sustainability and software engineering is a complex one and that understanding both direct and indirect effects is critical to helping humans live more sustainably. They also looked at the theory that the goal of sustainable software engineering is to create better software to ensure that the opportunities of so future generations are not compromised. Interestingly, they observed that sustainable software seems to be a minor concern for most users.

Penzenstadler et al. (2012) recognised that the most common definition of sustainable development was "*to meet the needs of the present without compromising the ability of future generations to satisfy their own needs*". They suggested that sustainable development has to satisfy the requirements of the three dimensions of society, economy and environment. They also suggested it needed to satisfy a fourth requirement of human sustainability whilst recognising it is discussed more less publicly but ought to be included as it is the basis for the other three dimensions. To make up for the lack of literature about software sustainability, they proposed a Body of Knowledge for Sustainability in Software Engineering to include related application domains and sustainability concepts from related disciplines that could be used to learn from while further investigating sustainability in software engineering.

Lami and Buglione (2012) acknowledged the growing attention that is being paid to the increasing global carbon dioxide production and the subsequent issue of sustainability in ICT-related projects. They stated that although the attention was growing, there was hardly any information and hardly any time being spent to determine a set of related measures from sustainable processes. To combat this, they proposed a sustainability measurement plan, aiming to integrate further information elements to improve the decision-making process. They recognised that sustainability represents one of the most viable concepts to be constantly analysed and understood for being profitably applied to any production process. They also noted that sustainability is mostly associated with environmental issues and that it should actually be seen from a wider perspective, taking wider viewpoints into account. They also come to the conclusion that because of the ongoing global economic crisis and the future of it, sustainability is no longer an option. They suggest that organisations should monitor and control themselves and that sustainability should be managed from different viewpoints and that different entities should be considered. These entities are; project, resources, process and product but the main resource is people who they believe are too often wrongly positioned at the same level as other assets.

Calero et al. (2013) made similar observations that sustainable software that has direct and indirect negative impacts on the four dimensions, economy, society, human beings and environment, resulting from development, deployment and usage of the software are minimal and/or which have a positive effect on sustainable development. They also acknowledged that while sustainability is a standardised practice in a number of engineering disciplines, the software engineering community currently has no awareness of it. This is similar to the observations of Amsel et al. (2011) but they observe that software users don’t have much interest in sustainability rather than the wider software engineering community.

Crouch et al. (2013) acknowledged that software is critical to research but many researchers are still to be convinced of the importance of developing well-engineered software. They make reference to the Software Sustainability Institute, which was established in 2010 to support UK researchers, but call for the introduction of worldwide research software support initiatives with coordination at both national and international levels. They believe it is clear that in many disciplines, the critical mass of expertise and development effort can only be successful if the community of users consolidate on a few community codes. To achieve this, coordination and cooperation is required internationally along with the ability of national funders to support software that has been developed elsewhere.

Roher and Richardson (2013) stressed that sustainability is not sufficiently considered in the development of modern software systems. They argued that, despite the looming threats of global climate change and environmental degradation, software companies are far more concerned with product "time-to-market" than longer-term impacts of the products. They aimed to overcome the barriers of incorporating sustainability into the software engineering process through the use of a recommender system which would be employed during requirements engineering. The aim of the system would be to recommend the kinds of sustainability requirements that should be considered in a given system. This would be based on application domain and deployment locale amongst other things. They hoped that this system would decrease the workload of eliciting appropriate sustainability requirements. In conclusion, Roher and Richardson (2013) felt that their suggestions would advance the stakeholders' ability to produce software systems that have less negative impact on the environment. It would also increase the stakeholders' general knowledge of sustainability and related types of requirements to consider.

Lago et al. (2015) defined sustainability as the "capacity to endure" and "preserve the function of a system over an extended period of time". They also recognised that discussing sustainability requires a concrete system or a specific software-intensive system. In their 2015 paper, they suggest that analysing the sustainability of a specific software system requires the developers to weigh four major dimensions of sustainability. These four major dimensions are; economic, social, environmental and technical. The economic, social and environmental dimensions stem from the Brundtland report and the fourth is included for software-intensive systems at a level of abstraction closer to implementation.

Condori-Fernandez et al. (2014) also explored the theory that sustainability definitions are based on the four dimensions; environmental, social, economical and technical. They set out the relevance between software sustainability and the four dimensions, observing that Environmental Sustainability looks at improving the welfare of humans, while protecting natural resources and optimising the amount of energy used. Whereas social sustainability focuses on supporting both the current and the future generations to have the same or even greater access to social resources by pursuing generational equity. With regards to technical sustainability, they believe that it addresses long-term use of software systems and their appropriate evolution in an execution environment that is continuously changing and economic sustainability maintains a focus on the preservation of capital and economic value.

Venters et al. (2014) argued that the current understanding of software sustainability bares similarity to the parable of the Blind Men and the Elephant, where there isn't an agreed definition of the meaning of software sustainability and how it can be demonstrated and/or measured. For reference, the Blind Men and the Elephant is the story of six blind men who touch only one part of an elephant in order to learn what it is like and based on their individual experiences, they suggest that the elephant is like a wall, spear, snake, tree, fan or rope. Following this, they compare their experiences, learning that they are all in complete disagreement with each other (Saxe, 1963). The paper acknowledges that there isn't an agreed nor definitive definition of software sustainability and how it can be measured and demonstrated. It's a vague term and isn't well understood within the software engineering community, they acknowledge that individuals, groups and organisations hold diametrically opposed views (Penzenstadler and Femmer, 2013). This ties in with Penzenstadler and Femmer's view that if there isn't a clear and commonly accepted definition of sustainability, contributions remain somewhat insular and isolated, and this has the potential to lead to ineffective and inefficient efforts to address the concept or result in its complete omission from the software system.

Penzenstadler et al. (2014) recognised that the definition of sustainability is the capacity to endure but also noted that context was required to interpret this. They go on to discuss the UN's definition of sustainable development as "meeting the needs of the present without compromising the ability of future generations to meet their own. They recognise that sustainability in software engineering must consider the second and third-order impacts of software systems, while it encompasses energy efficiency and green IT. To do this, sustainability must be considered as a first-class quality attribute and be specified as a non-functional requirement of IT systems.

Becker et al. (2015) recognised that the concept of sustainability is employed by numerous different communities and often ambiguously. Whilst mentioning that the Latin origin of sustainability was "sustinere" and it was used as both endure and furnish, they noted that sustainability refers to the 'capacity' of a system 'to endure' in modern English. They also mention that the 1968 NATO SE Conference decided on the definition of sustainability as "the capacity to endure". Becker et al. recognise that in order to achieve progress in understanding the role that software plays in the choices made by designers, there is a need to understand the nature of sustainability and find a common ground for a conceptual framework. They present the Karlskrona Manifesto for Sustainability Design, which is presented as a vehicle for much needed conversation about sustainability amongst the

software community. The Karlskrona Manifesto unveils and straightens out a number of common misunderstandings that relate to sustainability and software engineering.

Gesing et al. (2017) recognise that addressing sustainable software includes some diverse topics such as development and community building to increasing incentives for better software to making the existing credit and citation ecosystem work better for software. They believe that this can be addressed by software engineers following the best practices for sustainable software. They proposed "Bootcamp", an intensive week-long workshop for the leaders of gateways who wish to further develop and scale their work. This workshop proposed education on sustainability strategies and even assembled a team of experts that included a sustainability expert for digital resources. Following the completion of the "Bootcamp", the results found that in order to reach sustainability, providing the stage for more community interaction and further distinctive tasks is very important.

Winters (2018) has said that code is sustainable if there is the capability to update it as a response to necessary change for the expected lifespan of a project. They recognise that programming projects are usually short-term and don't really require any necessary change, in contrast to long-term projects, for which it's entirely possible that the code is required to live for years or even decades. They have observed limited awareness of sustainability issues amongst API providers or consumers and because of this limited awareness, they expect their code to work indefinitely, simply because it's working in the present. Winters (2018) feels that this approach will only work if everything is stopped and focus shifts to stability over everything else. They think that the belief in perfect stability is foolish and a conceptual approach that makes the responsibilities of both providers and consumers clear over time is what is needed.

Winkler (2018) recognised that software continues to have an increasing impact on individual well-being, the environment and society in general. It has been suggested that to mitigate this potentially harmful impact, we should go beyond traditional requirements and one of these approaches is to consider human values as requirements in the process of software development. These can be used to constitute dimensions of sustainability but even to this day, there's no consensus on which values that should be or where the relevant information can be found about them. Winkler identifies a list of values that can facilitate sustainable software development and how these values can define the social sustainability dimension.

Lago (2019) was also of the opinion that it is possible to define sustainability as the “capacity to endure” and “to preserve the function of a system over an extended period of time”. However, they expanded on this and acknowledged that those definitions tend to point towards technical sustainability over time and that sustainability could entail a much broader scope including social, economic and environmental sustainability. They also observed that we are lacking the suitable instruments to design sustainable software systems. They suggest sustainability should be treated as a software quality property and a software sustainability assessment method should be defined to fill this void. This method relied on decision maps, which are views that are aimed at framing the architecture design concerns around the technical, economic, social and environmental sustainability dimensions. Lago puts forward the idea that decision maps should be used to frame the sustainability-related architecture design concerns.

Condori-Fernandez and Lago (2019) note that a significant effort has been invested in providing principles and common basis of software sustainability. They also recognise that one of the key challenges for software sustainability is its characterisation as a software quality requirement (Lago et. al, 2015) and the ability to identify the impact of quality requirements on sustainability is a first step towards developing software intensive systems that fulfil sustainability requirements. Sustainability has been linked to software evolvability or longevity whereas software sustainability is much broader and is commonly defined in terms of social, technical, economic and environmental dimensions. They also note that these dimensions are tightly interdependent (Condori-Fernandez and Lago, 2018) but not all sustainability dimensions need to be addressed together in order to guarantee sustainability and the relevance of those dimensions can depend on the type of software system.

Carver et al. (2021) give a slightly different definition of software sustainability. They define it as the ability to ensure the usefulness of software over time, by fixing bugs, adding features and adapting to changes in both software and hardware dependencies. When software is sustainable, it increases reuse of the software and decreases repetitive work on it. They recognised that in order to study software engineering practices, it is entirely necessary to examine both the software and those that develop it. They emphasise that the use of good software development practices increases the likelihood of software being sustainable and recognise that there’s a need for developers to maintain software to keep it up to date with technology and the needs of users. This tells us that software sustainability relies on software developer sustainability. Carver et al. also understands that although Research Software Engineers can contribute to software sustainability, their overall status reflects that

of research software and this can cause problems. As there is a focus on research, helped by research grants, there isn't always the time or the focus on software sustainability and this leads to research software developers suffering from the lack of institutional support for developing software that is sustainable. Also, as there is frequent turn-over of developers, it leads to software redevelopment rather than reuse, which sees sustainability often overlooked. There is also the challenge of a lack of formal RSE training at undergraduate level and of professional certifications at postgraduate level. This has led to limited awareness of practical challenges faced while developing research software and the sharing of best practices and lessons learned is limited. This is limiting the emergence of a professional class of career Research Software Engineers.

As part of an International Workshop, Venters et al. (2021) recognised that there's no agreement on the definition of software sustainability or how sustainable software can be achieved and although a number of definitions have been suggested, the concept remains an elusive and ambiguous term with individuals, groups and organisations holding diametrically opposed views (Venters et al., 2014). The lack of clarity here leads to confusion, and potentially to ineffective and inefficient efforts to develop sustainable software systems. Because of this, they call for a clear definition of software sustainability and this must be addressed in the event of the development of a Body of Knowledge on Software Sustainability, for which the first workshop took place in 2021.

Scientific Software Development

Scientific software development has received increased attention over the past decade, which has included special issues of IEEE Software focusing on scientific software, an E-Science issue of IEEE Computer and a small number of academic papers on the subject.

Segal and Morris (2008) believe that the development of scientific software is fundamentally different from the development of commercial software. It's far easier to understand the requirements of commercial software compared to scientific software and because of this, a scientist must be heavily involved in the development of scientific software, simply because the average developer has a lack of understanding of the application domain. This means that the scientist is usually the developer. Another difference relates to the requirements, with HR people knowing what they want (basically understanding their domain). However, this might not be the case for scientists as the purpose of scientific software is often to improve domain understanding and full up-front requirements specifications are impossible. There are also problems with the lack of test oracles, an understanding of the domain which is incomplete

and the lack of cohesion between scientific software development and software engineering in general.

Sanders and Kelly (2008) recognise that scientific software development involves risk in the underlying theory, the implementation of it and the use of it, following interviews with scientific software developers. They aimed to identify the characteristics of the development of scientific software, whilst taking notice of the diversity of them and any interesting correlations between them. They found that knowledge implementation choices along with an iterative feedback development process provide a strong base for code development. They also acknowledged that care used by the developers in developing their theories transfers to the writing of the code. However, systems are growing past the ability of small groups of people to understand the concept and better testing methods are required for this and Sanders and Kelly feel that testing has the greatest potential for employing risk management.

Nguyen-Hoan et al. (2010) acknowledged the increased attention that scientific software development has received and that various problem areas have been identified. They attempt to build on previous research to identify where improvements could be implemented in scientific software practice by surveying 60 scientific software developers. Nguyen-Hoan et al. found that the situation in scientific software development has recently improved and although version control and IDEs are employed by around 50% of developers, further improvements in the uptake of these and other tools are entirely possible. Also, the cost to benefit ratio of documentation in the development of scientific software is something that can have improvement, though the time and effort that is required is still a hurdle. They also found that there's some testing and verification activities that could be used more widely by developers, including integration testing and peer review. In conclusion, any improvements in the development of scientific software must take reliability and functionality into account, with these two factors being the top two non-functional requirements that are considered the most important by the developers of scientific software.

Ackroyd et al. (2008) believe that the development of software in a scientific environment has a specific set of challenges and while it is frustrating at times, it is particularly fulfilling. They observe that the development of scientific software is sometimes an individual activity but the use of common libraries and object packages has helped to reduce development time. Despite developers within collaborations having individual tasks, collaborating allows ideas to be shared, broadens knowledge within the team and provides insights into different perspectives. In conclusion, scientific software development requires a flexible and pragmatic approach and

scientists in an experimental research environment need to develop requirements for their ongoing research. They believe that although the scientific research environment is driven by individuals that are highly focused, success in software projects calls for a close relationship between the developer and the scientist. They concede that attempts to impose this is unlikely to be a success but one way would be to build up mutual respect between the two and having a flexible approach to requirements can lead to good working relationships and a satisfactory final product. Ackroyd et al. believes that developing software in this environment is fun and creative.

Loynton et al. (2009) recognise that the importance of software tools that are both usable and useful tools is greater than it's ever been but concedes that there are practical constraints in academic scientific software developments projects. They identify the complexity and diversity of the working environment and practices of scientists. They also mention funding initiatives that tend to reward development teams that focus on innovative technology development over evaluation, refinement and promotion of existing systems as two of those practical constraints. These then lead to local 'custom' solutions rather than larger 'community-wide' solutions (Cassman et al., 2005) and a requirement to match software methodologies to problem domains (Glass, 2004). There have been attempts to address the issues that surround the constraints in the development of scientific software. The UK Engineering and Physical Sciences Research Council (EPRSC) have funded four projects that investigated the challenges of promoting usability in e-science. Loynton et al. focus on the Usable Image project, which applies a range of user centred design methods to an existing scientific software development project to explore novel approaches to the challenge of optimising the usability and usefulness of academic scientific software. They call for a new approach, The Project Community, which provides development teams, that are working under constraints, with a means to identify, store, share and capitalise on the data they have. This will allow them to understand more about the user and stakeholder community that their work is situated in.

Taweel et al. (2009) feels that managing and capturing research, in addition to traditional, team and project knowledge, early is critical for global scientific software development projects. They suggest that this requires tools that are suitable and sufficient and mechanisms that have been carefully developed, introduced and then adopted by the project teams. This knowledge is split into two main types, project knowledge and team knowledge and these tend to be managed using one or more than one approach. To manage this knowledge, Taweel et al. use a hybrid strategy by using a central repository for the project which is then linked to local repositories for the individual teams.

Wiese and Polato (2020), in a survey for a previous paper, found that the developers of scientific software face some problems during development of scientific software, mostly due to the community's lack of computer science background. These problems included cross-platform compatibility, poor software documentation and a lack of formal reward system. However, they felt that this question wasn't answered properly and dedicated a study to fully explore the main problems of developing scientific software. They found that traditional problems still prevailed such as reproducibility, the difficulty in defining the correct scope and the mismatch between the background skills and coding skills needed to conduct scientific work. They also discovered previously unknown problems such as emotional issues including ego and recognition and also social and scientific problems. To deal with this, Wiese and Polato called for more research on the social side of the development of scientific software.

Scientific Software Testing

Scientific software testing is a process that is used widely in the fields of science and engineering and this software plays an incredibly important role in critical decision making in a wide variety of fields such as the nuclear industry, medicine and the military (Sanders and Kelly, 2008). It is commonly developed by scientists in order to aid understanding or make predictions about real world processes (Lin et al., 2018). There are several definitions of scientific software existing in the literature, with Kreyman et al. (1999) defining it as software with a large computational component providing data for decision support. Kanewala and Bieman (2014) simply refer to it as software that's used for scientific purposes.

There's a number of types of scientific software that have been recognised, which includes research software that has written with the goal of publishing academic papers, end user application software with the aim of achieving scientific objectives and production software that has been written as tool support (Lin et al., 2018). Despite the different types of software, there are some things they share in common. Firstly, the size of scientific software generally ranges from 1,000 to 100,000 lines of code (Sanders and Kelly, 2008). Secondly, the person in charge of the project is usually a scientist and they often undertake the development of the software themselves (Morris and Segal, 2009). Thirdly, the process usually involves the scientists' development of discretised models, followed by transforming them into algorithms. A programming language is then used to code the algorithms (Kanewala and Bieman, 2014).

Cook et al. (1999) while recognising the growing need to ensure scientific software was fit for purpose and producing correct results within a prescribed accuracy, developed a general

methodology which was applicable to a range of scientific software (Cox and Harris, 1999). To test the numerical accuracy of scientific software, they proposed a general methodology which utilised the design and use of reference data sets and corresponding reference results to undertake black-box testing.

Cox and Harris (1999) acknowledged the wide use of scientific software but conceded that it is rarely tested in objectively or impartially. They recognised that some approaches for testing have been developed which address the individual software modules called directly by scientists or as part of software packages used in science. They referenced the approach that had been proposed by Cook et al. and recognised that this approach enables reference data sets and results to be generated in a manner that is consistent with the functional specification of the problem being addressed by the software.

Van Vliet (2007), in their *Software Engineering: Principles and Practice,* recognised three test techniques; coverage-based testing, fault-based testing and error-based testing. Coverage-based testing is testing coverage of the software, where all bases of the program are tested at least once. Fault-based testing focuses on the detection of faults and the fault detecting ability of the test set determines its adequacy. One example of this would be to place a number of faults in a program and then create a test set that finds these faults correctly. Error-based testing focuses on error-prone points that are based on knowledge of common errors that people make and aims the testing effort at this.

Kelly et al. (2011), following an exercise of how to test an example of scientific software, found that although there were a number of code defects that had been detected, the form and evolution of the activity itself had been the most interesting outcome. To analyse how the activity evolved, they applied a view of testing that was four-dimensional. They acknowledged that the four dimensions of test that had guided their analysis were context, goals, techniques and adequacy. The four dimensions had begun as eight, but Kelly found that there was significant overlap in the concepts that were included under each of the original dimensions, allowing the reduction from eight to four. Firstly, context saw them cast a wide net and included the historical and technical background of the software, its applications and the roles and knowledge of the developers and the users. This included the details of what needed to be tested. Secondly, goals saw them attempting to better understand the domain content of the software and how the code expressed this. As the understanding increased, they articulated more focused goals. Thirdly, techniques saw them use static and dynamic analysis, with differing positive results, and discovered the importance of including the tester as part of the

testing system, utilising their knowledge in the choice of technique. Finally, Kelly et al. found that goals determined adequacy and the teaser determined whether these goals had been satisfied. In conclusion, they developed a better understanding of how to test scientific software effectively and highlighted the advantage of considering the tester as part of the testing system. This made use of their existing knowledge of the software and helped to increase it and the combination of software engineer working with the scientists was successful as the software engineer brings ideas to the table and the scientist takes the ideas and fashions them into something that works for a specific situation.

Hook and Kelly (2009) identified the two main reasons for poor testing of scientific software, with these being the lack of testing oracles and the large number of tests that are required when any standard testing technique that's described in the software engineering literature is followed (Koteska and Mishev, 2015). They felt that these two factors highlighted the need for code faults in scientific software to be tested effectively and efficiently and that a small pool of well-chosen tests may reveal a high percentage of code faults in scientific software, which allows scientists to increase their trust (Hook and Kelly, 2009). They suggested that a goal of correctness is impractical and should be replaced by a goal of trustworthiness and aimed to identify a time-efficient and results-effective testing approach for scientific software.

Lin et al. (2018) also recognised that the oracle problem was a key challenge in the testing of scientific software. This problem is a situation in which appropriate mechanisms are unavailable for checking if the code produces the expected output when executed using a set of test cases. They acknowledged that testing hasn't been performed systematically by the developers of scientific software and the key problem to be addressed was the oracle problem. It's also noted that testing to assure the quality of software is not perceived as ultra-critical by developers and scientist developers haven't yet routinely adopted systematic testing techniques to achieve software quality.

Chen et al. (1998), while acknowledging the oracle problem, suggested metamorphic testing as a way of alleviating the problem (Chen et al., 1998). The basic idea of metamorphic testing shifts software testing from one input at a time to multiple ones whose outputs shall follow certain relationships (Lin et al., 2018).

Sanders and Kelly (2008) think that though scientific software is an outlet for scientific progress, the testing of scientific software is often anything but scientific. They identified two parts of scientific software that needed testing, computational engine and the user interface.

They also found that scientists are using testing to show that their theories are correct, rather than using it to show that their software doesn't work. They claimed that testing the computational engine is important but acknowledged the disorganisation of their work and the incorrect testing practices. They also referred to the oracle problem and acknowledged that when the oracle and the results from the software don't match, then the problem could be with the theory, the implementation of the theory, the data that has been inputted or the oracle itself. They suggested that research is needed in test-case selection methods that deal realistically with the lack of oracles amongst other problems. They found that risks to scientific software come from theory, implementation and usage, the developer often has no testing expertise and there's often a lack of knowledge in scientific domains from software engineers. Lastly, they recognised that effective and efficient methods of testing that have been specifically developed for scientists have not been put into the hands of the scientists and the interlocking risks that influence the testing of scientific software means that testing strategies from other domains cannot be imported directly. The unique challenge here is how to put effective testing strategies together to meet the goals of scientists.

Kanewala and Bieman (2018) noted that scientific software plays an important role in critical decision making, such as computation of evidence for the purposes of research publications. In the past, scientists have had to retract some of their publications due to errors that have been caused by software faults. They recognised that scientific software presents special challenges for testing and found that these challenges fell into two main categories, which were; testing challenges occurring because of characteristics of scientific software such as oracle problems and testing challenges occurring due to cultural differences between scientists and the software engineering community. To tackle this, they suggested that existing techniques such as code clone detection could help improve the testing process and called for software engineers to consider special challenges, such as oracle problems, when they are developing testing techniques for scientific software. Due to the complexity of scientific software and the required specialised domain knowledge, scientists often take up the mantle themselves and either develop the programs or have close involvement with the development. However, scientific developers may not be aware of the accepted software engineering practices and this can have an impact on the quality of scientific software, and also impact the testing of scientific software.

Lin et al. (2021) recognised that many scientific researchers rely on software to perform their research and also noted that there's many challenges faced by scientific software developers, focusing on the challenge that only a small group of scientific domain experts (SDEs), who

develop software themselves, could write the tests in relation to the scientific theories and models, and yet their primary goal is to further science. They also speak about metamorphic testing and deal with orchestrating its testing functions.

Koteska and Mishev (2015) state that the testing of scientific software is complex mainly because the results cannot be evaluated by users, but they are often compared with results from real experiments or are based on specific scientific theory. They recognised that testing is rarely performed or isn't performed if the final result is correct and that many bugs are found later, with problems arising due to the absence of software engineering testing practices. To address this, Koteska and Mishev wanted to improve the testing of scientific software, suggesting that current the current practices of scientific software testing must be altered and software engineering practices should be successfully included in scientific software testing.

Summary

Much of the literature makes reference to software sustainability being defined in terms of economic, social, technical and environmental dimensions and that these four dimensions are required to analyse the sustainability of a specific software system. Another common theme in the literature is the definition of sustainability, which has commonly been defined as the "*capacity to endure*" and to "*preserve the function of a system over an extended period of time*".

The literature overwhelmingly deals with the challenges within scientific software testing, with the two main challenges being the oracle problem and testing challenges, such as the large number of tests required when following any standard testing technique and also challenges that occur due to cultural differences between scientists and the software engineering community. There are some suggestions of how to deal with this, including metamorphic testing. The literature also tells us that scientific software testing is not always well handled or the lack of knowledge and technique from the scientists can be a problem.

Much of the literature on scientific software development deals with the difficulties faced within that area, with much of it in agreement that these difficulties are faced due to the scientists' lack of computer science background. They also seem to agree that scientific software development is very much an individual task but some of the literature has called for some cooperation and working together.

Section 4: Methodology

As we set out in the Introduction, the main aim of this study is to survey and explore the knowledge of Research Software Engineers (RSEs) with regards to software sustainability, software quality, software testing, software engineering principles and practices, software development lifecycles and sustainability design. We intend to investigate how the research software engineering community understands and measures software sustainability. We hope to learn about the perspectives and ideas of RSEs in relation to how they define sustainability, if they apply or consider it in their own work and if so, how important they consider it in relation to their working practice. It also aims to look at software quality and how important it is considered in the development of their code.

Study Design

We identified a qualitative research approach and the decision was taken for the research questions to be answered using interviews that were semi-structured. These interviews included questions that were open and questions that were closed. These are displayed in Appendix 1.

Planning

The interview questions were developed early in the process of this study and a pilot study was conducted with an interviewee and this resulted in some minor changes to the interview structure.

This was done in order to identify the strengths in our approach as well as the weaknesses. We wanted to identify any flaws in our process before we conducted the interviews en masse. It was also designed to give us an opportunity to correct any errors or parts of the process that didn't work or were confusing for the interviewee.

The pilot interview was conducted and recorded on Microsoft Teams, a backup recording was also made on a mobile phone. The results of the pilot were generally positive but we found that four separate questions had issues with the scaling. It was observed that the scales that were set out were 1-5 and these didn't really have a midpoint, making it difficult for the interviewee to choose an answer, especially on questions that they weren't really sure on or wanted to remain neutral on. Following this, the decision was taken to change the

scaling on all questions to 1-6. This meant there was a midpoint and the interviewee was able to answer the question more easily.

Overall, the pilot interview was an important part of the study as it helped us to iron out the interview process and tweak some things to make them work more easily for the rest of the interviews. Following this, there were no other comments made regarding the scaling questions, meaning that this change was a successful one and the pilot had served its purpose.

Interviewees

The subjects were selected from the Society of Research Software Engineers using the contact details that were specified on www.society-rse.org. Every interviewee approached was an RSE Fellow during the period that the study was conducted. The study included 12 interviewees of which eight were male and the remaining four were female. All interviewees were RSE Fellows at the time of the study. Another four RSE Fellows were approached but no response was received from any of them.

Data Collection

The data collection was done through a face-to-face (via Microsoft Teams), semi-structured interview which lasted for approximately 45 minutes. The interviews covered these key areas;

1. Personal background: which included demographics, education subject and level and experience in software engineering.
2. Software engineering education and training: interviewees were asked about formal and informal training in software engineering, how this applied to their work, how much training they had in various areas of software engineering and what certifications they have.
3. Software development lifecycles: software development lifecycles and models are discussed.
4. Software testing: various types of software testing are discussed, along with the most reliable types and the biggest challenges faced in software testing.
5. Software quality: non-functional requirements are discussed, along with various software qualities and the interviewees are asked if they test the quality of their own software.

6. Software sustainability: sustainability is discussed at length, related to sustainable software, whether software sustainability is a software quality, the core software qualities of sustainable software and if the interviewees measure the sustainability of their software.
7. Karlskrona Manifesto: the Karlskrona Manifesto principles are put to the interviewees to see how much they either agree or disagree with them.

Data Analysis

Once completed, all interviews were uploaded to Saturate, a qualitative text analysis tool. The qualitative content analysis method (Mayring, 2000) was then utilised to extricate views on software engineering, software quality and software sustainability from the interview transcripts. Using Saturate, the interviews were transcribed manually and then analysed using the qualitative text analysis tool. This saw the creation of a codebook which allowed the data to be classified and to allow trends in the data to be observed.

Threats to Validity

As is quite common with qualitative research (Gibson, 2017), the results cannot be generalised to the wider population of research software engineers. This is because the study has only interviewed a small selection of RSEs and even then, the interviewees were spread out across various domains and areas, leading to very little agreement across the data.

In addition, as the researcher was initially unfamiliar with the research software engineering field, there was potential for inappropriate and unsuitable research questions being written.

Also, as the number of RSEs interviewed for this study was quite a small sample of people, there is a threat that the sample size is too small or too inadequate to be reasonable. This means that there’s a potential that the validity of the results is not guaranteed.

Another threat could be a lack of expert evaluation, the results of a study ought to be evaluated by an expert, this will allow the results to be understood and interpreted to discover the significance and the real meaning of those results. However, without such an expert being involved, incorrect conclusions may be drawn.

As only 12 people were interviewed, the generalisability of the results must be taken with that view in mind. Further studies would seek to explore whether the evidence that was uncovered in this study was more generalisable to the software engineering community.

Section 5: Results

In this section, results of our study are presented and split into the seven sections that are defined in the methodology. These sections are as follows; background, software engineering education and training, software development lifecycles, software testing, software quality, software sustainability and sustainability design.

Background

For this section, we took the results of the interviews and assigned each of the 12 interviewees with their own unique marker, and in the following table, they are marked P1 to P12. These are presented in the following table.

List of RSE's

RSE	1st degree	Highest Education level	Subject area of highest degree	Years of Experience	Official job title	Main role & focus	Years in current role	Software development domain
P1	Marine biology & oceanography	PhD	Marine macroecology	4 years	RSE	Build software for researchers.	4 years	Ecological data, software and marine.
P2	Geophysics	PhD	Geophysics	10 years	RSE	Projects around the topic area of finite element.	5 years	Engineering but also stretches between mathematics and physics.
P3	Physics	PhD	Semi-conductor physics	3 years	RSE	Software engineering support for several projects	3 years	No specific domain.
P4	Chemistry	PhD	Theoretical chemistry	25 years	RSE	Takes widely used codes in chemistry/condensed metaphysics and tries to get them to run effectively and efficiently on computers or big parallel machines.	5 years	Chemistry and condensed matter physics.
P5	Biophysics	PhD	Visualisation	25 years	RSE	Works with particular scientists to develop novel imaging methodologies.	2 years	Novel imaging methodologies.
P6	Mechanical engineering	PhD	Mechanical engineering	35 years	RSE	Research and teaching.	2 months	Composites and textile manufacturing.
P7	Physics	PhD	Theoretical physics	10 years	RSE	Biophysical and complex fluids.	3 years	Biophysical and complex fluids
P8	Computer science	PhD	Computer science	10 years	RSE	Delivering RSRC Fellowship.	5 years	No particular domain.
P9	Physics	PhD	Plasma physics	10 years	RSE	Provides software engineering support for several projects	5 years	Plasma physics
P10	Natural sciences	PhD	Condensed metaphysics	25 years	RSE	Whatever takes their interest.	3 years	Condensed metaphysics or material science
P11	Astrophysics	PhD	Astrophysics	2 years	RSE	Varies from project to project.	2 years	No particular domain
P12	Computer science	PhD	Civil engineering	10 years	RSE	Varies from project to project.	2 years.	No particular domain.

Table 1 – List of RSE's

Of the 12 interviewees, three of them graduated with their first degree in physics, another two's first degree was in Computer Science and the rest had one each in natural sciences, biophysics, chemistry, marine biology and oceanography, geophysics, astrophysics and mechanical engineering.

All interviewees are educated to a PhD level, with one in visualisation, one in computer science, one in condensed metaphysics, one in plasma physics, one in theoretical chemistry, one in semiconductor physics, one in theoretical physics, one in civil engineering, one in marine macroecology, one in geophysics, one in astrophysics and one in mechanical engineering.

All interviewees identified as a Research Software Engineer (RSE) but with regards to the role and focus they undertake in their jobs, the response of the interviewees varied.

Four interviewees' roles and focus vary and not specific to any particular domain. P5 focuses on working with particular scientists, some of whom have worked very closely with national research facilities to develop software for novel imaging methodologies. P3 provides software engineering support for several projects. P7 works in biophysical and complex fluids. P9 provides software engineering support for several projects. P1 builds software for researchers. P2 works on projects around the topic area of finite element. P8 focuses on the delivery of the RSRC Fellowship. P6 focuses on research and teaching. P4 takes a number of widely used codes in chemistry/condensed metaphysics and tries to get them to run effectively and efficiently on computers or big parallel machines.

Three interviewees have been in their roles for two years, three for three years, one for four years, four for five years and one for just two months.

The domain that the interviewees primarily develop software for is also varied. Except for four of them who develop software for no particular domain, the other nine provided different answers which were;

- Condensed metaphysics or material science
- Biology, science and engineering in processing chemical engineering and material science.
- Plasma physics.
- Chemistry and condensed metaphysics.
- Biophysical and complex fluids.
- Ecological data, software and marine.
- Engineering, but also stretches between mathematics and physics.
- Composites and textile manufacturing.

All interviewees have varying experience in software engineering; six have 10 years' experience, three of them have 25 years' experience and then one each has two years', three years' and 35 years' experience.

Software Engineering Education and Training

Of the 12 interviewees, four have had some formal training in software engineering and development, and eight don't have any formal training, although one acknowledged that they've attended unspecified courses. All eight, who haven't had any formal training, have had some informal training and/or have learned on the job. All interviewees acknowledge that the training, whether formal or informal has helped them in their current roles.

The interviewees were asked how much training in their formal education or professional development there has been in the following areas; software requirements, software design, software architecture, software coding, software testing, software maintenance, software configuration management, software engineering management, software engineering process, software engineering models and methods, software quality, software engineering professional practice and software engineering economics.

The results of these questions are presented in a series of bar graphs.

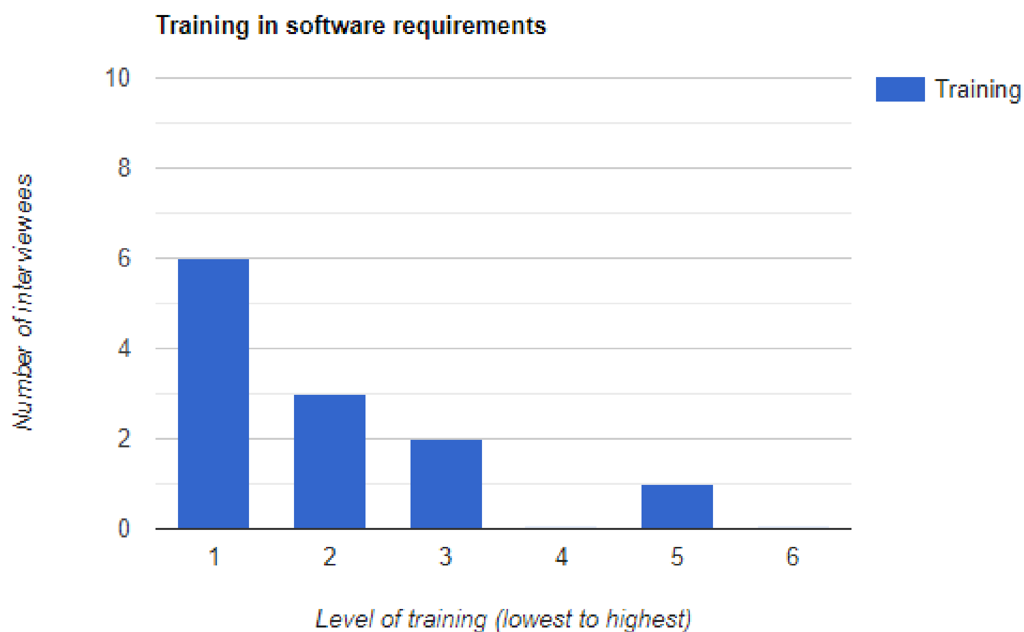

Figure 1 – Training in software requirements

The results in Figure 1 show that training in software requirements is quite low amongst the interviewees, with the majority having no training at all, with only one having some kind of training in this area.

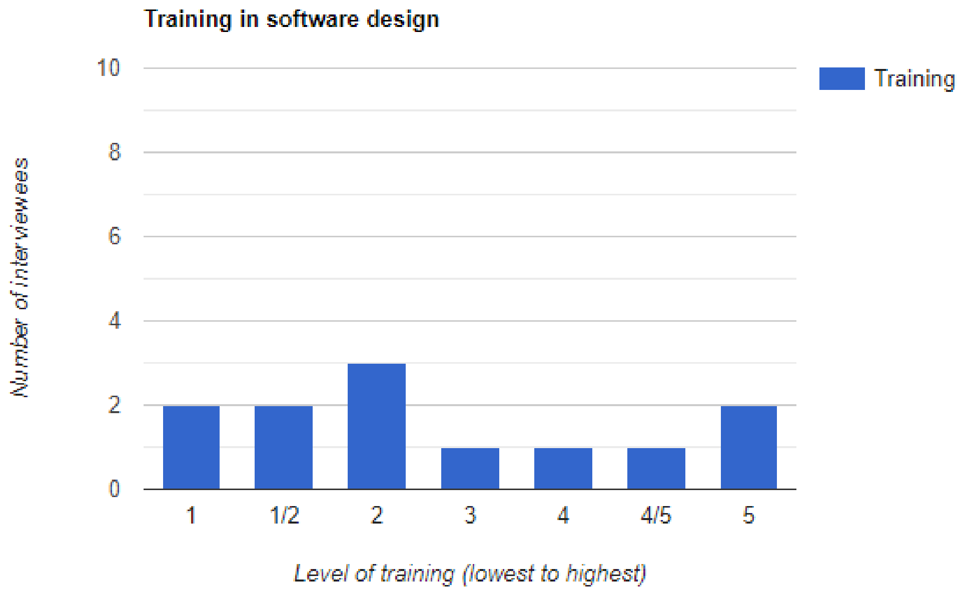

Figure 2 – Training in software design

Figure 2 tells a vague story, with absolutely no consensus on training in software design. It also led to some unsure answers with P8 and P9 citing 1 or 2 and P10 opting for 4 or 5.

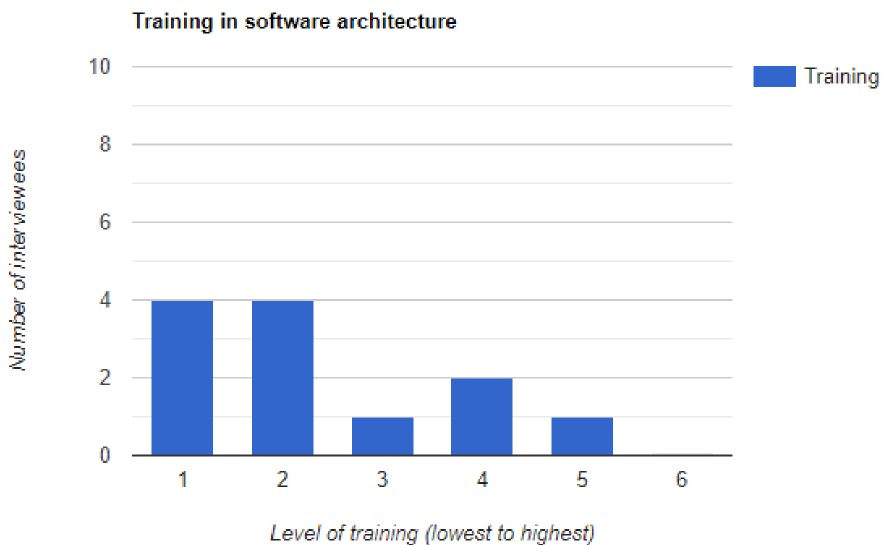

Figure 3 – Training in software architecture

Figure 3 shows that the vast majority of interviewees don't have any training in software architecture. Despite this, one or two interviewees acknowledged that they had received some training.

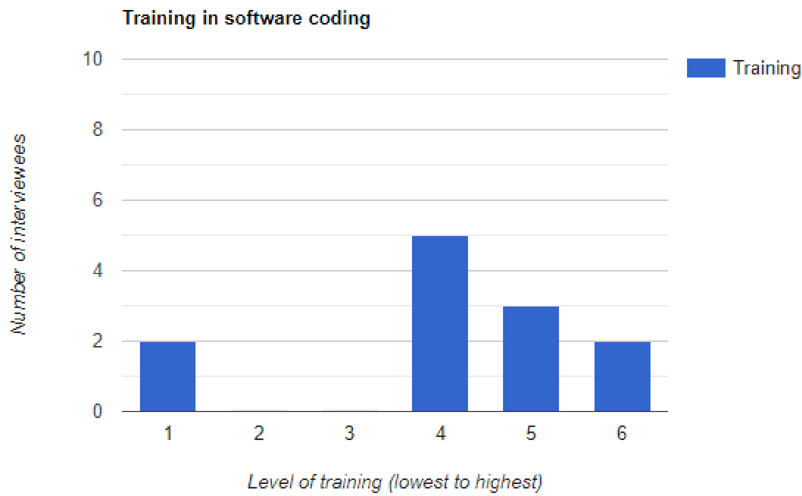

Figure 4 – Training in software coding

Figure 4 shows that the consensus among the interviewees is that they've received some form of software coding, with just two interviewees having received no training whatsoever in the area.

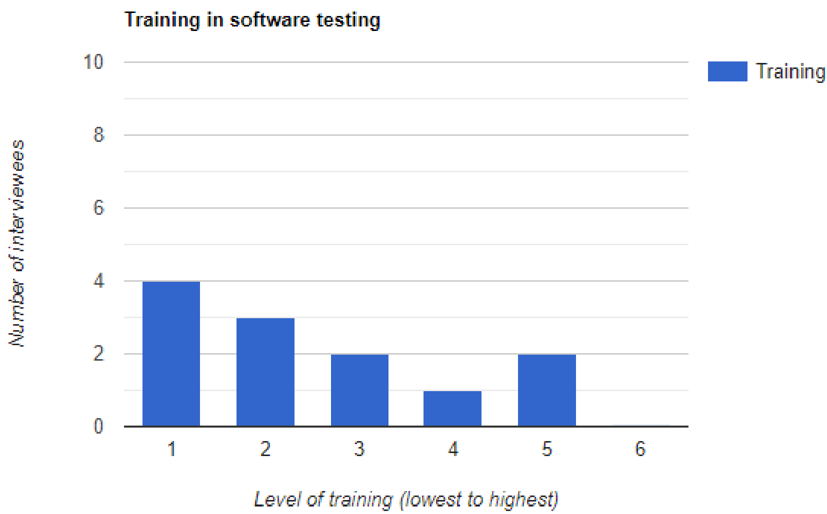

Figure 5 – Training in software testing

Figure 5 was consistently inconsistent with no consensus agreed amongst the interviewees regarding software testing training in their professional development and formal education.

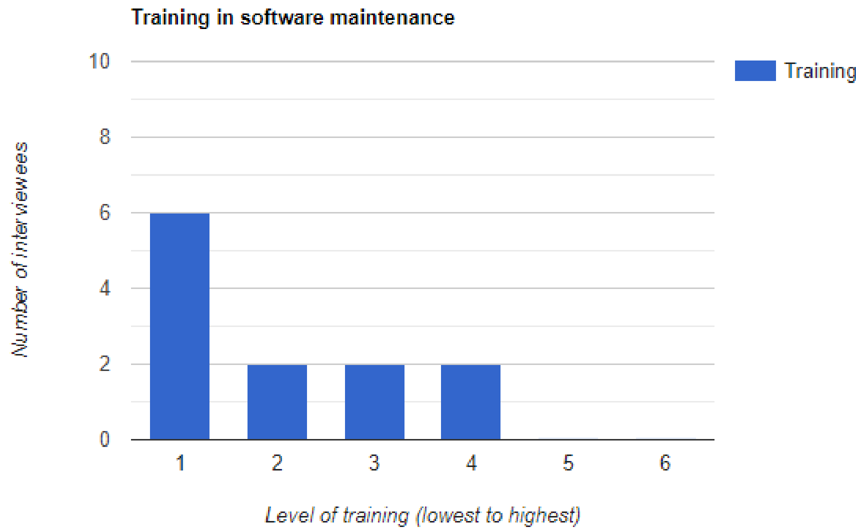

Figure 6 – Training in software maintenance

Figure 6 demonstrates that the interviewees haven't had much training in software maintenance in their professional development and formal education, with the majority acknowledging that they had very little training in this area.

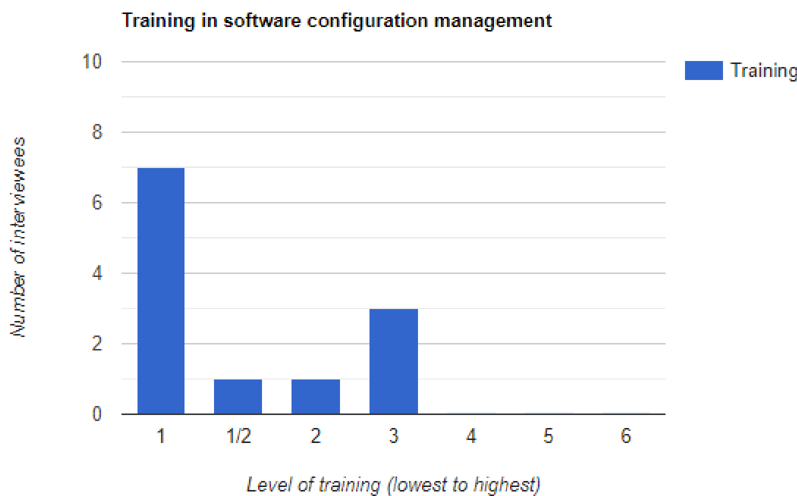

Figure 7 – Training in software configuration management

There was a near-consensus in Figure 7, with the vast majority stating that they had no training in software configuration. There was some indecision, with P5 unable to give a definitive answer, citing 1 or 2. A small minority of the interviewees swayed towards some training rather than one but only three of them said this.

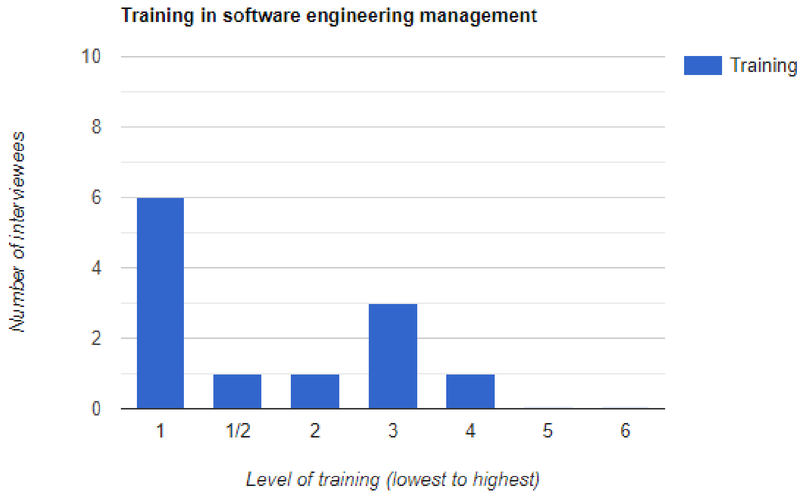

Figure 8 – Training in software engineering management

Figure 8 shows that although the majority of interviewees have no training in software engineering management, a small minority have had some kind of training. Although it doesn't add much to the discussion, P5 was undecided and cited 1 or 2.

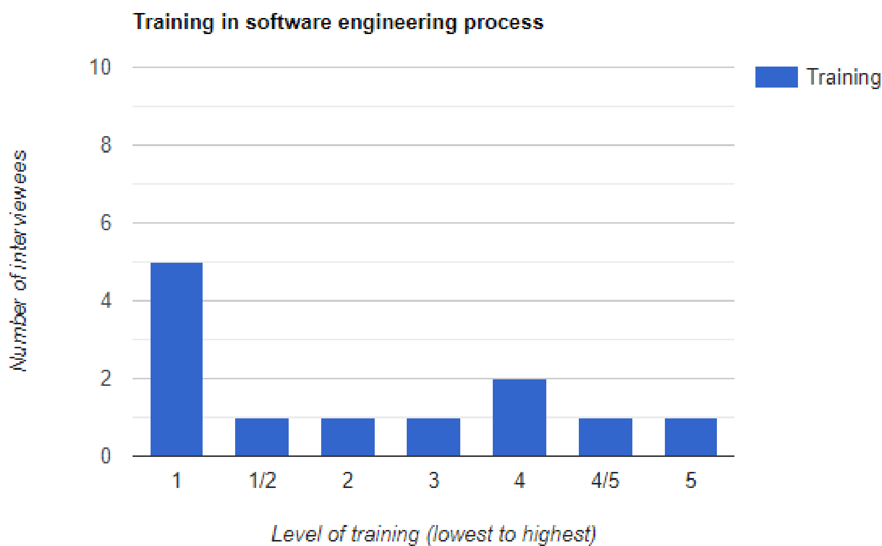

Figure 9 – Training in software engineering process

Figure 9 shows that there wasn't much training in software engineering process amongst the interviewees with the majority stating that they had no training in it, along with P5 who although undecided, cited 1 or 2. A small minority suggested they had some training in it, and further to this P9 was undecided but cited 4 to 5.

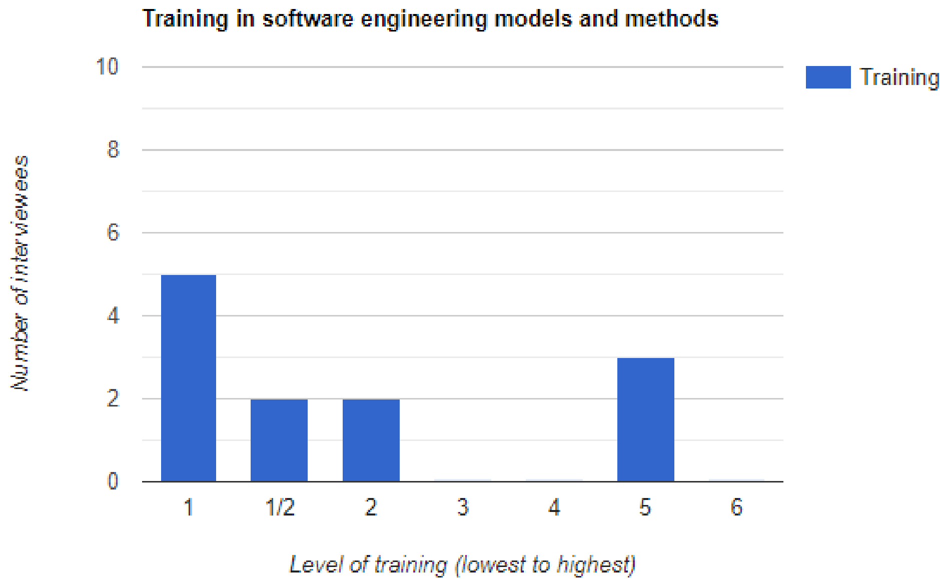

Figure 10 – Training in software engineering models and methods

Figure 10 is quite strange in that it shows that the vast majority of interviewees have had little or no training in software engineering models and methods and then on the flip side, three have had quite a bit of training. So although, there's a majority that hasn't, there's a small minority that has and this means there's no clear consensus. Also P3 and P5 were undecided but stated 1 or 2 on the scale as their answers.

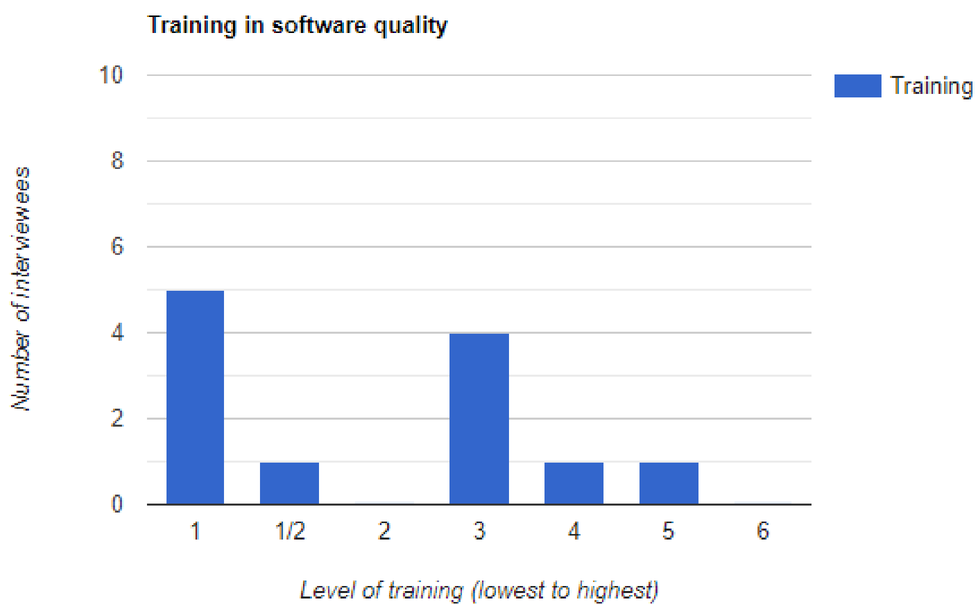

Figure 11 – Training in software quality

Figure 11 shows that there's no real consensus. Although almost half of the interviewees have had no training in software quality, there's a small minority that have. Further to this, P5 was undecided and cited 1 or 2.

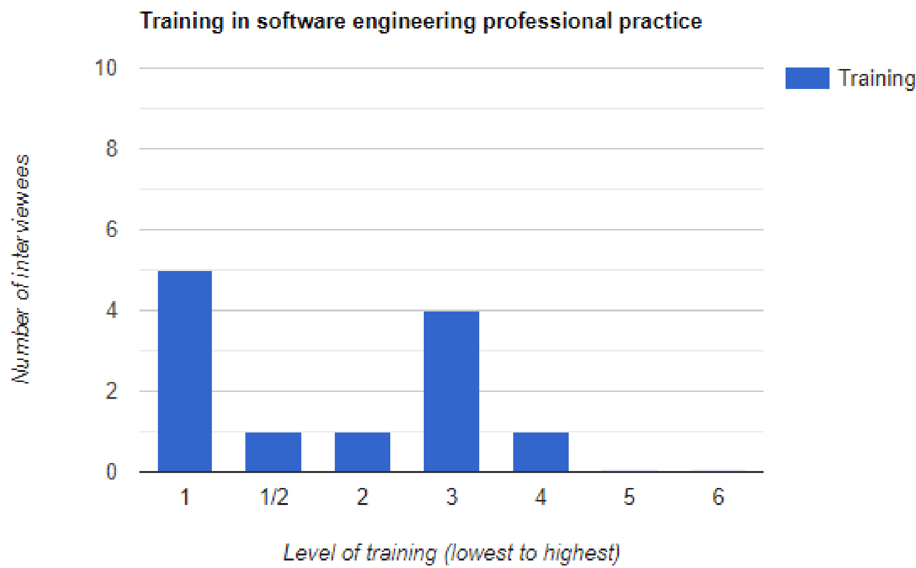

Figure 12 – Training in software engineering professional practice

Figure 12 shows that there isn't much training in software engineering professional practice amongst the interviewees and the majority seems to stray towards no training at all, with a small minority acknowledging that they've had some training in it. Also, P5 was undecided and cited 1 or 2 on the scale.

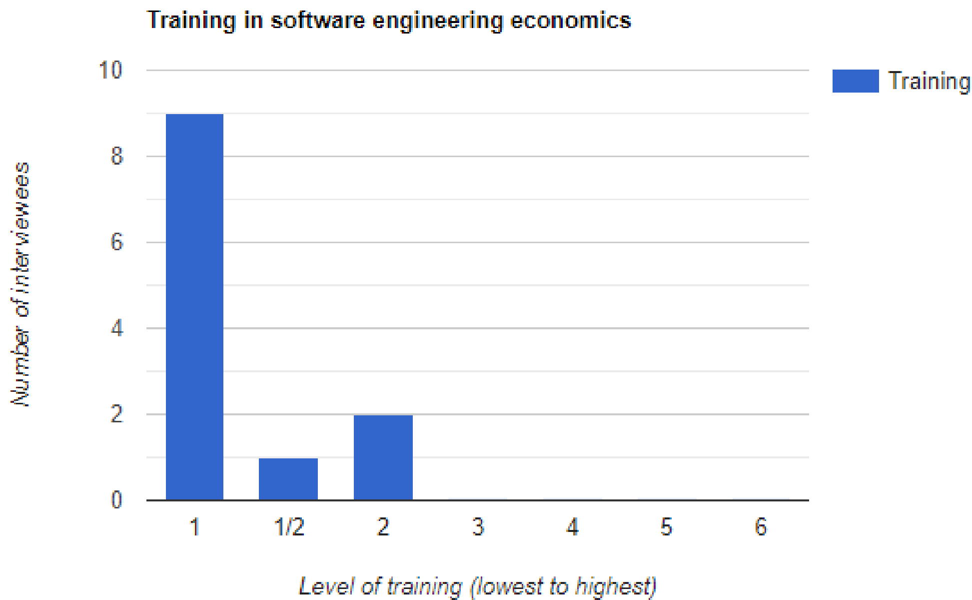

Figure 13 – Training in software engineering economics

Figure 13 demonstrates conclusively that the interviewees haven't received much training in software engineering economics. There was some uncertainty, with P5 citing 1 or 2.

10 of the interviewees have attended a Software Carpentry course and two haven't. Six of them attended as an instructor or helper. P1 taught the R data carpentry and software carpentry course. P5 has helped on courses for Unix skills and computational linguistics. P8 was vague in their answer, simply stating that they'd undertaken the training for software carpentry. P10 has hosted a carpentry approved hybrid of four carpentry lessons. P12 has attended some courses and is also a software carpentry instructor but did not elaborate. P3 was an instructor on an Imperial software carpentry course. P6 attended a Python based software carpentry course. P9 had attended one but didn't know the name, saying; "*It had Python, Makefiles, SQL and Git. I'm not sure if it had a particular name beyond software carpentry"*. P7 was vague in their answer; "*I've done programming courses and palletization and all of these things. Software carpentry, partly*". P2 admitted they'd been to *"one or two"* but when asked, could only say "*it was a course on testing*".

All but one of the interviewees have no certifications in software engineering and one acknowledged that they have, stating they're certified in software carpentry.

Software Development Lifecycles

When asked what software development lifecycle model they adopt on a typical project, the response was varied. P1 and P5 use the Iterative model, P6, P9 and P10 use the Agile method whereas P2 recognises, due to their lack of formal training, they have devised their own version which is similar to Agile, but also "beyond" Agile. P3 uses a method that is "between" Waterfall and Agile but admits it "*doesn't really fit into any of the traditional ones*". P4 doesn't use anything specifically, it is based on what project they are working on.

P7 doesn't use any type of model but are in the process of formalising the process as a team. P8 doesn't acknowledge a specific model/method, simply that they do releases and then do any feature development and bug fixes up until the next release. P11 doesn't use a model as they've been working on a project for a substantial amount of time and that project was already in existence long before that. P12 doesn't adopt a specific model either and had trouble fitting their answer into the question.

Nine of the interviewees feel the model/approach that they adopt is best suited to the projects they undertake whereas one doesn't. One interviewee wasn't sure and one interviewee answered "not applicable".

P1 conceded that they don't always stick to their preferred model whereas P3 feels that it would be beneficial to have a more formal requirements gathering process. Although P8 believes their model works, they also went on to say it's failing because the projects they do are lost research related and there's a lot of changes going on and it can cause tension. P11 wasn't sure they even adopted a specific model but did state that they had a "sort of lifecycle for the sub bits of the project".

The interviewees were asked the approximate percentage of time that they would spend on the following activities on a typical project; requirements elicitation, documenting requirements, software architecture, software design, coding, software testing, usability testing and project management. To demonstrate the wide variety of responses from the interviewees on each question, a series of line graphs have been created to show the results. As each interviewee works on different projects, the varying answers found in these graphs are to be expected.

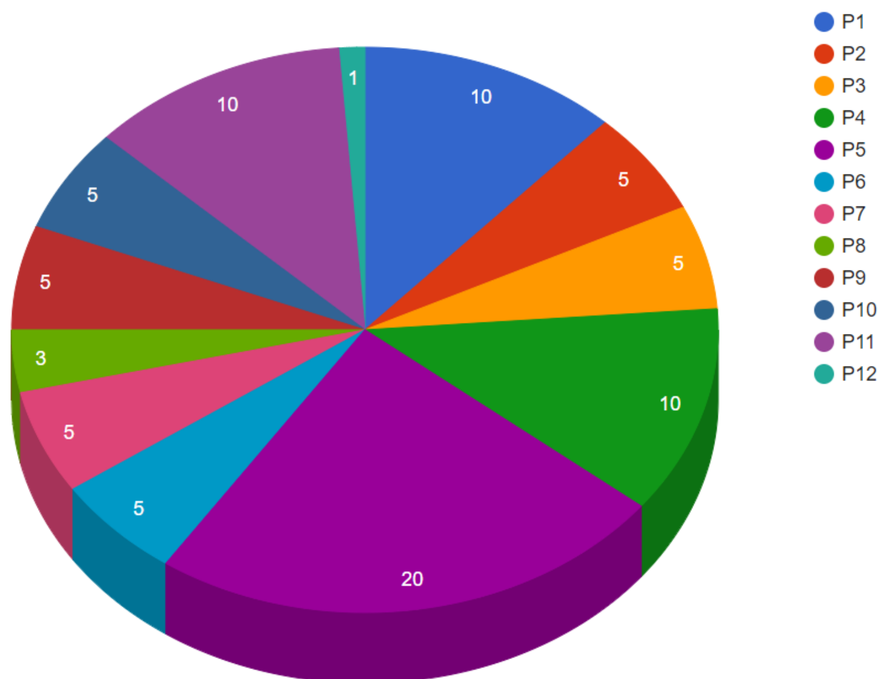

Figure 14 – Time spent on requirements elicitation

Figure 14 shows that although one interviewee spends approximately 20% of their time on requirements elicitation, the majority spend very little time on it during a typical project. Despite this, all interviewees spend at least some amount of time on it.

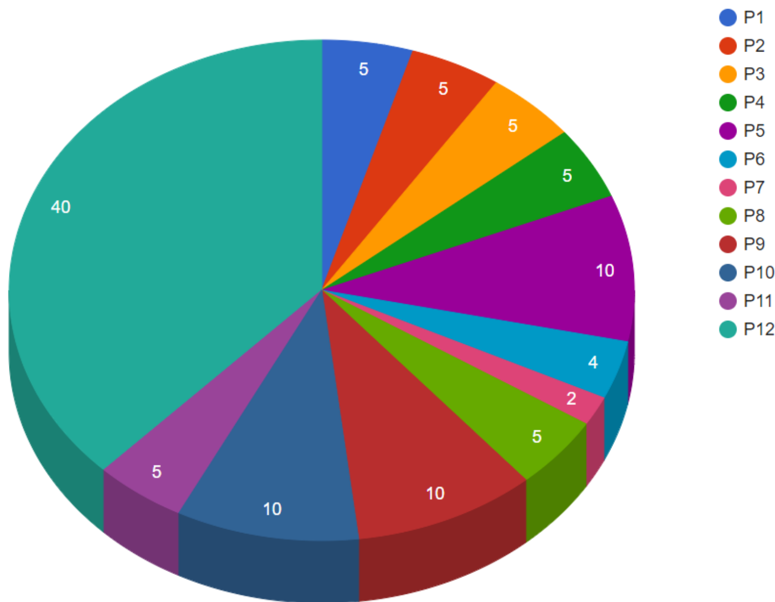

Figure 15 – Time spent on documenting requirements

Figure 15 demonstrates that although one of the interviewees spends approximately 40% of their time on it, the vast majority of the responses show that very little time is spent on it. Despite this, all interviewees acknowledge that they do spend at least some time on it.

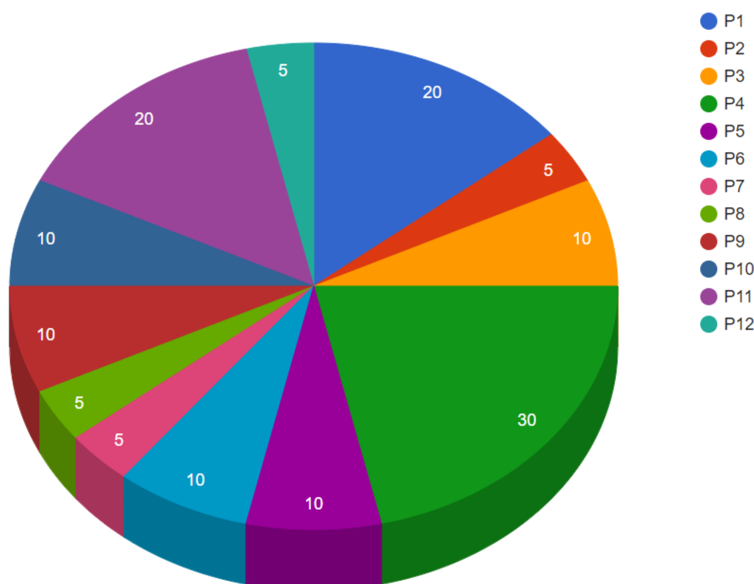

Figure 16 – Time spent on software architecture

Figure 16 shows no clear consensus for the approximate time spent by the interviewees on software architecture. Just one of the interviewees stated that they spend approximately 30% on it, another two stated 20% but the rest spend either 10% or less on it. Once again, despite the varying responses, all the interviewees spend some time concentrating on this area.

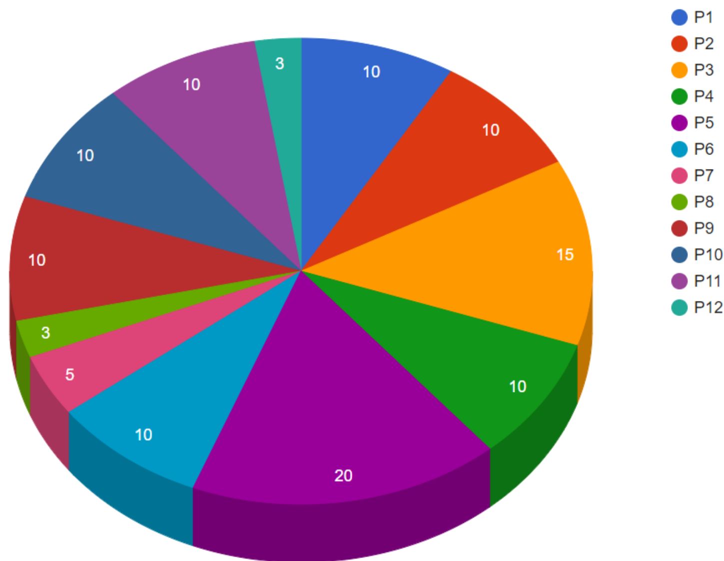

Figure 17 – Time spent on software design

Figure 17 confirms that more than half of RSEs spend approximately 10% of their time on software design with two more acknowledging they spend an even higher amount of time, but not exceeding 20%. Only three of the interviewees spend less than 10% on *software design* but all of them do spend some time on it for their projects.

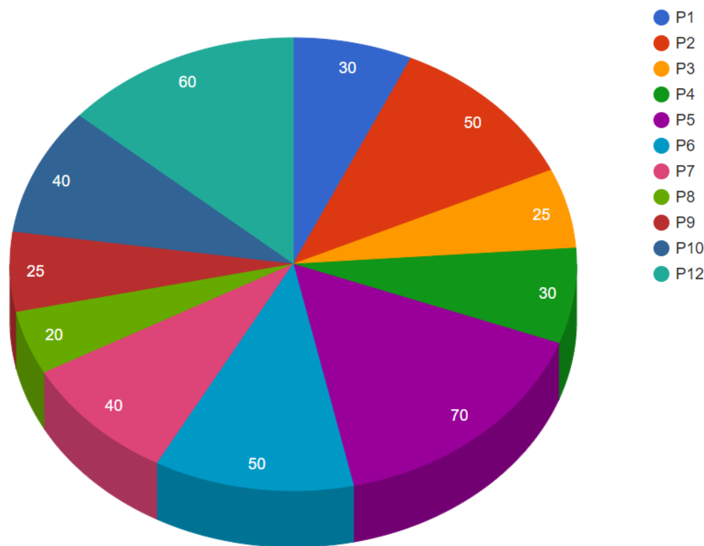

Figure 18 – Time spent on coding

Figure 18 shows the range of different projects that the interviewees work on, with no clear consensus on the amount of time that is spent on coding, with only two interviewees giving the same answer.

**P11 stated that they couldn't really answer this question as "I'm finding it hard to give an actual percentage on it, it's not like we have design phases, we just do the project. There's nothing typical at our institute". Because of this, they are not included on the chart.*

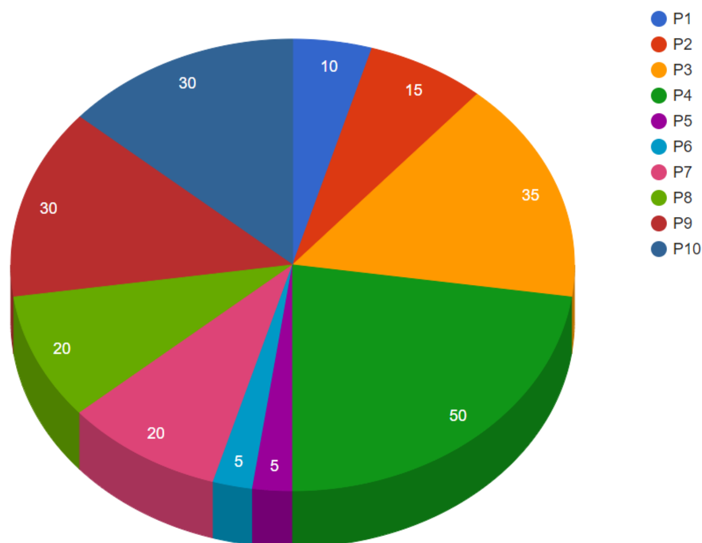

Figure 19 – Time spent on software testing

Figure 19 shows that there is no clear consensus on the time spent on software testing on a typical project, with the interviewees providing varied answers. There was some agreement, with two interviewees each answering 5%, 20% and 30%.

**P11 stated that they couldn't really answer this question as "I'm finding it hard to give an actual percentage on it, it's not like we have design phases, we just do the project. There's nothing typical at our institute". Because of this, they are not included on the chart.*

**P12 said; "This is difficult to answer because we have projects where we're writing software for people and other projects where we're more working on the infrastructure around the main development effort. So on some cases we'll be doing, say 60% of software development effort, maybe sort of 10-20% on testing. Other times, it might be actually 40% of the effort is spent on testing, 40% on documentation. It varies". Because of this, they are not included on the chart.*

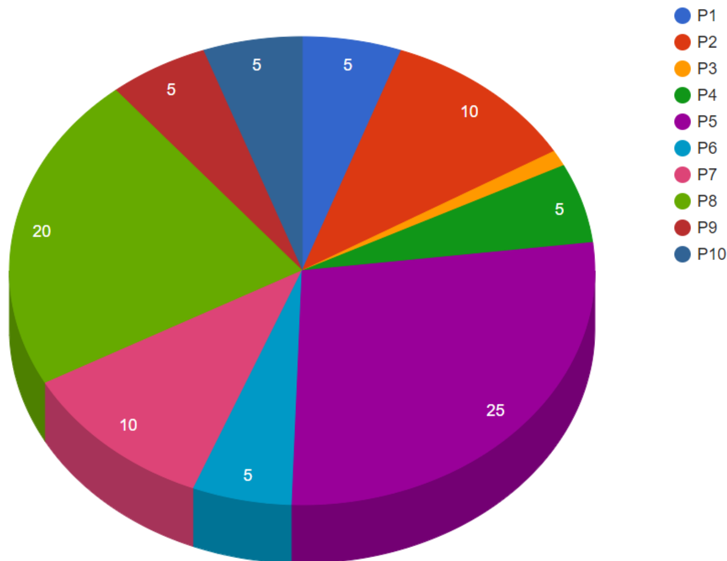

Figure 20 – Time spent on usability testing

Figure 20 doesn't quite provide a clear consensus but when asked how much time they approximately spend on usability testing, there was some agreement found in the data, with five of the 12 interviewees giving an answer of 5% and two more giving an answer of 10%.

**P12 said "This is difficult to answer because we have projects where we're writing software for people and other projects where we're more working on the infrastructure around the main development effort. So on some cases we'll be doing, say 60% of software development effort, maybe sort of 10-20% on testing. Other times, it might be actually 40% of the effort is spent on testing, 40% on documentation. It varies". Because of this, they are not included on the chart.*

**P11 stated that they couldn't really answer this question as "I'm finding it hard to give an actual percentage on it, it's not like we have design phases, we just do the project. There's nothing typical at our institute". Because of this, they are not included on the chart.*

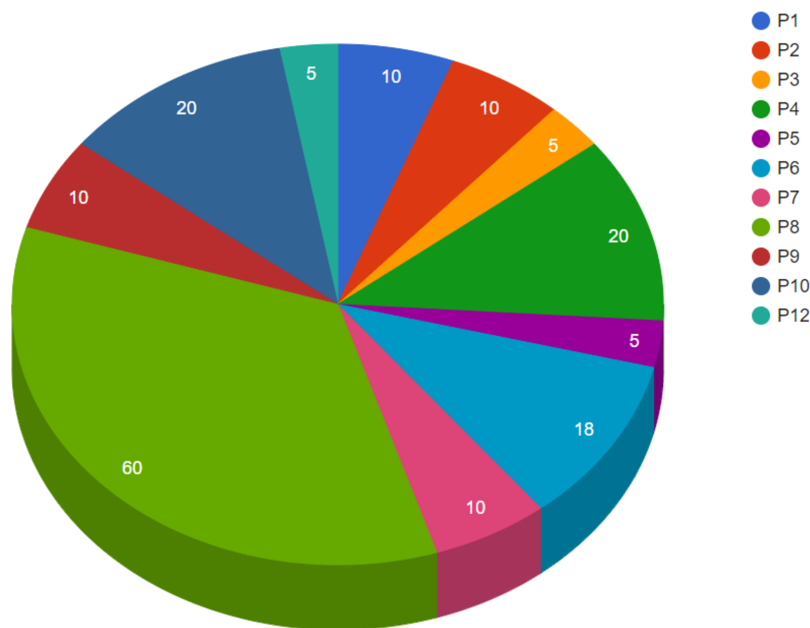

Figure 21 – Time spent on project management

Figure 21 shows no clear consensus of how much time the interviewees spend on project management, though there is some agreement, with four interviewees answering 10% and another three answering 5% and then two more answering 20%.

**P11 stated that they couldn't really answer this question as "I'm finding it hard to give an actual percentage on it, it's not like we have design phases, we just do the project. There's nothing typical at our institute". Because of this, they are not included on the chart.*

Software Testing

All 12 interviewees carry out software testing in their roles but the responses varied to what extent they plan for this. Eight of the interviewees do plan and the other four gave varied answers alluding to the suggestion that they don't really plan at all.

Of the four interviewees, P6 doesn't really plan; *"I don't really plan for it. It's just one of those things that if I'm developing a new feature for the software, then I would. Or developing some new code, then I would write the tests to go with it. It's just one of those things that I just do when needed".* P9 struggled to give a straight answer; *"The two main codes I maintain are legacy codes that I came into the position with an already fully formed code so they don't have much chance to plan for testing. I've tried to retrofit tests on that. When I write new code, the planning and testing goes hand in hand with the software design and I try to do both at the*

same time. So it's hard to give a straight answer for that". P11 said; *"I don't plan because the software itself isn't planned. A lot of it is basic sanity checking, tests, like making sure you're getting the right answer and output. If I'm doing software testing, it's being developed alongside the code and if I've not planned the code, I can't really plan the tests*". P1 said; *"Not too much. If I'm building a feature, I'll be writing a test that this feature is working. Then as I move along and find little bugs, and I'll just be adding tests to capture as bugs. I guess it's going hand in hand with architecture design. And then obviously bugs as they pop up"*.

P4 felt it was quite important saying; *"Part of the project design is working out how I'm going to test. I think there's absolutely no point in writing software unless it gives the correct answers. My job is to make code run fast. The constraints of getting the right answers is very important indeed so I do spend an awful lot of time working out how I'm actually going to check the answers are right"*. P8 simply said; *"I tend to (plan) wherever possible, write the tests first so with C++ and Python that's relatively easy. So we plan and kind of factor that in"*.

When asked what type of software testing they implement, the responses were quite similar. The most popular answer was unit testing, with nine of the 12 interviewees confirming that they implement it. Another six of them stated that they implement integration testing. Two interviewees also stated that they implemented input testing.

While acknowledging their use of unit testing, P6 went further; *"Basically, we've got unit tests. What I use is programmed in C++ and I use CPP unit. I've got tests that are written to run those so because of the nature of the program, it's not necessarily strictly unit tests as in down to the individual functions. Feature tests, if you like"*. P1 acknowledged that they implemented snippets of different types of testing; *"We've got a bit of everything, really. I unit test individual functions. I have snippets of input that are testing for a specific output but then I will test a longer part of the software as well".* P4 said; *"While I don't use formal unit testing framework for the vast majority of the software teams, I develop programs and will think of a way of poking some numbers in there and checking the right numbers come out".* P5 had previously worked in visualisation and applied this thinking to the question; "*Because I've been in visualisation, it's whether the results look right. If you're visualising a human lung and it doesn't look like a human lung, then you've got something wrong! You need to check that it matches with reality"*. P7 implements a variety of different techniques, performance tests, build tests, regression tests. P9 also implements a variety, regression, system tests and golden answer tests. P11 mainly uses integration testing but expanded on this; "*I mostly do integration type testing, where I've written a function and I'm expecting a certain answer back. I'll feed in fake data and*

make sure the answer I'm getting back is the same, so every time I run those tests, if that test breaks, I know it's because the function is now doing something different to what I think it was. I've also done a little bit of mocking, so there's lots of faking data".

When asked about their experiences with coverage-based, fault-based and error-based testing respectively, 10 interviewees stated that they had experience with coverage-based testing and two interviewees did not. Two interviewees stated that they had experience of fault-based testing whereas 10 did not. Four interviewees stated that they had experience with error-based testing and eight did not.

When asked which was the most reliable type of testing, the answers also varied. P8 felt that unit testing was the most reliable; *"because of the coverage that you get and the ability to isolate independent functional units"*. P6 also listed unit testing, while acknowledging error-based testing. P9 were also in agreement regarding unit testing but also listed integration testing. P4 was another that was in agreement with unit testing. P5 felt that the deliberate use of bad data was the most reliable type of testing. P2 simply stated; *"You've just got to try and think of as many different tests as possible"*. P1 listed *"lots of small tests"* whereas P11 didn't give any specifics; *"I think you probably need a little bit of all of them and it depends on the projects"*. P10, while acknowledging *"they're all good for different things"*, listed correctness tests and regression tests. P7 listed coverage-based testing as the most reliable. P6 stated that error-based testing was the most reliable along with unit testing.

P3 disagreed with the question; *"I think all of them have their importance for different things. Unit testing is definitely important, but very often you find that you put together functions that are perfectly fine in isolation but don't work well together. It's not the question of being reliable or not, they serve different purposes"*.

When asked what the challenges around software testing were, the interviewees gave varied responses. P1 found that not knowing where to start was a challenge and put this down to their lack of formal training. P12 also referenced lack of knowhow being a challenge. P3 felt that the challenging part of testing is "*to have a code base that is modular enough to allow for thorough testing of the individual units. Testing these things is very complicated and when we do write the code, we need to avoid these situations. When we need code that's someone else's or we are asked to help a refactoring of someone else's code, that's a situation we find very often"*. P2 found that coverage was a challenge as well as writing tests. They also mentioned the concept of tension; *"You might not think of all the possible failures mode*

yourself. It's inevitable and that's where the problem comes with this tension between wanting to make sure it's a fast one. A unit test is not going to take you five minutes to run, it's going to be tested in a fraction of a second preferably". P9 also spoke about writing tests being a challenge along with large amounts of tests, complex code and substantial code bases. P11 was also in agreement with regards to writing tests being a challenge. P6 felt that the biggest challenge wasn't actually in the testing itself but it was getting people to write any tests. They said; *"If you've got people who aren't necessarily RSEs, writing for the software then getting people to actually write tests and to do it themselves, that's the biggest challenge"*. P4 says that their biggest challenge is the fear of missing something when their tests are implemented. P5 says that time is the biggest challenge as; *"writing unit tests for a user interface is very time consuming and building the user interface is very time consuming"*. P11 also mentioned time being a particular challenge of software testing. P2 and P10 both stated that a challenge was not knowing what the correct answer was. P10 continued and stated that getting people on board was a challenge they experienced. P12 was also in agreement; *"testing software, when the architecture and the right answers to some sort of given function call aren't even known, because it's very much research in progress".* P7 felt that getting a test suite up and running was a challenge and also different functionality; *"If you rely too much on existing functionality being tested in preceding scenarios, and you move on to new scenarios, then the tests can still pass, but you still have a problem somewhere. So over-reliance".*

P8 also had differing views; *"The testing of complex systems models where you've got emergent behaviour. So you don't have an output that you can say "this is the correct answer". The output is emergent and having an acceptance test for that means that you've got to derive some statistical properties. And understand what an acceptable margin of error for those statistical values is"*.

Software Quality

11 interviewees consider software quality in the development of their software. All of whom have various different thoughts on the subject matter.

P3 considers it very important and says "*the whole point of testing is actually about software quality*". They consider a number of qualities; *"Modularity is one of them, the idea that it's functional, it's a block of code to do one thing and one thing only. That's a good qualitative metric of good quality code. Documentation is another one, so if there's something that does the job, and does it well, it's very likely that you will need something that if it's not there, the option that is available, will have already been tested. So the chances of you writing the same*

thing and doing it wrong are higher, producing things that are already there. So that's the sort of thing that we try to look into, the modularity of the code, criteria of the code".

P7 considers; *"Quality, predominantly in the sense of robustness and reliability but also repeatability and extendibility"* whereas P8 acknowledges; *"We always try to write good quality software. We try to follow particular coding standards, tools like Linting"*. P6 also admits that they consider it but concedes *"It's not formalised but if researchers do pull requests, I might throw it back to them and say "put this or that right" and give it some meaningful variable names"*. P5 says; *"For me, there's three things that I want to have for a piece of software and first of all, the person who's going to use it can install it and get it working easily. Secondly, I want them to get the right answers out of it. And thirdly, I want them to find it easy to use. You can have great software, but if no one can install it, it doesn't really matter. You also have to put documentation in for all of those*". P9 also considers documentation but goes further; *"My big thing is readability, obviously it's got to be correct. Have a few tests and check that but really want the code to be readable. So that includes documentation but also variable names, formatting, things like that"*. P4 is the only interviewee to answer "no" and their reasons for this were; "*I don't have a metric for software quality so I think I have to answer no. There are certain things I do strongly consider. I work in compiled languages and I will turn every warning I can find on the compiler and I will fix every single warning that I get from there so it just compiles totally cleanly. But in terms of an actual metric, no I don't have one*". Their reason for not considering software quality was *"I've got two, three, four decades and it's not something I've learned along the way and once you're stuck in a rut, it's difficult to actually change your practices sometimes and that's one practice that hasn't changed over the years"*.

Out of the 12 interviewees, four were aware of ISO:9126 and eight were not. Also, three were aware of ISO: 25010:2011 and nine were not.

The interviewees were asked how important they rated the following software qualities in the development of their code; functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. To demonstrate the wide variety of responses from the interviewees on each question, a series of bar graphs have been created to show the results.

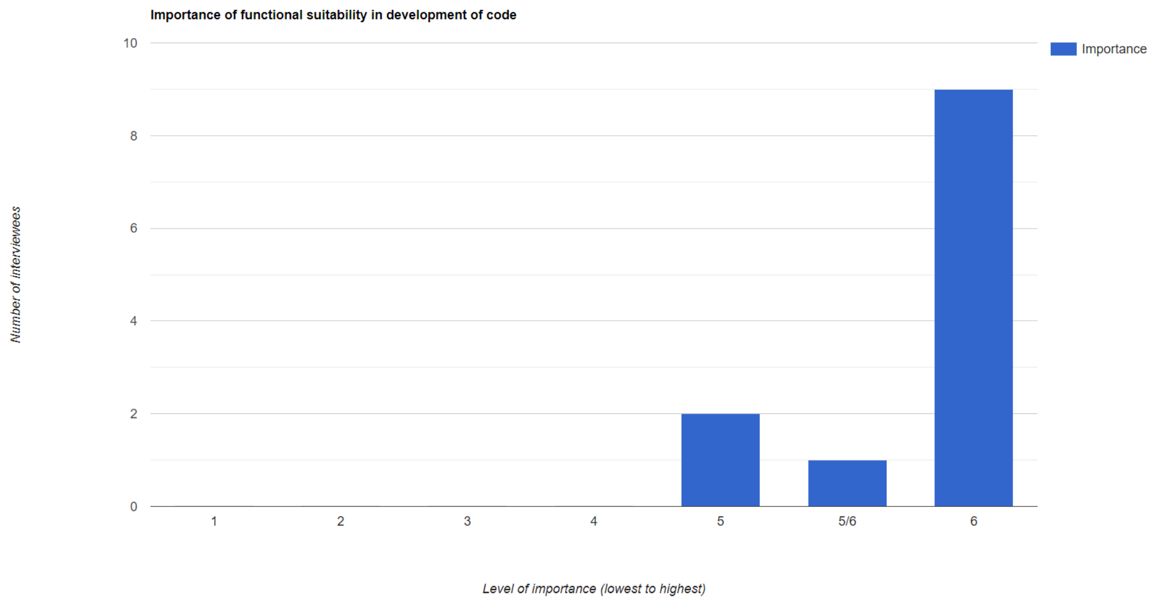

Figure 22 – Importance of functional suitability in development of code

Figure 22 shows that the interviewees rate functional suitability very highly, with the majority of them giving the highest answer. There was some uncertainty from P5, but they cited 5 or 6.

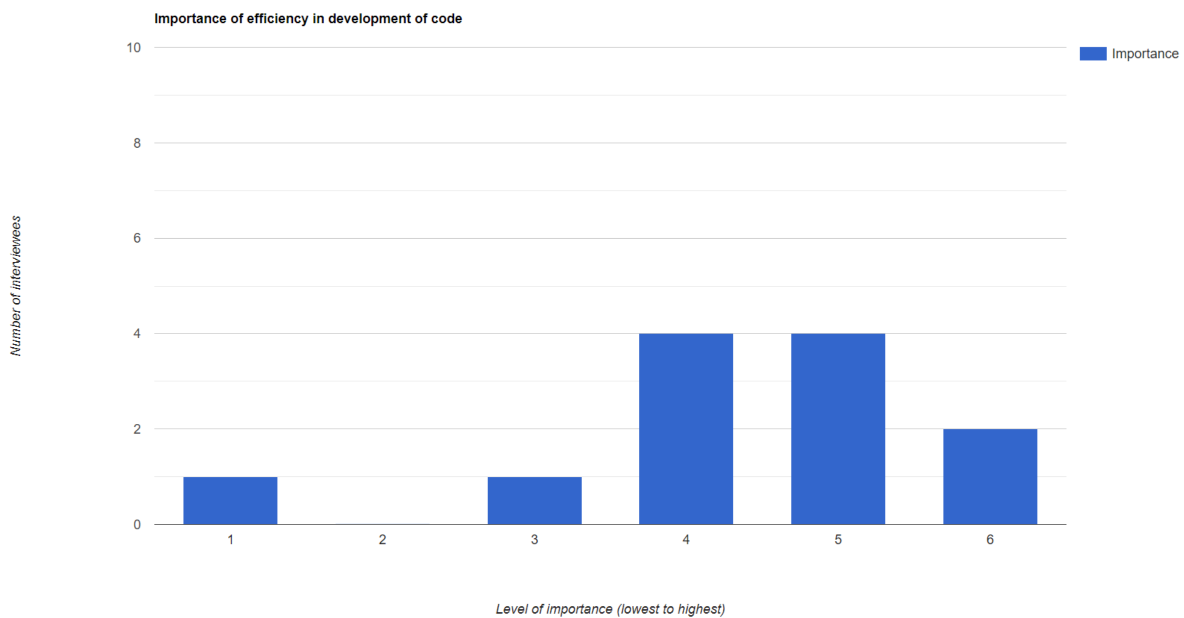

Figure 23 – Importance of efficiency in development of code

Figure 23 demonstrates that although there is no clear consensus among the interviewees of how important they rate performance efficiency in the development of their code, they are straying towards it being quite important.

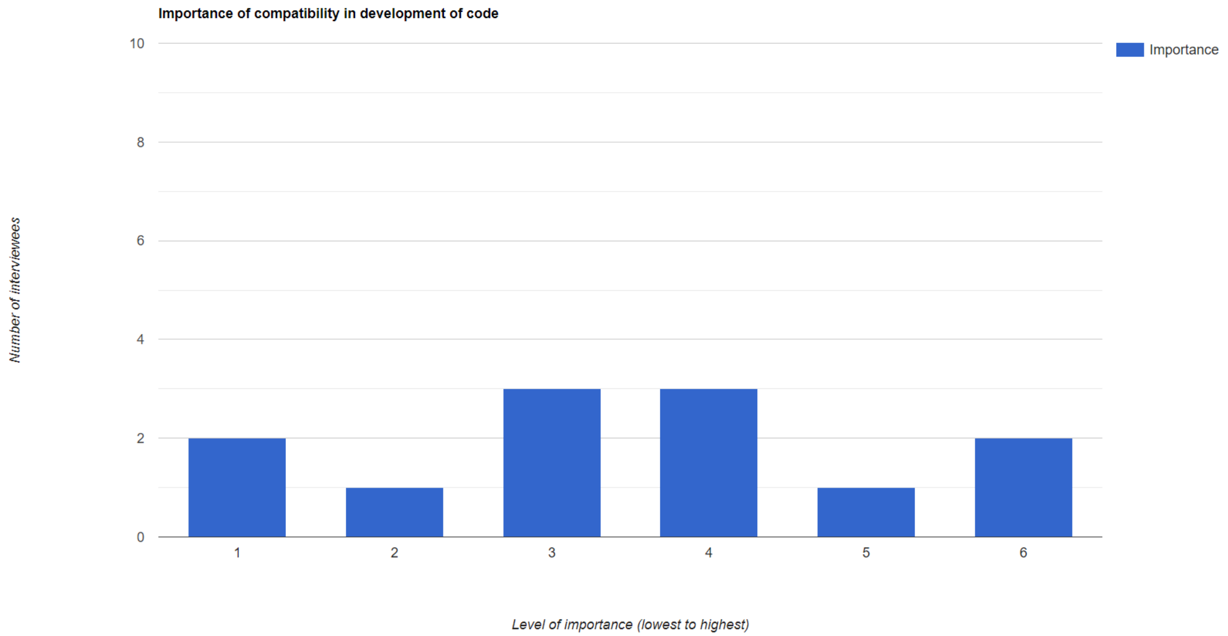

Figure 24 – Importance of compatibility in development of code

Figure 24 shows that opinion is split right across the board, with no clear consensus on how important they rate compatibility in the development of their code.

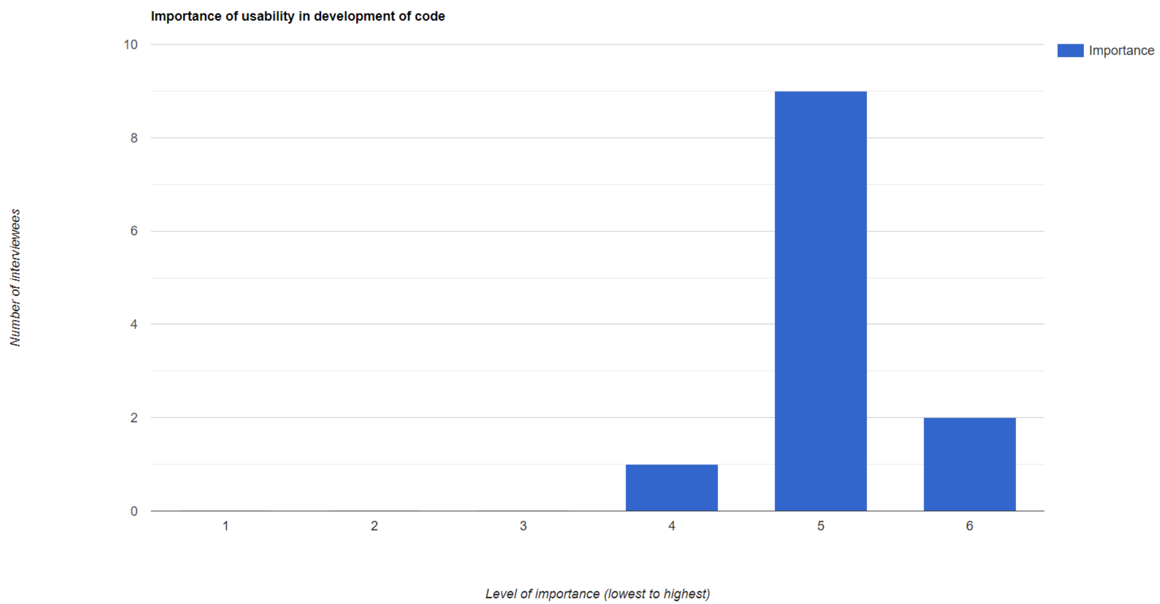

Figure 25 – Importance of usability in development of code

The overwhelming majority stated in Figure 25, that they rate usability very highly in the development of their code, with nine opting for 5 on the scale and another two opting for 6. There is a clear consensus here.

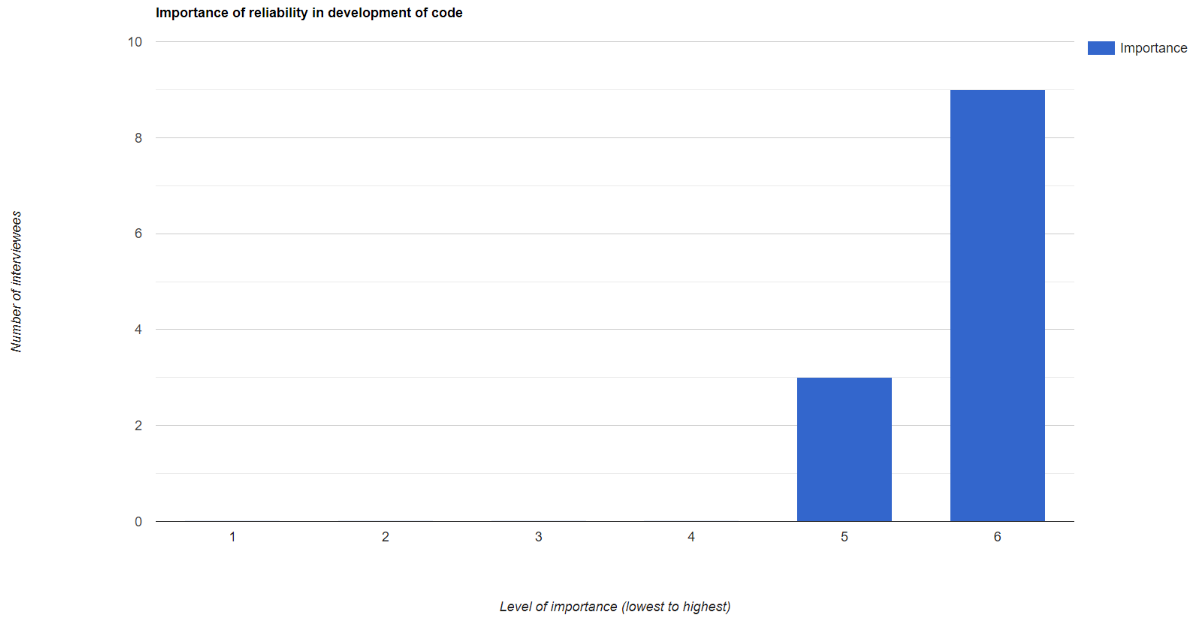

Figure 26 – Importance of reliability in development of code

Figure 26 proves that there is a clear consensus amongst the interviewees that reliability is very important in the development of their code, with the majority opting for 6 on the scale, and the remaining three opting for 5.

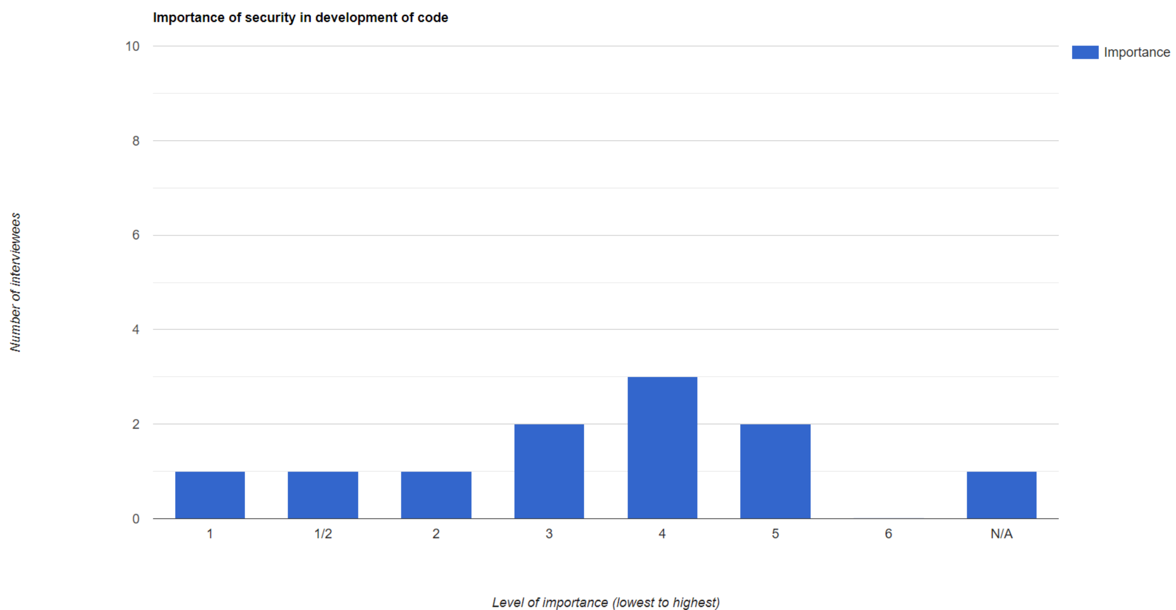

Figure 27 – Importance of security of development of code

Figure 27 shows that there is no agreement amongst the interviewees when asked how important they rate security in the development of their code. With answers all across the board, including some who stated it wasn't applicable for their role and P8 who was undecided and stated 1 or 2, it shows that there's no consensus among the data here.

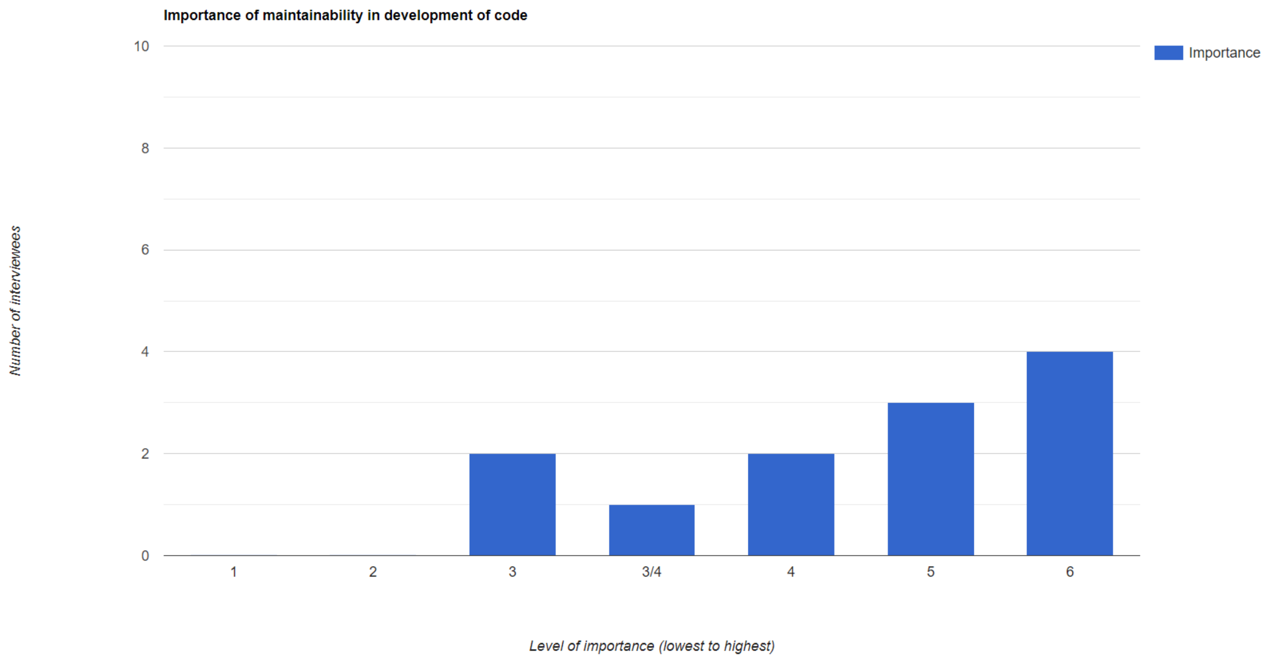

Figure 28 – Importance of maintainability in development of code

For Figure 28, although there is no clear consensus, interviewees seem to be straying towards rating maintainability in the development of their code as quite important but the opinion is split between 3 and 6 on the scale, with nobody opting for any lower than 3. Further to this, P4 was undecided and cited 3 or 4.

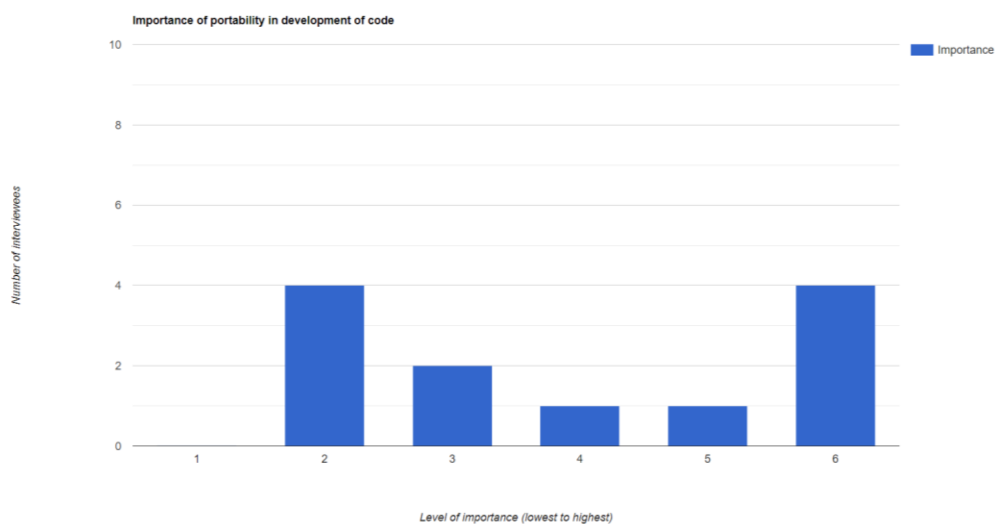

Figure 29 – Importance of portability in development of code

Figure 29 demonstrates the lack of consensus for this question, as the same amount of RSEs don't consider portability important as those that consider it very important. Opinions differ greatly here.

When asked if they test the quality of their software, the responses were varied. Four interviewees (P9, P10, P5 and P3) do test the quality of their software, whereas two say they do but not formally (P11, P12). Another two interviewees (P8 and P2) acknowledged that they do bits. Three interviewees (P1, P6 and P7) don't test the quality of their software with P1 saying; *"I don't really know how to do it"*. P7 was in agreement; *"I wouldn't know how to"* and P6 completed the trio saying; *"I'm not sure how I would"*. P4 said they don't; *"other than the odd coverage analysis being used in any of the projects that I use on a regular basis".*

Software Sustainability

When asked to define sustainable software, there was no consensus on the definition, with a whole range of different definitions discussed.

Although not everyone mentioned it, there was wide agreement that software needed to be extended for future use in order to be sustainable. P3 said sustainable software "*is software that can be taken by someone else in the future and be able to be understandable".* P4 was of the same opinion; "*sustainable software is software that's going to satisfy my user's needs for the foreseeable future".* P9 expanded a little bit; "*The simplest definition that a sustainable software project is one that I can use, tomorrow, in a few months, or a few years without needing to find the maintainer personally and talk to them"*. P12 also said;" it's *software that can be used and reused and possibly extended after the end of the initial project".*

Many of the answers were specific to an interviewee or a couple of interviewees. For example, P3 and P8 felt that sustainable software was software that could be improved. P3 said; "*Sustainable software is something that is possible to run and to add things to it or to improve things in the software"*. P8 agreed; "*sustainable software is something that can be continually developed and improved"*.

Maintainability is a view shared amongst some of the interviewees, with P6 stating; *"Sustainable software is software that can be maintained and if the current person developing it should leave the project, it wouldn't disappear".* P7 shares a similar view; *"It ties a lot into maintainability. So sustainable software needs to be maintainable. To apply the code to new problems so you don't have to start again from scratch, which by the way, is not really possible very often. So I'd say maintainability and extensibility is key".* P10 says; *"Sustainable software has to be maintainable"*.

P8 and P11 are also in agreement that sustainable software has to be reproducible, with P11 discussing this at length; *"Sustainable software for me, is this idea of reproducibility and that we should be able to reproduce results of research to verify our knowledge, to verify the literature and sustainability of that means preservation in terms of whether code and the environments are living, running, being accessed from a team and make sure that the code still runs and to check on it every now and again"*. P8 summed it up in fewer words asking; *"Is it open, accessible, reproducible? These are things that make software sustainable"*.

Another trend in the answers was the idea that the software has to be accessible with P8, P9 and P11 all agreeing on this point. With P8 saying; *"Sustainable software obviously has to have appropriate things in place to make that as easy as possible, whether it being available, whether it being accessible. So is it viable? Is it open, accessible, reproducible? These are the things that make software sustainable"*.

P9 puts an emphasis on good documentation, whereas P12 thinks that reusability defines sustainable software; *"I think the bare minimum, it's software that can be used and reused"*. P4 thinks that sustainable software is something that satisfies user's needs and P10 feels portability is important for sustainable software.

When asked if the sustainability of the software they developed was an important aspect of their working practice, 10 of the 12 interviewees said it was though both positive and negative answers were given to explain this. To add, two interviewees remained neutral, with one of them answering *"yes and no".*

P1 swung both ways on the question saying; *"Theoretically it is, yes. But the honest truth is, once you're off a project you've not really got the time and you're not being paid anymore. Ultimately, if I finish a project and move on to the next one, even if people are opening lots of issues, I'm not necessarily being paid and I'm not going to take on all these projects as an open source thing that I do on my weekend. Because I already have open source stuff that I do on my weekend. So this, I think, is a big issue that we're not addressing at the minute".*

A lack of funds or resources was a popular reason amongst the interviewees for sustainability being important, with P9 saying; *"Because it's research software, it's normally funded from grants and that grant may run out at some point, like they normally do!".* P7 agrees; *"If your software is not sustainable, it means you won't be able to run it on future architectures. So you need to do something drastic to then make the step to run on future architectures, but also*

on the functionality level. You can't extend it to take into account new scenarios, new research topics. At the same time, starting again from scratch is not feasible. There's simply no funding to rewrite large software suites from scratch".

When asked if they agree that software sustainability is a software quality, the response was also mixed on this question. Three of the interviewees were in total agreement, P3 said; *"I fully agree. And I think it goes after the functional part because if the software doesn't do what it's supposed to do, then there's no point checking anything else but it's definitely among the top things that should be checked whenever our software is meant to last beyond shelf life"*. P12 also agreed but stated; *"Yes. If by quality that also includes processes and the ecosystem around the software itself. Without things like contribution guidelines, for say, open-source software that's come out of research, the quality will invariably deteriorate over time as the software stagnates"*.

Five other interviewees agreed but felt it wasn't the only factor. P9 said; *"I would say that quality software is sustainable. That is a necessary factor in software and being of high quality. But I would not say it's the only factor, I could imagine other factors, for example, high performance code. High performance has nothing to do with sustainability so software that is sustainable is not necessarily of high quality. But it's a higher quality than the same software if it wasn't sustainable. I agree to some extent but it's not the only factor".* P5 was in agreement but did not expand; *"It is one feature of the quality of the software, I will give it that".* P6 was similarly vague; *"Software quality? I think that's one aspect of it, but I don't think that's everything"*. P10 said; *"I wouldn't define it that way but there's a strong correlation, so I have a lot of sympathy for that"*. P4 said; *"I think good software quality is going to very much help with software sustainability. I don't think it's entirely the story, you should always be encouraging high quality software. But just because something is considered by the current metrics not being particularly good, but is still demonstrable that's working and doing what the users want. I don't think it should be pitched in that circumstance"*.

Three interviewees remained neutral on the subject, with P11 saying; *"Yes and no, I think it is a software quality in that the code needs to be written in such a way that is easy for people to come into it and maintain it and track down bugs, especially if they weren't part of the original core development team. But I also think it's got a lot to do with funding and people and community. Who is going to sustain the software? And do they have the resources and backing to be able to sustain it? And I think that's the bigger problem at the minute, people just don't have time or money. So it's not prioritised"*. P8's view was; *"In one way, you can say*

yes, it is a software quality but another way you could say software sustainability is determined by a range of other software qualities. So I'd gather a few other software qualities together to determine whether or not something is sustainable as opposed to it being a single metric that you would define. So I guess it's yes and no". P2 said; *"I don't think it's intrinsic to the software itself. It's an aspect, you can have software which is sustainable because it's being supported by a community but it may be very badly written software. That's just because people are supporting it and keeping it alive. But if the software is written well, and has many good features that make it easy to maintain, then it's more likely to stay alive. So I don't know, I wouldn't say it's necessarily a property or just a property of the software"*.

P1 was the only interviewee to disagree completely; *"No because that is dynamic and it depends. As soon as you have something that has dependencies, you can't guarantee that it's going to be sustainable without someone keeping it alive without maintenance, basically"*.

When asked what the core software qualities were of sustainable software, the responses were varied. Four interviewees stated that good documentation was a core software quality, P12 saying; *"Ideally, some documentation which is tightly integrated with software itself also ideally managed as part of the same version control repository".* P3 went a bit further; "*It should be modular, should be easy to read, should use standard tools and whenever possible there should be clear answers documented in some ways, with some practical things like meaningful variable names, and the sort of things that will help whoever reads it afterwards, to figure what the goal is".* P5 explained; *"Ideally, it would be good to check the documentation fits accessibility standards. There are some online tools that you can use to test for accessibility for the documentation".* P8 related it to fair principles for data; *"I mentioned about fair principles for data, effectively it's whether it's findable, accessible and reproducible".*

Three of the interviewees referred to *maintainability* as a core software quality of sustainable software in their answers but did not expand further. Further to this, P7 said; *"Being maintainable is not necessarily a core quality but reflective of core qualities of sustainable software".* Three of the interviewees stated that accessibility was a core software quality. P2 said; *"There is some kind of aspect of making the code accessible and maintainable, because it's somehow easy to get into. So the fact that you can attract new developers by making it accessible makes it easy to get started with".* Two interviewees felt that community was a core software quality, P11 said; *"Community, I think you need the people to be able to do the work and they need to be resolved well enough to do the work".* P2 felt it was very important; *"I think having a linear living community of developers who are working on it, that's the most important*

thing. The fact that there's a group of people who are working on it. Of course, it's like software which is kept going because they have a team". Two interviewees also referred to *portability,* P4 saying; *"Portability has to be an absolute key one because the architectures we were using 10 years ago are not the architectures we're using today so anything that aids portability in the languages that you use by actually doing what the language guarantees rather than getting in mucky corners and hoping you're going to get away with it".* P7 also listed portability as a core software quality. P5 and P12 referred to documented processes with P12 saying; *"publicly documented processes around how version control should be used, and how people identify issues, suggest fixes for issues, suggest and possibly contribute new features".*

There were also a number of qualities that were only mentioned by specific interviewees such as having a robust build system as P9 said; *"It needs a robust build system that doesn't rely on one person's environment".* P4 referred to easy installation; *"It's all well and good to write the most wonderful software ever and I am looking at the Python rule to a certain extent but if the users can't install it, it's utterly useless, it will never get used. It has to have an easy install and test procedure to actually get used and once it starts getting used, then you have some hope of sustainability".* Continuous integration, extensibility, funding, portability, packaging, performance, version control and having a basic test suite were others that were mentioned.

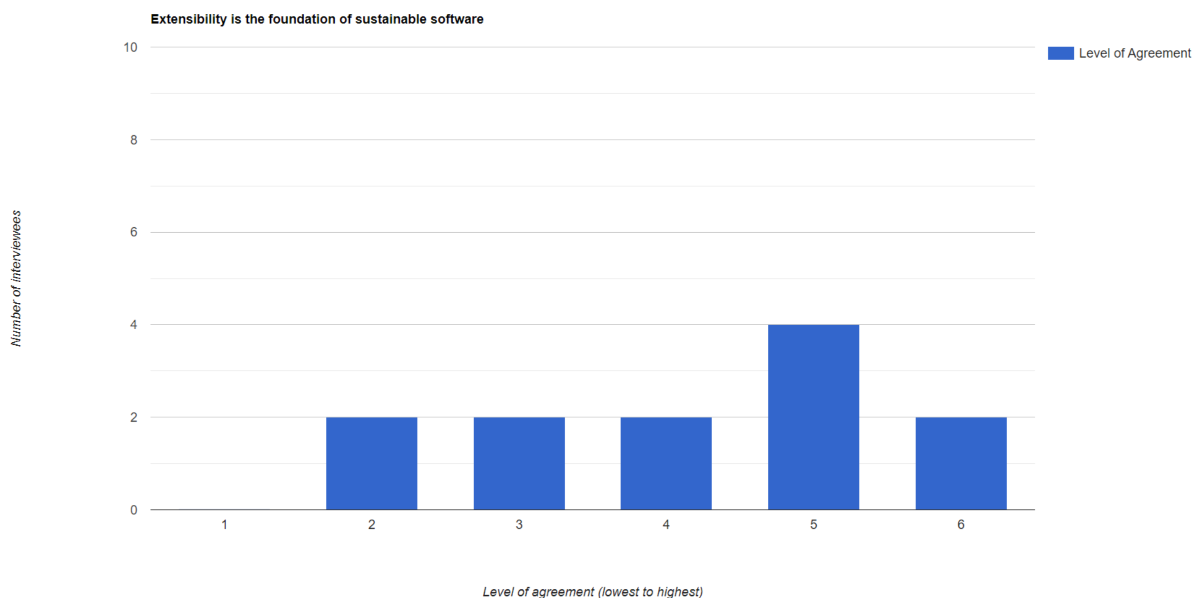

Figure 30 – Extensibility is the foundation of sustainable software?

While asking to what extent extensibility is the foundation of sustainable software, Figure 30 illustrates that there is no overwhelming consensus among the RSEs that were interviewed.

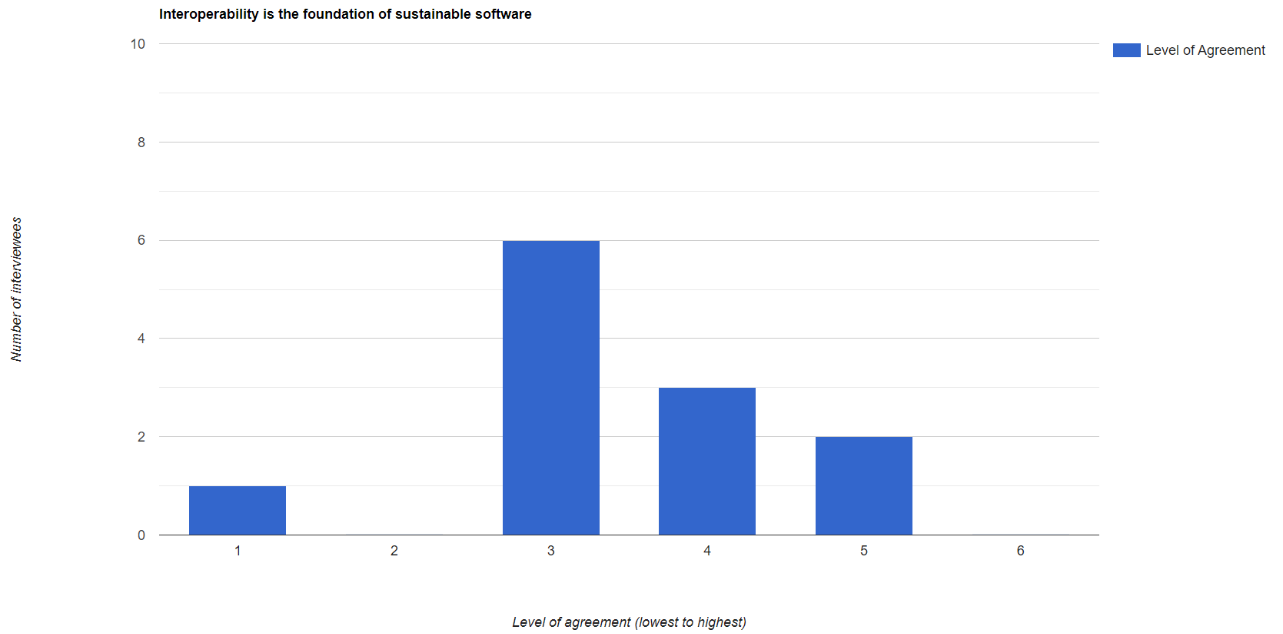

Figure 31 – Interoperability is the foundation of sustainable software?

The majority of interviewees sat on the fence in Figure 31 when asked if they agreed that interoperability is the foundation of sustainable software, although someone disagreed completely and no one completely agreed. This could be seen as a near consensus that the interviewees don't agree or disagree one way or the other on this question.

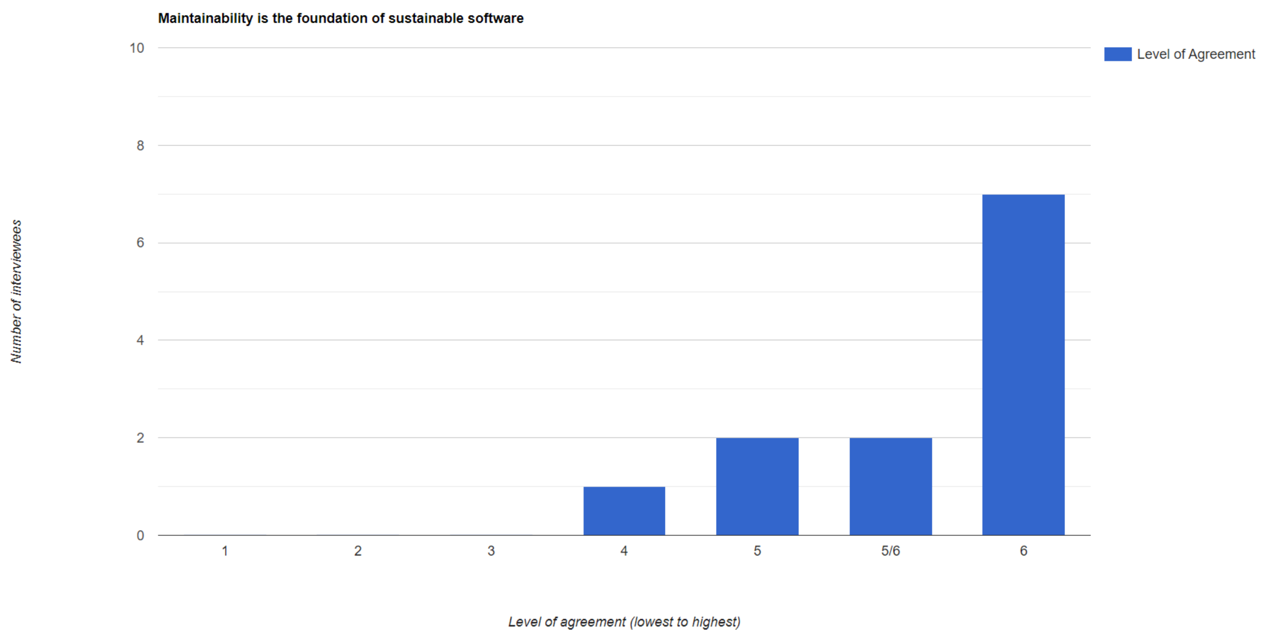

Figure 32 – Maintainability is the foundation of sustainable software?

Figure 32 shows that the majority of interviewees completely agree that maintainability is the foundation of sustainable software. To add, P8 and P9 were undecided but cited 5 or 6 in their answers, giving further proof of agreement.

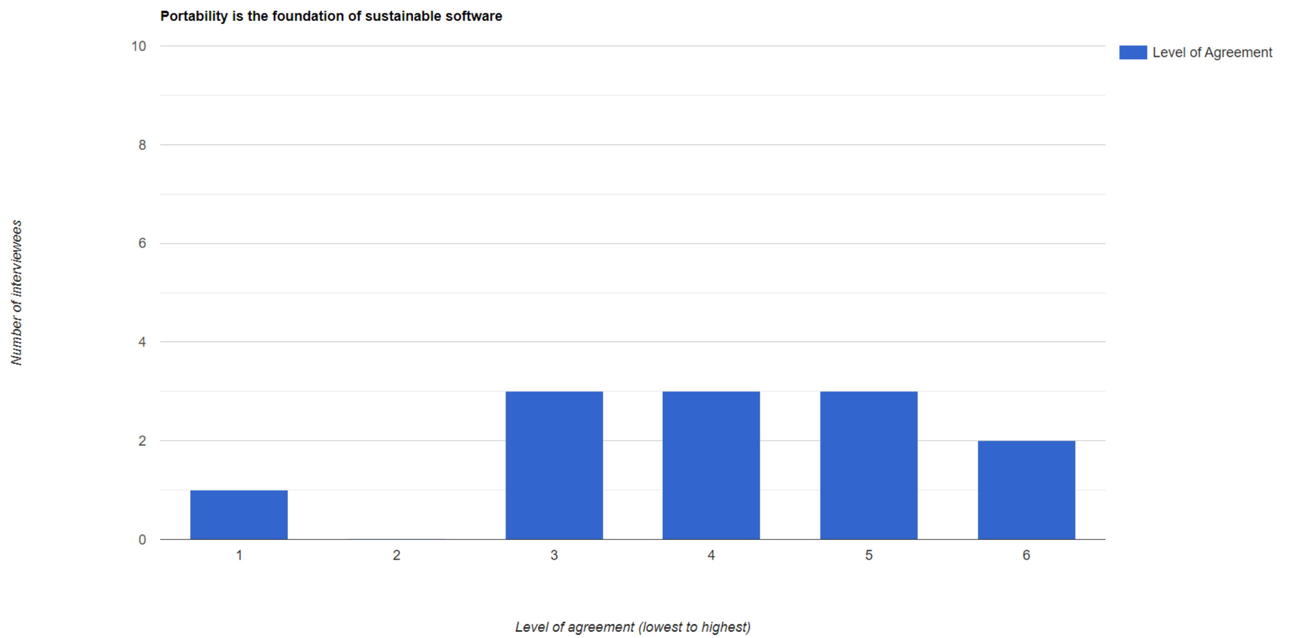

Figure 33 – Portability is the foundation of sustainable software?

Figure 33 demonstrates that there is no consensus amongst the interviewees on whether they agree that portability is the foundation of sustainable software, with varying answers all across the board.

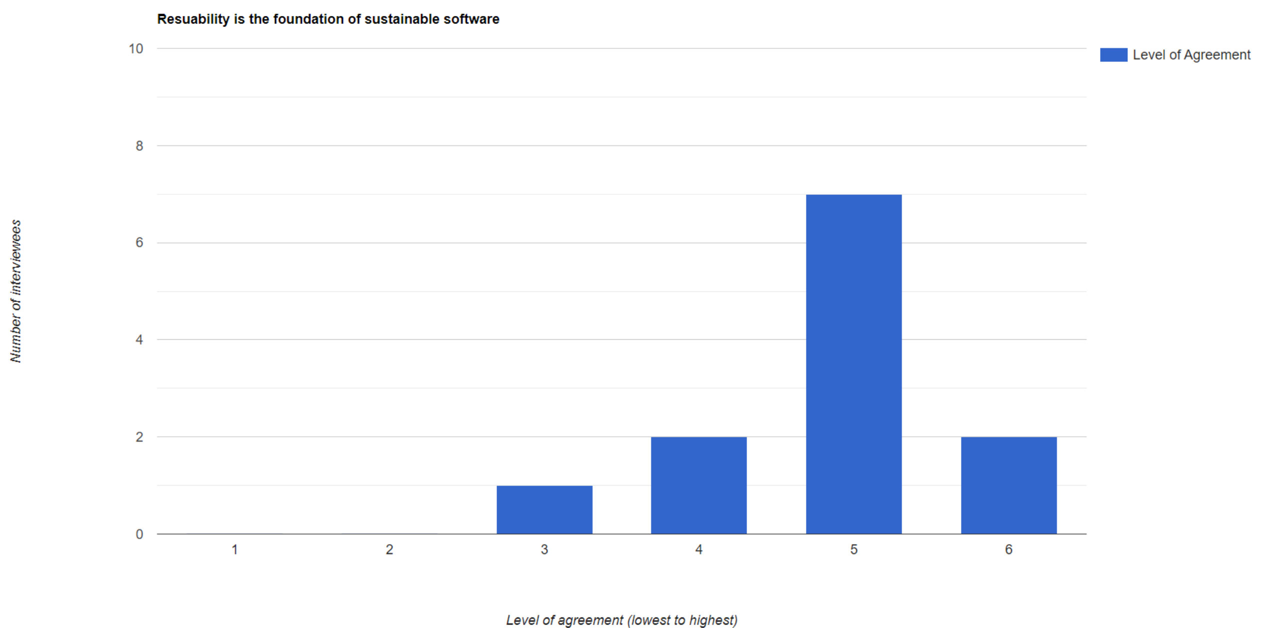

Figure 34 – Reusability is the foundation of sustainable software?

Figure 34 shows that although the majority of interviewees don't completely agree that reusability is the foundation of sustainable software, there appears to be a consensus that it is quite important, with the majority opting to answer 5 on the scale.

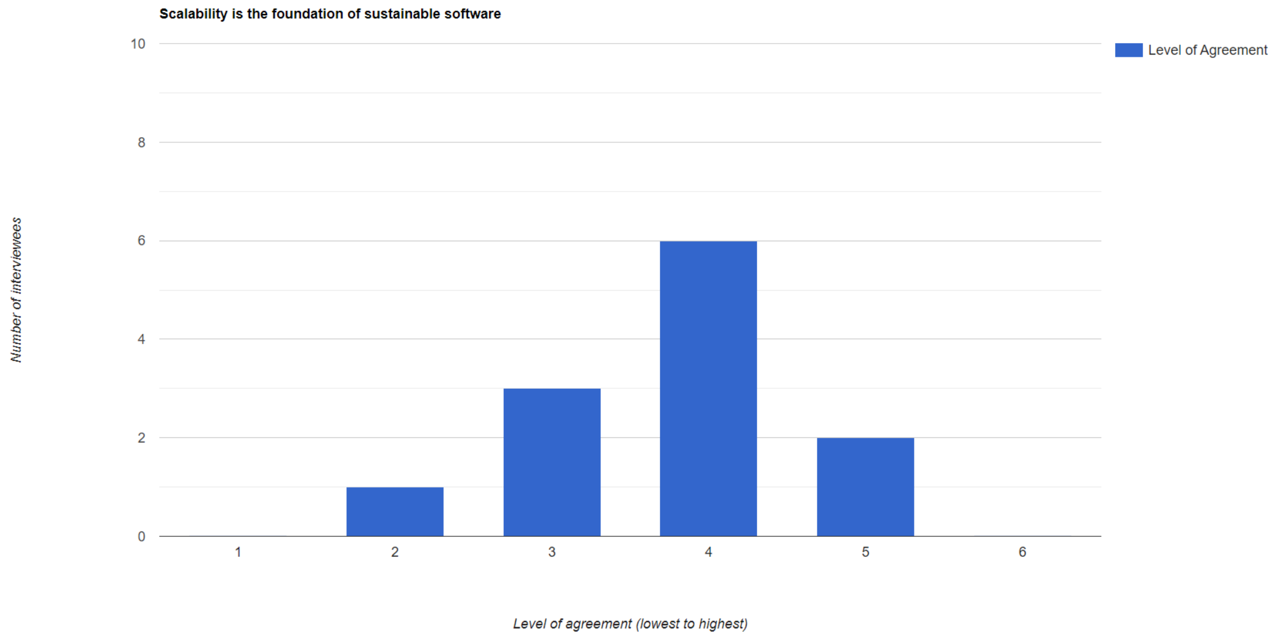

Figure 35 – Scalability is the foundation of sustainable software?

In Figure 35, when asked if scalability is the foundation of sustainable software, the majority of interviewees didn't sway one way or the other, opting to remain in the middle. There was no one who either completely agreed or disagreed with the question. This could be seen as a near consensus that the interviewees are quite neutral on the subject.

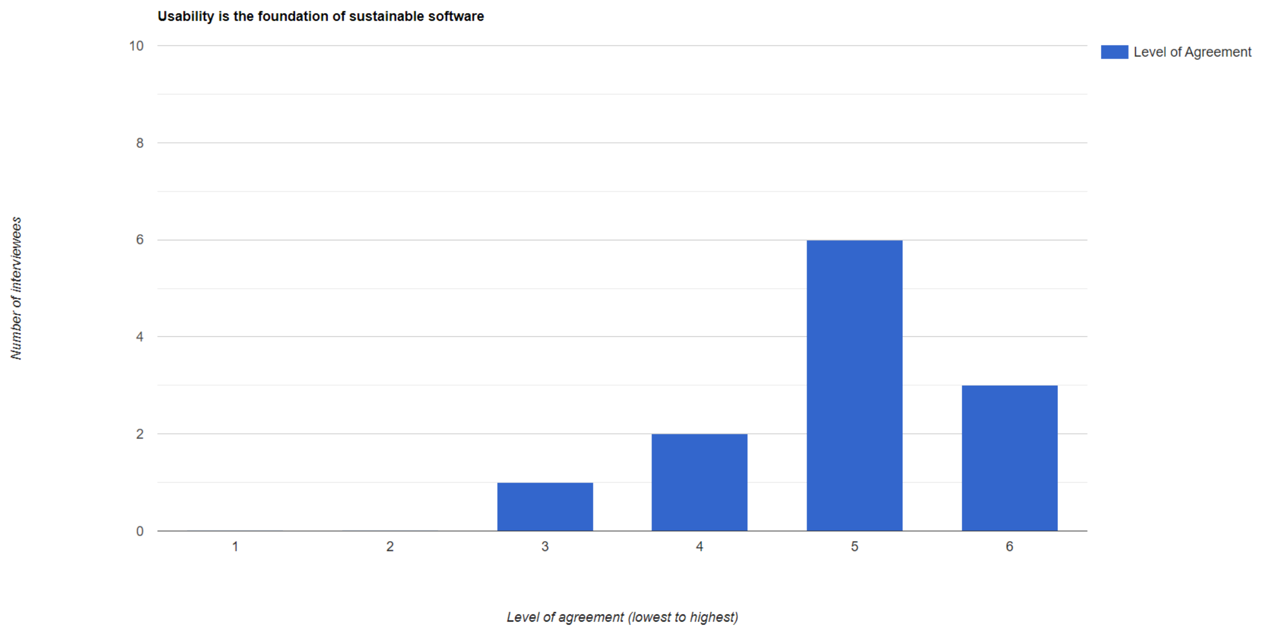

Figure 36 – Usability is the foundation of sustainable software?

Figure 36 demonstrates that although the majority of interviewees don't completely agree that usability is the foundation of sustainable software, there appears to be a consensus that it's quite important, with the majority opting to answer 5 or 6 on the scale.

Seven interviewees don't measure the sustainability of their software, whereas two do measure it. Additionally, P12 said; "*There have been times when we've been asked to, or we've asked to, or we've chosen to do a sustainability review. In those circumstances we've used the Software Sustainability Institute evaluation process but it's sort of a framework for evaluation sustainability. We've used it on a couple of occasions but it's not something we do for every project*". P9 said; *"I have measured it, but not actively"*. And P3 said; *"Not with any specific metric, it's knowhow rather than a specific metric that measures that"*.

When asked their reasons for not measuring it, the main response was lack of knowhow, with P1, P7, P8 and P11 all stating this. P11 also mentioned the lack of time with P6 admitting it had never occurred to them to test the sustainability of their software and P5 sees their involvement as being quite early on in the product development so doesn't have to worry too much about testing the sustainability of the software as it will be someone else's problem. P2 and P8 also mentioned the lack of specific metrics or methods to carry out the process.

Sustainability Design

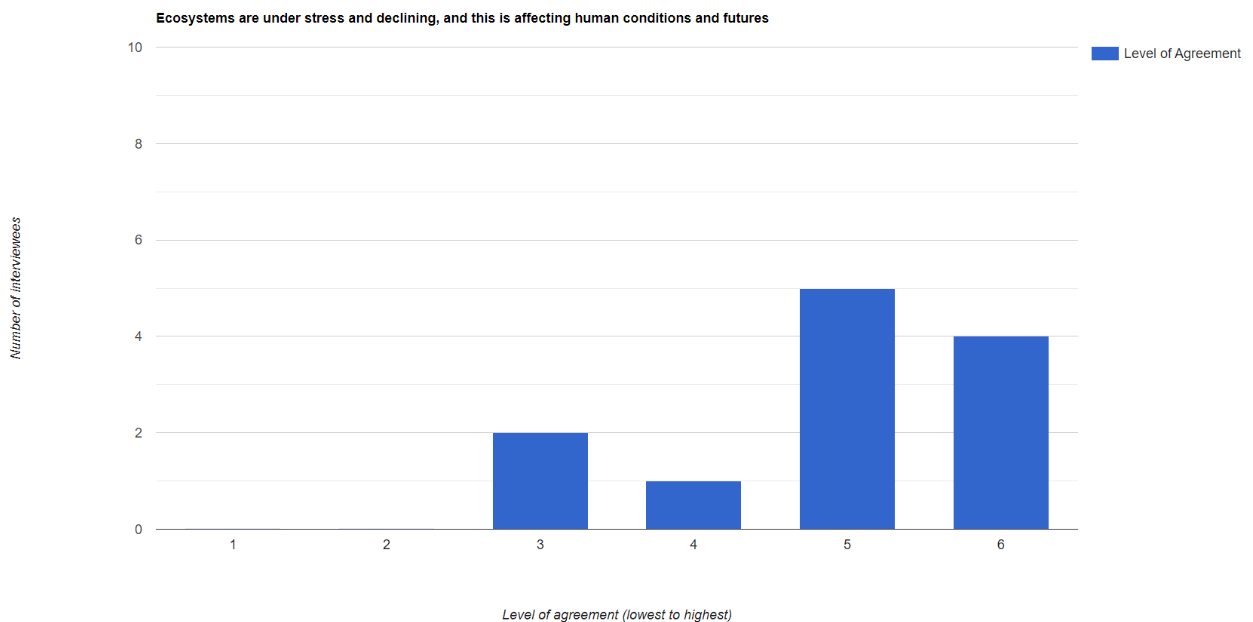

Figure 37 – Ecosystems are under stress and declining, and this is affecting human conditions and futures?

Figure 37 shows that none of the interviewees disagree that ecosystems are under stress and declining and it is affecting human conditions. The data shows that a majority stray towards completely agreeing.

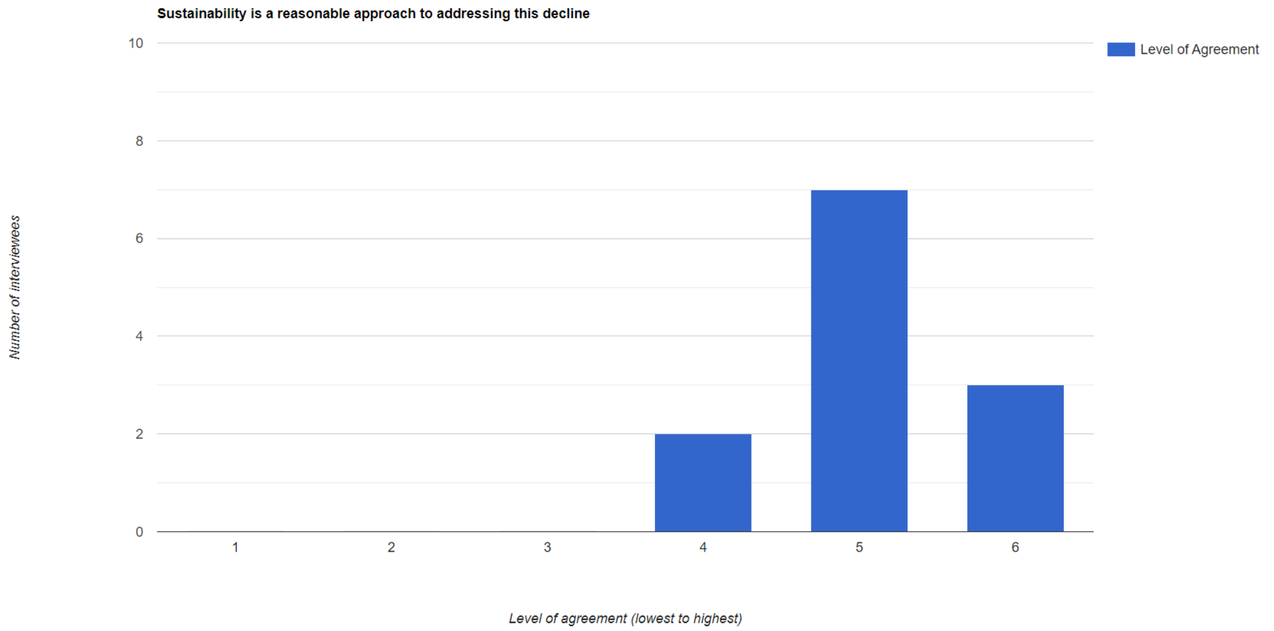

Figure 38 – Sustainability is a reasonable approach to addressing this decline?

Figure 38 shows that the interviewees are in agreement that sustainability is a reasonable approach to addressing the decline of ecosystems, with the only disagreement being how much they agree with the statement.

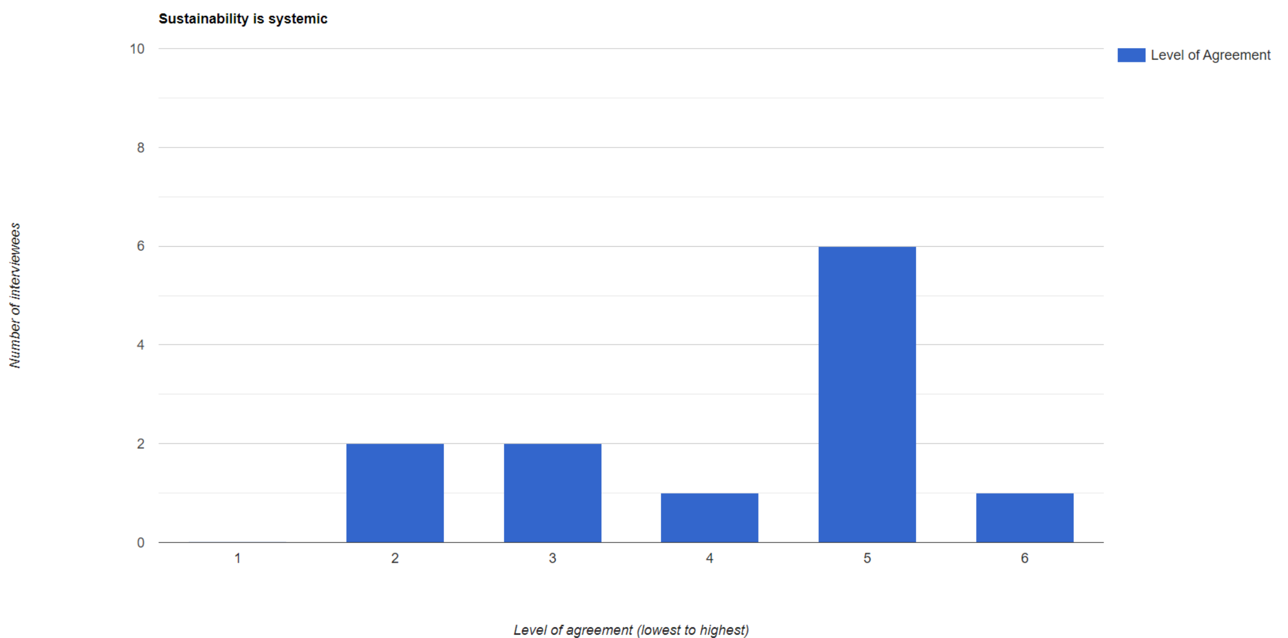

Figure 39 – Sustainability is systemic?

Figure 39 demonstrates the lack of consensus among the interviewees when asked if sustainability is systemic. Though there is a majority that agree that it is, it's not conclusive.

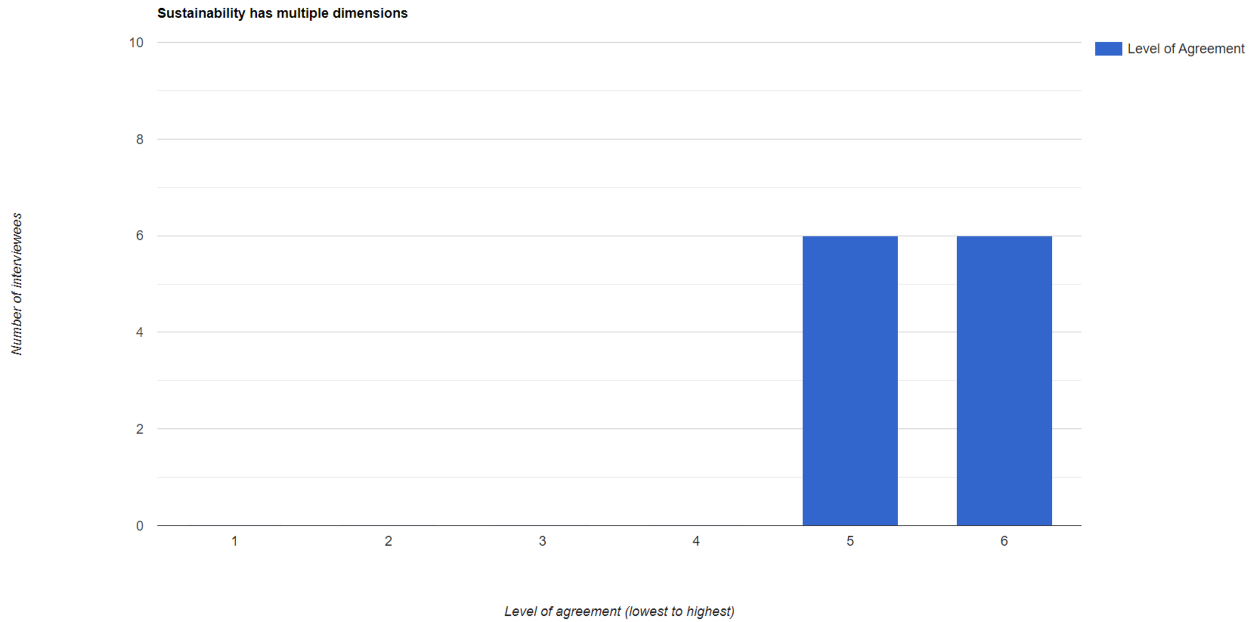

Figure 40 – Sustainability has multiple dimensions?

Figure 40 concludes that the interviewees completely agree that sustainability has multiple dimensions, with just a slight difference on how important, with six stating 5 on the scale and the other six stating 6 on the scale.

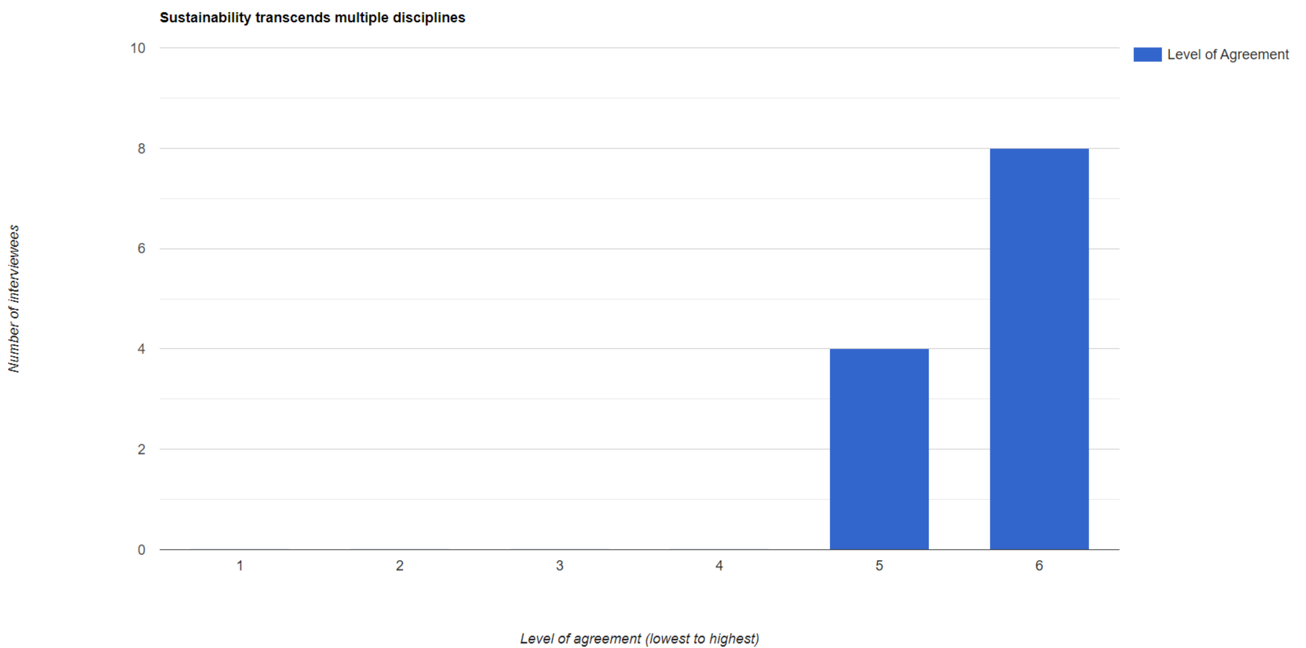

Figure 41 – Sustainability transcends multiple disciplines?

When asked if sustainability transcends multiple disciplines, the interviewees agreed it did, as Figure 41 demonstrates.

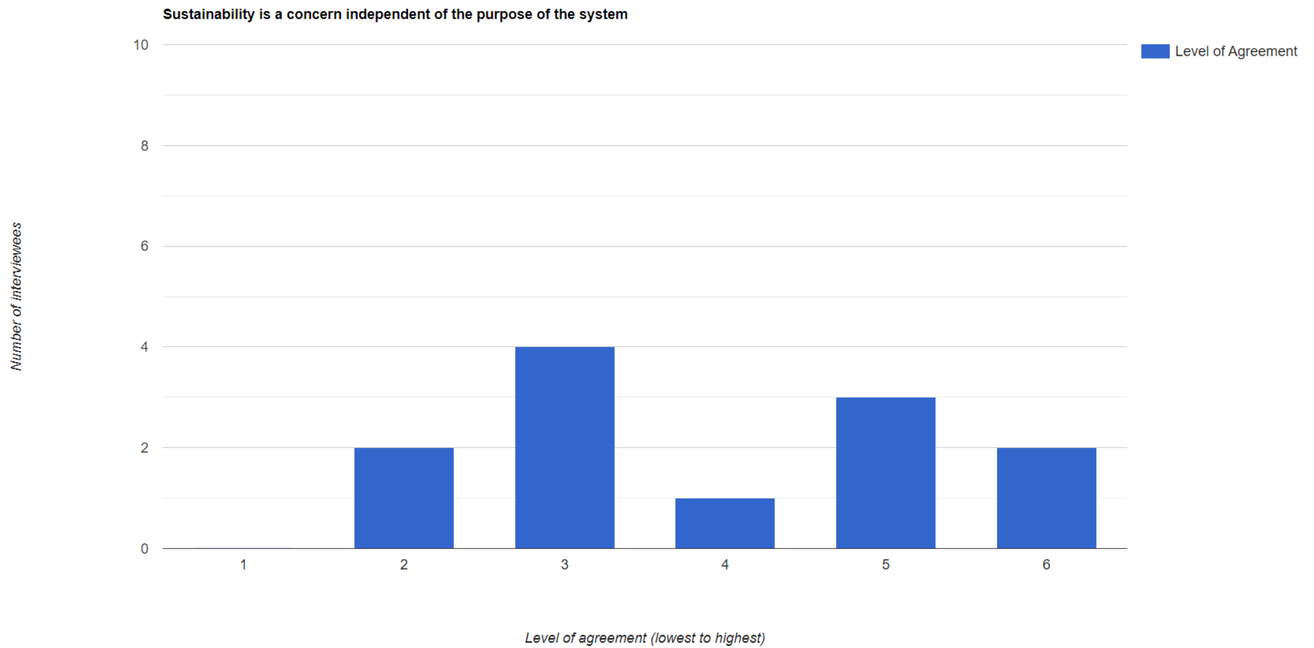

Figure 42 – Sustainability is a concern independent of the purpose of the system?

The data in Figure 42 demonstrates that there is no clear consensus, with varying answers and although some interviewees state that they completely agree that sustainability is a concern independent of the purpose of the system, there's some that tend not to agree.

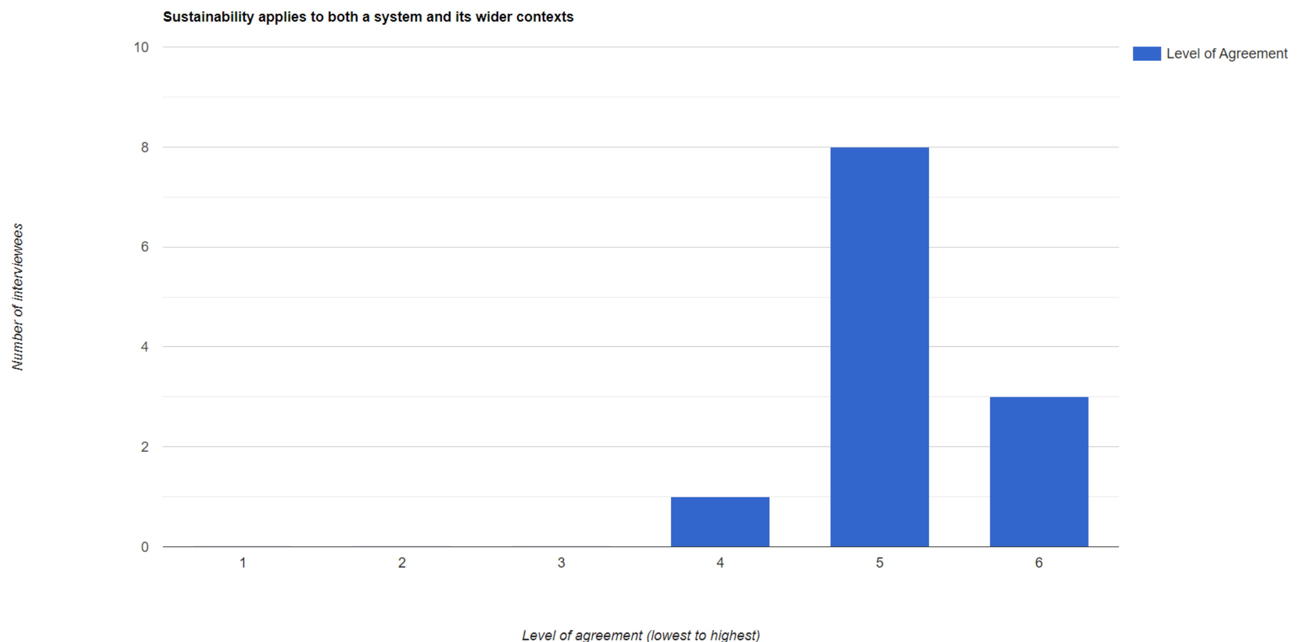

Figure 43 – Sustainability applies to both a system and its wider contexts?

Figure 43 shows conclusively that the interviewees are in agreement that sustainability applies to both a system and its wider contexts, with the majority opting to choose 5 on the scale.

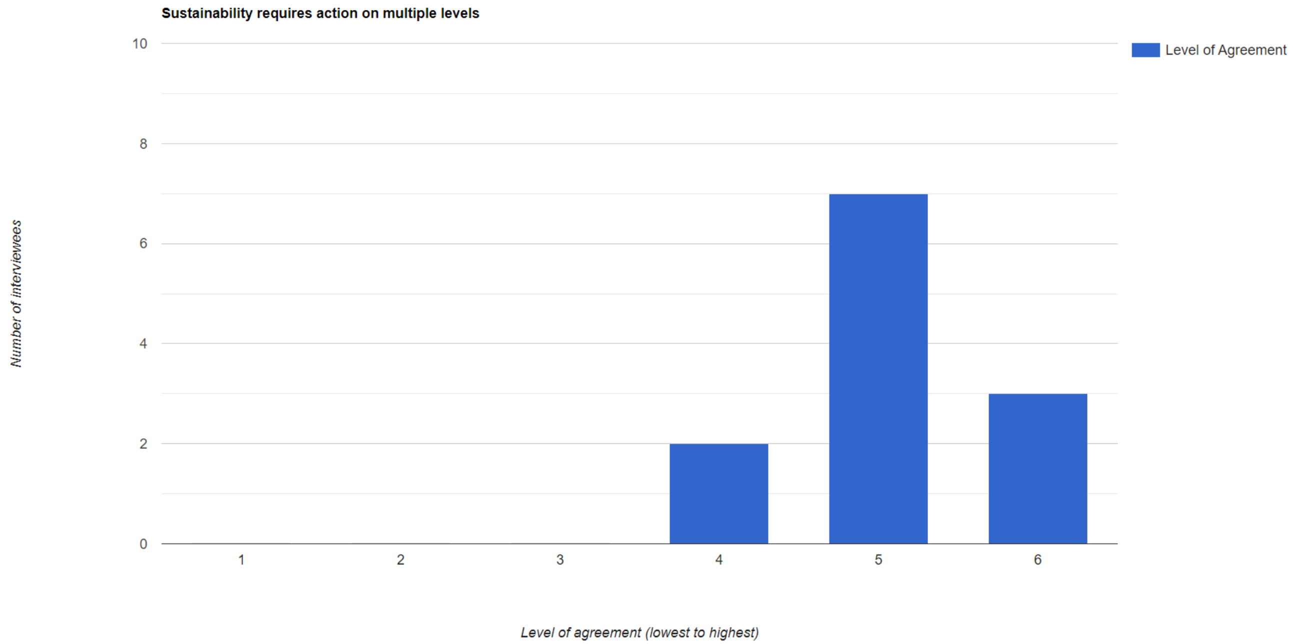

Figure 44 – Sustainability requires action of multiple levels?

When asked if sustainability requires action on multiple levels, the interviewees felt it was quite important, as the data proves in Figure 44. Though just three of the interviewees stated it was Very Important (6 on the scale), with the majority opting for 5 on the scale.

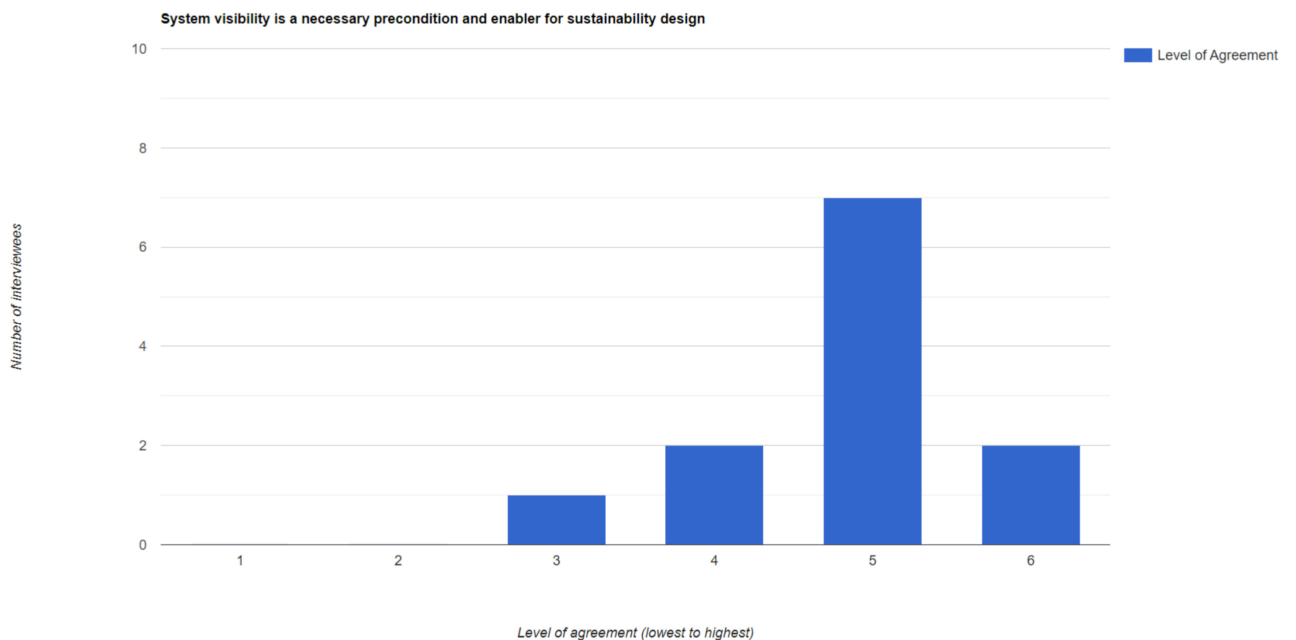

Figure 45 – System visibility is a necessary precondition and enabler for sustainability design?

Figure 45 demonstrates a near consensus amongst the interviewees on the issue of system visibility and if it's a necessary precondition and enabler for sustainability design. Despite only

two interviewees opting for Completely Agree (6 on the scale), the majority have selected 5 on the scale, suggesting that they are almost in complete agreement with the statement.

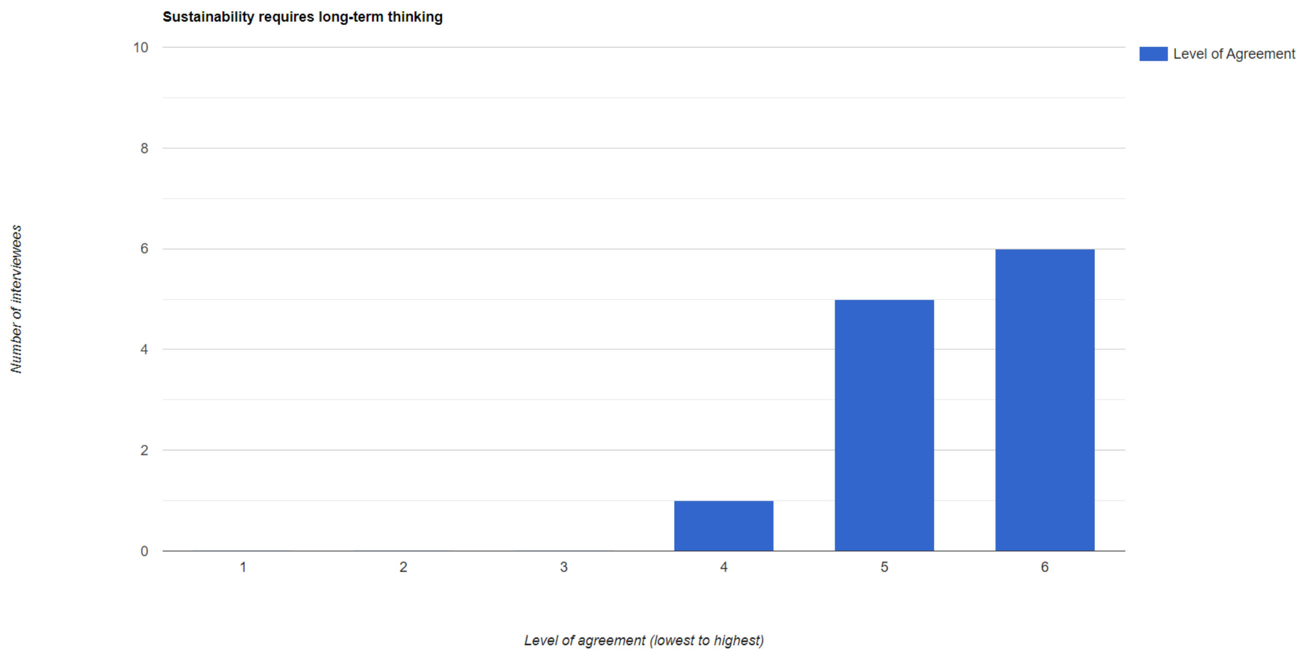

Figure 46 – Sustainability requires long-term thinking?

Although not all interviewees were in complete agreement, the data in Figure 46 is pretty conclusive and there's a consensus that agrees that sustainability requires long term thinking, with half of the interviewees completely agreeing with the statement.

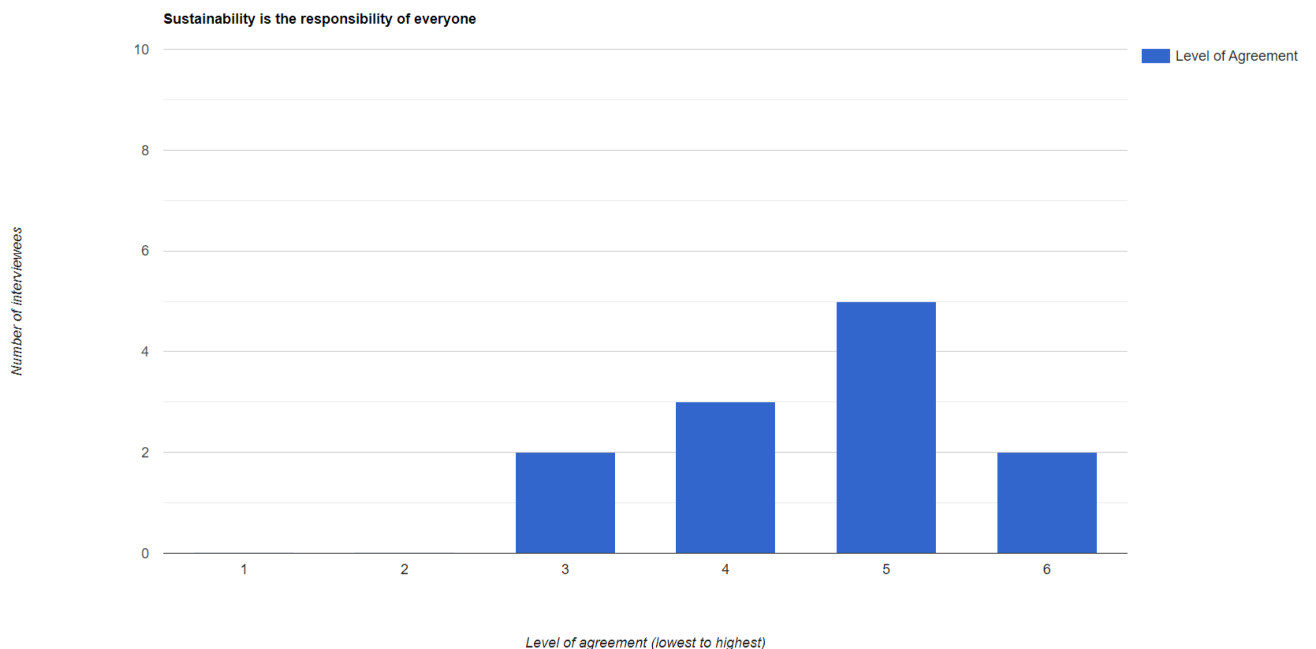

Figure 47 – Sustainability is the responsibility of everyone?

Figure 47 shows that although there isn't a clear consensus, the interviewees stray towards agreeing that sustainability is the responsibility of everyone, during their whole lives, including

at work. The interviewees tended to stray towards agreeing but only two were willing to completely agree with the statement, with many of them opting for slightly lower marks on the scale, with the majority selecting 5 on the scale.

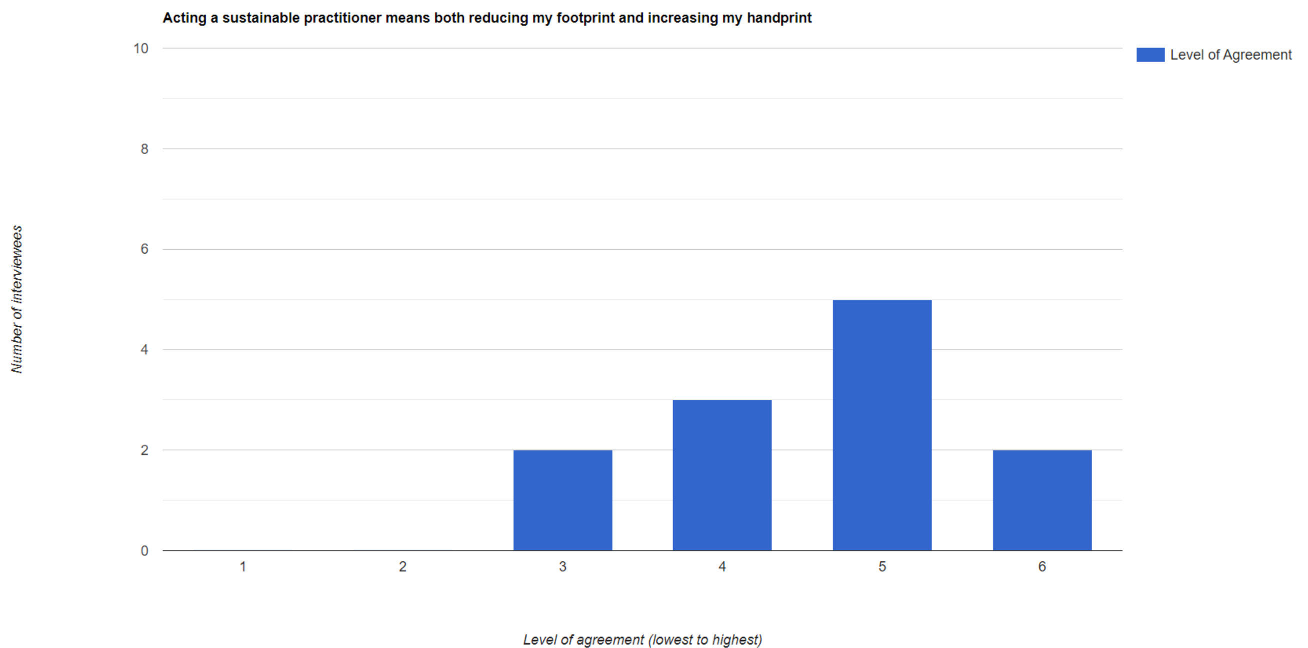

Figure 48 – Acting as a sustainable practitioner means both reducing my footprint and increasing my handprint?

Again, Figure 48 shows no clear consensus on the statement, with the answers varying between 3 and 6 on the scale. This means that opinions differ on whether acting as a sustainable practitioner means both reducing footprints and increasing handprints.

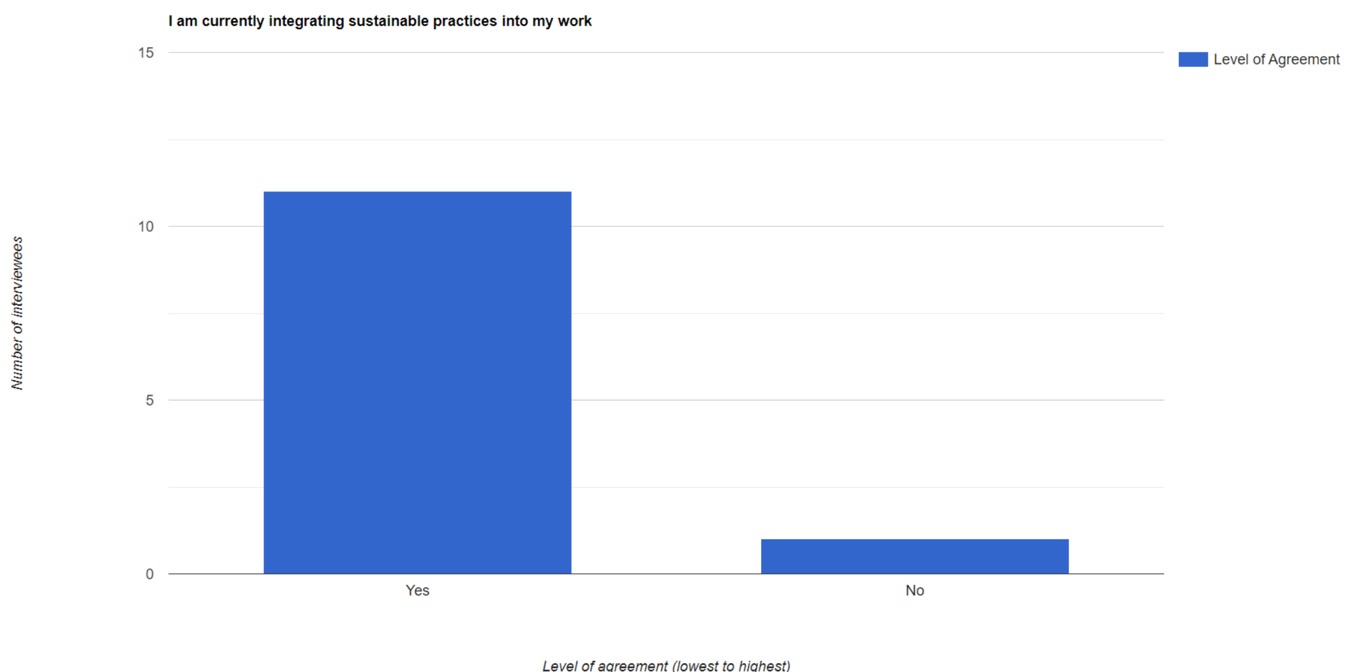

Figure 49 – I am currently integrating sustainable practices into my work?

Figure 49 paints quite the picture, with an overwhelming majority of interviewees telling us that they are currently integrating sustainable practices into their own work, with just one of them conceding that they don't.

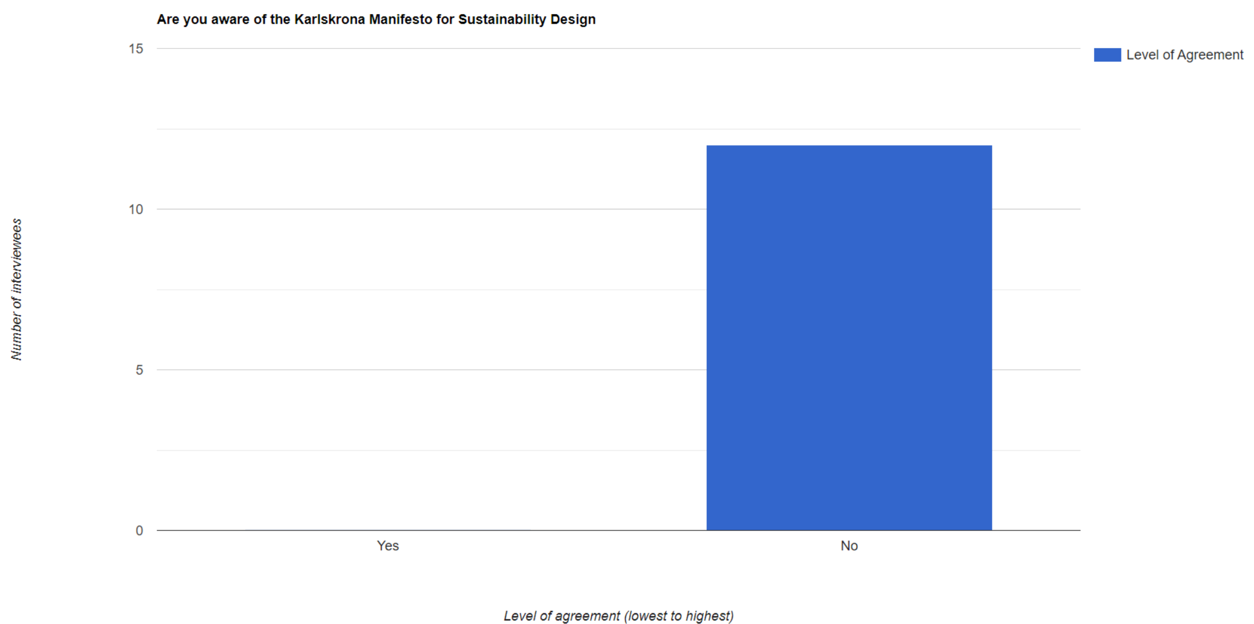

Figure 50 – Are you aware of the Karlskrona Manifesto for Sustainability Design?

Figure 50 is incredibly conclusive, with all interviewees stating that they weren't aware of the Karlskrona Manifesto for Sustainability Design before the interview had taken place.

Summary

In conclusion, we have a set of 12 interviewees with 12 very different sets of answers. As we've demonstrated, there is some agreement amongst the questions but by and large, there's a lot of differing opinions. This is to be expected as although all interviewees are RSEs, their backgrounds differ considerably as well as their areas of expertise and the institutions that they are currently employed in.

This study interviewed 12 RSEs, previous studies of this kind have been quite limited but the number of interviewees is fairly similar to other studies that have been conducted.

A study by Groher and Weinreich in 2017, focusing on sustainability concerns in software development projects, interviewed 10 software project team leads (Groher & Weinreich, 2017). A 2019 study by Rosado de Souza et. al on what makes research software sustainable collected data in two phases, during the first phase, nine developers were interviewed and during the second phase, a further 19 research software engineers were interviewed (Rosado de Souza et. al, 2019).

Also noteworthy is the fact that some interviewees declined to answer certain questions, or couldn't provide adequate answers. This does have an impact on the results on occasion because although the study was conducted with 12 interviewees, certain questions only have 11 answers, so perhaps does not give as fair a reflection on the interviewees as it could do.

During the interviews, it would have entirely possible to influence the respondents to give answers, however, this was decided against as it may have led to the interviewees giving answers that weren't true, we did not want to force people to answer questions that they couldn't answer. Untrue answers would have affected our study and could have led to incorrect conclusions being drawn with regards to understanding software sustainability in the field of Research Software Engineering.

Section 6: Discussion

In this section we discuss the results and provide some analysis. We will see if the findings relate to the literature and what has gone before. The principle themes and key findings will be looked at in depth here.

Background

The backgrounds of interviewees are always likely to be quite diverse, especially when those interviewees are spread out amongst various institutions and working in different areas. Amongst the results there are many differences, such as ages, years of experience and the main role and focus in their job. From our data, 66% of RSEs were male and 33% were female, this is different from the 2018 RSE Survey, which found that 80% were male and 20% female. The survey also found that the RSEs educated to a doctorate level were 70%, whereas our study was 100% (Philipe, 2018).

Some of the interviewees' first degrees were in the same subjects, namely computer science and physics, but the vast majority were all in different areas including natural sciences, biophysics, chemistry and geophysics. This was the same with their PhD degrees, with all interviewees' doctorates varying in subject matter except for two who had both completed their PhD in computer science, the rest was a variety which included theoretical chemistry, civil engineering and marine macroecology.

The domain for which the interviewees primarily develop software was also, as expected, a wide variety of different domains. The only trend here, being that three of the interviewees develop software for no particular domain whereas the rest all develop software for various domains such as condensed metaphysics or material science, plasma physics and biophysical and complex fluids.

Apart from all being RSEs, there are some clear trends amongst the interviewees and their backgrounds, with all of them being educated to a PhD level. These findings differ from the results found in the RSE Surveys, which demonstrates a wide variety of interviewees educated at various levels between undergraduate, masters and doctorate degrees.

The only trends amongst the interviewees' roles and focus was that four of them stated that their role and focus varied. The rest of the interviewees all provided different answers when

asked what their role entails and all work across a variety of different areas in software engineering, including developing software to improve research, biophysical and complex fluids and research and teaching.

The interviewees' years of experience in software engineering was quite interesting, with half of them having 10 years' experience and another three with 25 years' experience. This tells us that the experience of the RSEs is quite strong and despite Carver et al. (2021) claiming that there is a frequent turn-over of developers in software engineering, these pool of interviewees seem to represent the opposite of that view.

Software Engineering Education and Training

From the interviews, we observed that although four of the interviewees have had some sort of formal training in software engineering and development, the majority of RSEs admitted that they had none at all. The overwhelming response from these interviewees was that they had received informal training within their roles and also learned on the job. This shows that although the education and training opportunities are out there, much of the interviewees' training has been on the job and through informal training they've received in their roles.

There was some consensus across the interviewees in that they all acknowledged that the training they'd received, whether formal or informal, had helped them in their current roles though they weren't prepared to elaborate on how it had helped them.

There's also some clear trends in the type of training that the RSEs have undertaken in their formal education and professional development. For example, with all but one interviewee receiving no training at all in software requirements. Similarly, although one or two interviewees said they had received some training in software architecture, the majority of them had received none. The answers were the same for software maintenance, with the majority of RSEs acknowledging that they had little training in this area. Software configuration management was similar in that the vast majority of RSEs had no training in the area, as was software engineering management, software engineering process and software engineering professional practice and software engineering economics. This suggests that these specific areas are not considered very important for training and there's a lack of coverage of these areas in the formal training that has been received by the interviewees.

On the other hand, there were certain subjects where the majority of RSEs had received some kind of training, such as software coding with just one RSE having had no training whatsoever

in the area. The study threw up some odd results such as software engineering models and methods. Although the majority answered that they'd received absolutely no training in the area, three of the interviewees had received quite a bit of training. Finally, there were some of them where there was no consensus at all with interviewees giving various different answers between no training and a lot of training. This was the case for software design, software testing and software quality in particular.

Whether they had received formal training or informal training, all interviewees acknowledged that their various examples of training had helped them in their current roles.

The interviewees also demonstrated a lack of formal RSE training and professional certifications, which aligns with Carver et al. (2021), who have observed a lack of formal RSE training at undergraduate level and of professional certifications at postgraduate level. They believe that this is limiting the emergence of a truly professional class of career RSEs (Carver et al., 2021).

In conclusion, there's some areas that receive a substantial amount of coverage in the training that the interviewees have received but there's quite a few areas that have received little coverage in this training, formal or otherwise.

Software Development Lifecycles

The interviewees demonstrated that although they used different methods, there was agreement that these methods were best suited to the projects undertaken. There were some clear trends amongst those that do use lifecycle models, with the Iterative model and Agile being the most common answers. Although one interviewee uses Waterfall, it's perhaps a surprise that it wasn't a more popular answer as it's one of the most well-known and recognised lifecycle models out there.

One interviewee admits that their lack of formal training has led to them devising their own lifecycle model that is "*similar to agile, but also beyond agile and doesn't fit into the traditional ones*". This is interesting and suggests that because of the lack of formal training, the participant is just making their own way in the area without any proper guidance or trained skills.

Nine of the 12 interviewees acknowledged that their adopted model (or way of working) was best suited to the projects that they work on. This shows that the interviewees are aware of software development lifecycle models and are using them to their full advantage in their roles. Those interviewees are aware of their pros and cons and accept that these are the best suited models to use. Interestingly, one interviewee thinks that their lifecycle model isn't best suited. This shows that the interviewee has recognised the limitations of their working practice.

For those that don't use one, it begs the question, would their project be more sustainable or could the project be improved with the use of a software development lifecycle? This is perhaps something that can be explored in the future.

When asked to provide approximate percentages on the amount of time spent on different activities on a typical project, the responses were of a wide variety. This simply shows that the roles and responsibilities of the RSEs are very different and is to be expected.

Software Testing

All interviewees (100%) carry out software testing in their roles, which is what would be expected. This is more than the 2018 RSE Survey, which found that 81% of RSEs did their own testing (Philipe, 2018). Although all interviewees carry out software testing, only eight of them plan for it, with four admitting that they don't plan at all. Of those that don't, they tend to just write tests as they go along rather than putting any planning in place.

When asked what types of testing they used for this, there was some agreement amongst the data. Unit testing was the overwhelming choice of testing amongst the interviewees, with nine of them acknowledging its use. Integration testing was another popular choice, with half of the interviewees admitting to its use. Input testing was also mentioned by two of the interviewees. It seems that the RSEs tend to use dynamic analysis methods rather than static analysis. It is also evident and perhaps worrying that there seems to be a big focus on unit testing and integration testing but not much else. Code needs to be tested from different perspectives and it seems that the focus here is very one-sided leaving the software exposed to various problems. Dynamic analysis has its benefits but it also has its limitations.

As always, there were a number of responses that were specific to the interviewee, with one seeming to employ snippets of different types of testing which includes bits of input testing. Another uses a variety of regression, system and "golden answer" tests (which are more examples of dynamic analysis). Another applies visualisation to their testing, having worked

in that industry previously. This highlights the wide variety of knowledge and experience amongst the RSEs and shows that one or two of them are willing to go beyond just unit and integration tests.

When asked about *coverage-based, fault-based* and *error-based testing*, the responses were varied. Much of the RSEs admitted to experience with *coverage-based testing* but mostly struggled to even understand the concept of *fault-based testing*, although two RSEs had experience with it, and another four had experience with *error-based testing*. Most of the answers were given only after the definitions had been given for them. Despite these three types of testing being set out in *Software Engineering: Principles and Practice,* the RSEs have limited knowledge of them.

The answers were very user specific when asked about the most reliable type of testing. Again, unit testing was a common answer among the RSEs, with integration testing, correctness tests, regression tests, error-based testing and coverage-based testing all getting mentions. There was one RSE that disagreed with the question, being of the opinion that different types of testing served different purposes so it wasn't a case of which was the most reliable.

Once again, the challenges around software testing saw more varied answers from the RSEs. The main challenges we observed were writing the tests, being unaware of what the correct answers should be and time constraints. More challenges included lack of knowhow, complex code, substantial code bases, getting people to actually write the tests and getting people on board.

Two of the RSEs mentioned that they used C++ and Python for their software. The 2018 RSE Survey asked RSEs what language they prefer and split it into three choices. The most preferred programming language was Python, with 76% of RSEs choosing it as their preferred language. C++ was their second preferred language, with 38% opting for it (Philipe, 2018).

Software Quality

All but one of the interviewees consider software quality in the development of their software and despite them giving varying answers on what qualities they consider, there were some clear trends. Documentation was one of the most common answers, with a number of RSEs considering it a quality. Interestingly, one interviewee said *"the whole point of testing is actually about software quality"*.

Reliability, extendibility, modularity, robustness and repeatability were all mentioned by single RSEs as software qualities they considered. Reliability was mentioned by Nguyen-Hoan et al. (2010), who felt that any improvements in the development of scientific software must take into account the factors; reliability and functionality, which are the top two non-functional requirements that are considered the most important by the developers of scientific software developers. Extendibility, although not referred to as such, ties in with Lago et al's (2015) belief that sustainability is to "*preserve the function of a system over an extended period of time*".

The lone RSE who doesn't consider software quality admits that this is because they have no metrics for achieving software quality and they also acknowledge that their vast experience in software engineering (over three decades) has probably held them back because they are stuck in their ways and find it difficult to change so far down the line.

The aim of writing good quality software seems to be woven through the RSEs' answers and it seems that they all want to achieve the same end goal of this, they just all seem to have differing methods of how to get there. We think that this is down to the different domains that they are all working in, and also down to the training they've had and the different experiences they carry with them. This study proves that there's many different answers for achieving sustainable software.

When asked if they were aware of ISO:9126 and ISO: 25010:2011, although a small minority had heard of them, most of the RSEs hadn't. This seems to have been a trend across the interviews in that the RSEs were mostly unaware of any official certifications, which isn't surprising as most haven't had any formal training or hold any certifications in software engineering.

On the subject of testing the quality of their software, responses were once again varied. There were four RSEs who said they did and another two who claimed that they did but it wasn't formal. Another two acknowledged that they do bits of testing on the quality. Three RSEs said they didn't test the quality of their software, with all three in agreement in their reasoning, which was simply a case of not knowing how to do it. Another RSE stated they don't really test the quality other than the odd coverage analysis being used on their regular projects.

Software Sustainability

All interviewees in this study demonstrated a very broad understanding of software sustainability and all shared differing views across the whole section, providing various different answers to the questions. The variety of answers show that there isn't really a consensus, although there were some agreements.

When asked to define sustainable software, the answers varied but there was some clear agreement in and amongst the data, such as defining sustainable software as software that needs to be extended for future use. P9 gave perhaps the best definition *"sustainable software is software that's going to satisfy my user's needs for the foreseeable future"*. This agrees with Lago (2019), who has defined sustainability as the "*capacity to endure*" and emphasised the need to "*preserve the function of a system over an extended period of time*". It also agrees with Amsel et al. (2011), who suggests that sustainable software engineering aims to create reliable, long-lasting software that meets the needs of users.

Sustainable software was also defined as something that was maintainable, which is in agreement with Carver et al. (2021), who recognised the need for developers to maintain software in order to keep it up to date with technology and user needs. Out of interest, maintainability is one of the eight software quality characteristics set out in ISO/IEC 25010:2011.

Software with the ability to improve was another common answer, with P8 giving the best definition *"sustainable software is something that can be continually developed and improved"*. Surprisingly, this is at odds with the literature as there's no reference to anything to do with this. There were also other answers that weren't covered in the literature such as the idea that sustainable software was software that is reproducible. P8 once again summed it up by saying *"Is it open, accessible, reproducible? These are things that make software sustainable".* Software that is accessible was another common answer in the data. Accessibility is something else that appears to be absent from the literature.

Good documentation, reusability and portability were all things mentioned by a single interviewee and what they felt defined sustainable software. Again, there was nothing in the literature on these things although portability is defined in ISO/IEC 25010:2011.

Much of the data is at odds with what is in the literature and although it does make a mention of extending software and also maintainability, much of the data, such as reproducibility, software with the ability to improve, reusability, portability and good documentation are all absent from the literature.

When asked if sustainability of their software is an important aspect of their working practice, there was a near consensus here, with almost all the interviewees agreeing. This tells us that sustainability is a high priority amongst RSEs. However, there were many reasons given for why it was important.

One of the common answers was a lack of funds/resources meant that sustainability was important. As research software is normally funded from grants and these grants could run out in the future. If the software isn't sustainable, then it would be very difficult to run it on future architectures. Because of this, software needs to be sustainable as there's simply no funding to rewrite large software suites from scratch.

P1 was honest in their assessment, while acknowledging that the sustainability of the software is an important aspect, they admitted that once they've come to the end of a project, they seldom return to it, simply because they're not being paid for this and there's more current issues to be dealt with. This ties in with the lack of funds/resources mentioned above and suggests that although the interviewees do think sustainability of their software is an important aspect, it is very much driven by the funding and when that dries up, the software is forgotten about. This highlights the importance of sustainable software because if the software is sustainable, it can be run in the future, long after the funding has run out.

When asked if software sustainability is a software quality, there was a mixed response, with some agreement, some disagreement and a number of interviewees remaining neutral. The lack of consensus here shows how diverse the idea is and suggests that software sustainability should be considered as a software quality but there are other factors to consider as well, such as high performance code, which they acknowledged has nothing to do with sustainability and that sustainable software is not necessarily of high quality. Others feel that there is a strong correlation between software sustainability and software quality.

The neutrality saw the suggestion that community was important, as it needed to support sustainable software. Another interviewee listed community as a software quality. It was also suggested that funding was another software quality.

The sole disagreement saw an interviewee state that it wasn't a software quality; "*Because that is dynamic and it depends, like as soon as you have something that has dependencies, you can't guarantee that it's going to be sustainable without someone keeping it alive without maintenance, basically*".

The lack of consensus continued when the interviewees were prompted on the core software qualities of sustainable software.

Good documentation was a common answer but is not referenced in the literature, though it has been mentioned by an interviewee in reference to defining sustainable software.

Maintainability was another common answer which is in agreement with Carver et al. (2021), who recognised the need for developers to maintain software in order to keep it up to date with technology and user needs.

Interestingly, two of the interviewees felt that community was a core software quality of sustainable software. This agrees with Hong and Voss (2008), who came to the conclusion that to be properly sustained, digital objects must evolve and to maintain this evolution in usage, community input is necessary.

There were also a number of qualities that were mentioned by specific interviewees only such as having a robust build system, easy installation, continuous integration, extensibility, funding, portability, packaging, performance, version control and having a basic test suite were others that received a mention.

When asked to what extent they agree on what software qualities are the foundation of sustainable software; functional suitability, usability and reliability are rated quite highly. This tells us that these are the most important software qualities according to the RSEs and they are in agreement. However, performance efficiency, security, maintainability and portability have varying answers, providing no consensus. This tells us that there is no agreement amongst the RSEs on these qualities. This shows the differing opinions amongst the RSEs on what software qualities are the foundation of sustainable software.

We found that the majority of interviewees don't measure the sustainability of their software, with the most common answer being a lack of knowhow. Lack of specific metrics and methods

also had some agreement between a couple of the interviewees. This highlights the need for some set metrics and methods to guide the RSEs through such a process.

There were also other specific answers such as a lack of time, the fact that it had never occurred to the interviewee that they should measure the sustainability and one interviewee's involvement is early on in the development so they see this as someone else's problem.

Much of the literature made reference to software sustainability being split into four dimensions; economic, social, environmental and technical. But this wasn't reflected in the answers, with none of the interviewees referring to software sustainability in this way. In fact, there wasn't much data that was mentioned by interviewees that was referenced in the literature. The only common themes identified in the literature seemed to be maintainability, community and extending the use of software.

Sustainability Design

From the interviews, we observed varying answers from the RSEs but despite this, there is some agreement amongst some of the responses to the questions.

When pressed on ecosystems being in decline and sustainability being a reasonable approach to address this issue, there was much agreement amongst the RSEs. Although there were varying opinions on how much they agreed with this notion, no one disagreed with this. The majority of interviewees agreed that sustainability is systemic but there were some disagreements with this as well, meaning it wasn't conclusive. There was also overwhelming agreement that sustainability has multiple dimensions, transcends multiple disciplines, requires action on multiple levels and applies to both a system and its wider contexts. The interviewees also agreed that system visibility is a necessary precondition and enabler for sustainability design and that sustainability requires long-term thinking.

There were some questions which saw varying answers, such as the issue of sustainability and if it was the responsibility of everyone during their whole lives. There was a lot of agreement with the statement but there was a small minority that tended to disagree with it. There were also differing answers when asked if sustainability is a concern independent of the system, with some completely agreeing and others not. One question that saw varied answers was whether acting as a sustainable practitioner means both reducing footprints and increasing handprints.

All but one interviewee acknowledged that they are currently integrating sustainable practices into their work and this shows us that RSEs are moving in the right direction. Future work could see us extending this work and asking them to share these sustainable practices with us.

Interestingly, none of the interviewees were aware of the Karlskrona Manifesto for Sustainability Design. Although they were unaware that these were the principles of the Manifesto, the majority were in agreement with much of it.

Section 7: Summary and Conclusions

This research investigated the knowledge of Research Software Engineers (RSEs) to gain an understanding of their views and experiences with software sustainability, software development lifecycles, software engineering education and training, software quality and sustainability design. We aimed to investigate how the research software engineering community understands and measures software sustainability.

Our interviews highlighted the fact that many RSEs have little to no formal training in software engineering and much of their training has been informal or learnt on the job. This suggests that there isn't much of an emphasis on formal training within software engineering and much of the expertise and experience has come from learning on the job. It seems that there isn't much emphasis on formal certifications or formal training in order to become an RSE.

We have identified that although much of the literature on software sustainability talks about the capacity to endure and to "preserve the function of a system over an extended period of time", it was seldom mentioned by the interviewees. The literature has also identified four major dimensions of sustainability: economic, social, environmental and social, but these weren't mentioned by the interviewees either.

This difference suggests that there is a marked difference between academia and on the job learning and knowledge. In conclusion, there appears to be a huge gap between the literature and what RSEs see as their reality. The results of the interview tell us that RSEs don't really agree on very much and they lack knowledge of basic software engineering concepts and principles. We have found a fundamental gap between basic software engineering practice and RSE activity. This gap needs to be closed in a drive towards achieving sustainable software. One of the things that we suggest is the creation of some set metrics and methods to guide the RSEs through the process of testing the sustainability of software.

There doesn't appear to be much of a plan and there's no cohesion between the RSEs that we interviewed, each one seems to go their own way. We feel it would be beneficial for RSEs to follow a set of practices. There's nothing that stands out with RSEs that allows us to identify them easily. It seems that anyone who is doing research and writing software for it is given the name of Research Software Engineer, regardless of their background.

Although they all conduct software testing in their day-to-day roles, the RSEs have little knowledge of software testing other than unit testing and integration testing, they have no knowledge of industry recognised types of testing such as error-based, coverage-based and fault-based testing. Testing code bases requires more than just unit testing, it needs to be tested from many different perspectives in order for it to be sustainable. Although they can give various definitions of sustainable software, they don't really show much interest in software sustainability. This ties in with Calero et al. (2013) who observed that while sustainability is a standardised practice in a number of engineering disciplines there is currently no such awareness within the software engineering community. This is similar to the observations of Amsel et al. (2011).

Future work and directions

With the lack of awareness of the Karlskrona Manifesto for Sustainability Design, this suggests that there could be a drive to raise awareness of the Manifesto and perhaps provide some education and training in it for RSEs.

We are calling for more formal software testing for the RSE role, as currently their practices fall well short of the defined software engineering principles and practices. This fundamental gap between software engineering practice and RSE activity could be addressed by more formal education for the RSEs.

We also believe that this study could be enhanced by gathering more quantitative data and speaking to a wider variety of RSEs to improve the results and data that we have already collected.

References

Ackroyd, K., Kinder, S. H., Mant, G. R., Miller, M. C., Ramsdale, C. A., and Stephenson, P. C. (2008). Scientific Software Development at a Research Facility. IEEE Software, 44-51.

Ahmed, B. H., Peck Lee, S., and Ting Su, M. (2020). The Effects of Static Analysis for Dynamic Software Updating: An Exploratory Study. 8, 35161-35171.

Amsel, N., Ibrahim, Z., Malik, A., and Tomlinson, B. (2011). Toward sustainable software engineering: NIER track. 2011 33rd International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 1, 976-979. doi: 10.1145/1985793.1985964

Asahara, A. (2020). Dynamic Analysis vs. Static Analysis. https://www.sleeek.io/blog/dynamic-analysis-vs-static-analysis

ASQ. (2020). What Is Software Quality? ASQ. https://asq.org/quality-resources/software-quality

Baker, C. Review of D.D. McCracken's "Digital Computer Programming". Mathematical Tables and Other Aids to Computation 11, 60 (Oct. 1957), 298-305.

Ball, T. (1999). The concept of dynamic analysis. ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, 216-234.

Barber, D. (2012). Why Test-driven Development? http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/

Bauer, V., Penzenstadler, B., Calero, C., and; Franch, X. (2012). Sustainability in Software Engineering: A Systematic Literature Review.

Beck, K. (1999). Extreme Programming Explained. Addison-Wesley.

Beck, K. (2002). Test Driven Development: By Example. Addison-Wesley Professional.

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., and Venters, C. (2015). Sustainability Design and Software: The Karlskrona Manifesto. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 467-476.

Bellairs, R. Perforce. (2020). What Is Static Analysis? Static Code Analysis Overview. https://www.perforce.com/blog/sca/what-static-analysis

Booch, G. (2018). The History of Software Engineering. IEEE Software, 35, 108-114.

Brundtland, G. H. and UN World Commission on Environment and Development (1987), Our common future. Oxford University Press.

Budgen, D., Kitchenham, B., Charters, S., Turner, M., Brereton, P. and Linjkman, S. (2007), Preliminary results of a study of the completeness and clarity of structured abstracts, in 'Proc. of the 11th Int. Conf. on Evaluation and Assessment in Software Engineering 2007', pp. 64– 72.

Bulldiser, (2016). History of Software Testing. Asphalt Panthers. http://www.asphaltpanthers.com/2016/02/16/history-of-software-testing-2/

Calero, C., Bertoa, M., and Angeles Moraga, M. (2013). A systematic literature review for software sustainability measures. 2013 2nd International Workshop on Green and Sustainable Software (GREENS), San Francisco, CA, USA, 1, 46-53. doi: 10.1109/GREENS.2013.6606421

Carver, J., Cosden, I., Hill, C., Gesing, S., and Katz, D. (2021). Sustaining Research Software via Research Software Engineers and Professional Associations. 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS), 23-24.

Cassman, M. (22-29 December 2005). Barriers to progress in systems biology. Nature, 438, 1079.

Chen, T. Y., Cheung, S. C., and Yiu, S. M. (1998). Metamorphic testing: a new approach for generating next test cases. The Hong Kong University of Science and Technology, Hong Kong.

Codoid. (2019). The Basics of Software Quality Attributes. Codoid. https://codoid.com/the-basics-of-software-quality-attributes/

Condori-Fernandez, N., and Lago, P. (2018). Characterizing the contribution of quality requirements to software sustainability. Journal of Systems and Software, 137, 289-305.

Condori-Fernandez, N., and Lago, P. (2018). Characterizing the contribution of quality requirements to software sustainability. The Journal of Systems and Software, 137(1), 289-205.

Condori-Fernandez, N., Lago, P., Luaces, M., and Catala, A. (2019). A Nichesourcing Framework applied to Software Sustainability Requirements

Condori-Fernandez, N., Procaccianti, G., and Ali, N. (2014). Metrics for Green and Sustainable Software. 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement.

Cook, H. R., Cox, M. G., Dainton, M. P., and Harris, P. M. (1998). A methodology for testing spreadsheets and other packages used in metrology.

Cox, M. G., and Harris, P. M. (1999). Design and use of reference data sets for testing scientific software. Analytica Chimica Acta, 380, 339-351.

Crouch, S., Chue Hong, N., Hettrick, S., Jackson, M., Pawlik, A., Sufi, S., Carr, L., De Roure, D., Goble, C., and Parsons, M. (2013). The Software Sustainability Institute: Changing Research Software Attitudes and Practices. Computing in Science and Engineering, 74-80.

Deek, F. P., McHugh, J. A. M. and Eljabiri, O. M., Strategic Software Engineering: An Interdisciplinary Approach. Auerbach Publications, 2005.

DuPaul, N. (2013). Static Testing vs. Dynamic Testing. VeraCode. https://www.veracode.com/blog/secure-development/static-testing-vs-dynamic-testing

Edwards, A. (2020). *The AI revolution: Software testing's new best friend*. Techbeacon. https://techbeacon.com/app-dev-testing/ai-revolution-software-testings-new-best-friend

Ernst, M. D. (2003). *Static and dynamic analysis: synergy and duality*

Fingent. (2018). Quality Assurance in Software Testing – Past, Present and Future. Fingent. https://www.fingent.com/blog/quality-assurance-in-software-testing-past-present-future/#:~:text=The%20first%20test%20team%20was,also%20introduced%20during%20this%20time.

Functionize. (2021). Top AI Trends in Testing for 2021. Functionize. https://www.functionize.com/blog/top-ai-trends-in-testing-for-2021/

Garnage, TA. (2017). Behavior Driven Development (BDD) and Software Testing in Agile Environments. https://medium.com/agile-vision/behavior-driven-development-bdd-software-testing-in-agile-environments-d5327c0f9e2d

Gelperin, D., and Hetzel, B. (1988). The growth of software testing. Communications of the ACM, 687-695.

Gesing, S., Heiland, R., Marru, S., Pierce, M., Maron, N., Zentner, M., Casavan, J., Vorvoreanu, M., and Mullinix, N. (2017). Science Gateways Incubator: Software Sustainability Meets Community Needs. 2017 IEEE 13th International Conference on eScience, 477-485.

Ghahrai, A. (2017). Static Analysis Vs Dynamic Analysis in Software Testing. DevQA. https://devqa.io/static-analysis-vs-dynamic-analysis-software-testing/

Gillis, A.S. WhatIs.com. (2018). Static analysis (static code analysis). WhatIs. https://whatis.techtarget.com/definition/static-analysis-static-code-analysis

Glass, R. L. (2004). Matching methodology to problem domain. Commun. ACM, 47(5), 19-21.

Goble, C. (2014). Better Software, Better Research. *IEEE Internet Computing, 18*(5), 4-8.

Gomes, I., Morgardo, P., Gomes, T., and Moreira, R. (2009). An overview on the Static Code Analysis approach in Software Development [Thesis, Faculdade de Engenharia da Universidade do Porto].

Groher, I., & Weinreich, R. (2017). An Interview Study on Sustainability Concerns in Software Development Projects. *Euromicro Conference on Software Engineering and Advanced Applications 2017*, Vienna, Austria, 350-358. https://doi.org/10.1109/SEAA.2017.70

Hannay, J. E., Langtangen, H. P., MacLeod, C., Pfahl, D., Singer, J., Wilson, G. (2009). How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, BC, Canada, 1-8.

Hannula, E. (2016). A short history of software quality. https://www.linkedin.com/pulse/short-history-software-quality-esko-hannula/

Hicken, A. Parasoft. (2018). What Is Static Analysis? Code Analysis Tools. https://www.parasoft.com/blog/how-does-static-analysis-prevent-defects-and-accelerate-delivery/

Hilty, L. Lohmann, W. and Huang, E. (2011). “Sustainability and ICT — an overview of the field,” in Proceedings of the EnviroInfo 2011.

Hong, N. and Voss, A. (2008). Why Good Software Sometimes Dies – And How to Save It. 2008 IEEE Fourth International Conference on eScience, Indianapolis, IN, USA, 1, 475-477. doi: 10.1109/eScience.2008.80

Hook, D., and Kelly, D. (2009). Testing for Trustworthiness in Scientific Software. ICSE '09 Workshop, Vancouver, Canada, 59-64.

Huynh, F. (2020). Software quality: Origin story. Coders Kitchen. https://www.coderskitchen.com/software-quality-origin-story/

Jackson, W. (2009). Static vs. dynamic code analysis: advantages and disadvantages. GCN. https://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx

Kan, Stephen H.: Metrics and Models in Software Quality Engineering. Pearson Education, Inc., Boston (2003)

Kanewala, U., and Bieman, J. (2014). Testing Scientific Software: A Systematic Literature Review.

Kanewala, U., and Bieman, J. (2018). Testing Scientific Software: A Systematic Literature Review.

Kelly, D., Thorsteinson, S., and Hook, D. (2011). Scientific Software Testing: Analysis with Four Dimensions. IEEE Software, 84-90.

King, T. (2020). The Current State and Future Trends of AI in Software Testing. Perfecto. https://www.perfecto.io/blog/ai-in-software-testing

Kitchenham, B. A., Dyba, T. and Jorgensen, M. (2004), Evidence-based software engineering, in 'Proc. of the 26th Int. Conf. on Software Engineering (ICSE 2006)', IEEE Computer Society, pp. 273–281.

Koteska, B., Mishev, A., and Pejov, L. (2015). Scientific Software Testing: A Practical Example. Proceedings of the 4th Workshop of Software Quality, Analysis, Monitoring, Improvement, and ˇ Applications, Maribor, Slovenia.

Kreyman, K., Parnas, D. L., and Qiao, S. (1999). Inspection procedures for critical programs that model physical phenomena.

Lago, P. (2019). Architecture Design Decision Maps for Software Sustainability. 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), Montreal, QC, Canada, Canada, 1, 61-64.

Lago, P., Akinli Kocak, S., Crnkovic, I., and Penzenstadler, B. (2015). Framing Sustainability as a Property of Software Quality. Communications of the ACM, 58 (10), 70-78.

Lami, G. and Buglione, L. (2012). Measuring Software Sustainability from a Process-centric Perspective. 2012 Joint Conference of the 22nd International Workshop on Software

Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, Assisi, Italy, 1, 53-59.

Larus, J.R. and Ball, T (1994). Rewriting executable files to measure program behavior. Software-Practice and Experience, 24(2):197-218.

Lewis, W. E. (2017). Software Testing and Continuous Quality Improvement (3rd ed.). CRC Press.

Lin, X., Simon, M., and Niu, N. (2018). Hierarchical Metamorphic Relations for Testing Scientific Software. 2018 ACM/IEEE International Workshop on Software Engineering for Science, 1-8.

Lin, X., Simon, M., and Niu, N. (2021). Software Testing Goes Serverless: Creating and Invoking Metamorphic Functions. IEEE Software, 61-67.

Loynton, S., Sloan, D., Burel, J., & Macauley, C. (2009). Towards a project community approach to academic scientific software development. *2009 5th IEEE International Conference on E-Science Workshops*, Oxford, UK.

Mayring, P. (2000). Qualitative Content Analysis. Forum Qualitative Sozialforschung/Forum: Qualitative Social Research, 1(2).

Merrill, P. (2019). Top trends: 5 ways AI will change software testing. TechBeacon. https://techbeacon.com/app-dev-testing/5-ways-ai-will-change-software-testing

Milewicz, R., & Rodeghero, P. (2019). Towards Usability as a First-Class Quality of HPC Scientific Software. *International Conference on Software Engineering 2019*, Montreal, Quebec, Canada.

Monteith, D. T., Stoddard, J. L., Evans, C. D., & de Wit, H. A. (2007). Dissolved organic carbon trends resulting from changes in atmospheric deposition chemistry. *Nature, 450*(7169), 537-540.

Mori, A. (2020). Anomaly Analyses to Guide Software Testing Activity. 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 427-429. https://doi.org/10.1109/ICST46399.2020.00055

Morris, C., and Segal, J. (2009). Some challenges facing scientific software developers: the case of molecular biology. In International Conference on e-Science (eScience), Oxford, UK, 216-222.

Nachtigall, M., Nguyen Quang Do, L., and Bodden, E. (2019). Explaining Static Analysis - A Perspective. 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), San Diego, CA, USA, 29-32.

Nguyen-Hoan, L., Flint, S., and Sankaranarayana, R. (2010). A Survey of Scientific Software Development. ESEM'10, Bolzano-Bozen, Italy.

North, D. (2006). Introducing BDD. https://dannorth.net/introducing-bdd/
Oettinger, A. (1966), "President's Letter to the ACM Membership," Comm. ACM, vol. 9, no. 8.

Oxford English Dictionary Online (2003), 2nd edition, http://www.oed.com/.

Parker, R. (2020). Pros and Cons of Static Code Analysis. https://software-testing.dudaone.com/pros-and-cons-of-static-code-analysis

Penzenstadler, B and Femmer, H (2013). Towards a Definition of Sustainability in and for Software Engineering. SAC'13: Proceedings of the 28th Annual Symposium on Applied Computing.

Penzenstadler, B., Raturi, A., Richardson, D., and Tomlinson. (2014). Safety, Security, Now Sustainability: The Nonfunctional Requirement for the 21st Century. IEEE Software, 14 (0740-7459), 40-47.

Philipe, O. (2018). What do we know about RSEs? Results from our international surveys. Software Sustainability Institute. https://www.software.ac.uk/blog/2018-03-12-what-do-we-know-about-rses-results-our-international-surveys

Roher, K. and Richardson, D. (2013). A Proposed Recommender System for Eliciting Software Sustainability Requirements. 2013 2nd International Workshop on User Evaluations for Software Engineering Researchers (USER), San Francisco, CA, USA, 1, 16-19.

Rosado de Souza, M., Haines, R., Vigo, M., & , C. J. a. y. (2019). What makes research software sustainable?: an interview study with research software engineers. *CHASE '19: Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, Montréal, Canada, 135-138. https://doi.org/https://doi.org/10.1109/CHASE.2019.00039

Sanders, R., and Kelly, D. (2008). Dealing with Risk in Scientific Software Development. IEEE Software, 21-28.

Sanders, R., Kelly, D., July 2008. The challenge of testing scientific software. In: Proceedings Conference for the Association for Software Testing (CAST). Toronto, pp. 30–36.

Saxe, J G (1963). The blind men and the elephant. McGraw-Hill.

Seacord, R., Elm, J., Goethert, W., Lewis, G., Plakosh, D., Robert, J., Lindvall, M. (2003). Measuring Software Sustainability. In Proceedings of the International Conference on Software Maintenance.

Segal, J., and Morris, C. (2008). Developing Scientific Software. IEEE Software, 18-20.

Sommerville, I (2004) Software engineering, 7th ed. Addison Wesley.

Tainter, J. A. (2006) “Social complexity and sustainability,” Journal of Ecological Complexity, no. 3, pp. 91–103.

Taweel, A., Delaney, B., and Zhao, L. (2009). Knowledge Management in Distributed Scientific Software Development. 2009 Fourth IEEE International Conference on Global Software Engineering, 299-300.

Thoppil, R. (2018). Quality Assurance in Software Testing – Past, Present and Future. Fingent. https://www.fingent.com/blog/quality-assurance-in-software-testing-past-present-future/

Tozzi, C. (2016). *Quality Assurance and Software Testing: A Brief History*. SauceLabs. https://saucelabs.com/blog/quality-assurance-and-software-testing-a-brief-history

Trofimov, G. Parasoft. (2021). 2021 Guide to Behavior Driven Development (BDD) Testing Frameworks. https://www.parasoft.com/blog/bdd-testing-framework-guide/

Turing, Alan (1950), "Computing Machinery and Intelligence", Mind LIX (236): 433–460, doi:10.1093/mind/LIX.236.433, ISSN 0026-4423.

Ullah, S. (2019). A brief history of software testing. https://salsadigital.com.au/insights/a-brief-history-of-software-testing

Upadhyay, R.K. (2020). Types of Static Analysis Methods. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-static-analysis-methods/

van Vliet, H. (2007). Software Engineering: Principles and Practice. Wiley.

Venters, C. C., Jay, C., Lau, L. M. S., Griffiths, M. K., Holmes, V., Ward, R. R., Austin, J., Dibsdale, C. E., and Xu, J. (2014). Software Sustainability: The Modern Tower of Babel. CEUR Workshop Proc, 1216, 7-12.

Venters, C. C., Kocak, S. A., Betz, S., Brooks, I., Capilla, R., Chitchyan, R., Duboc, L., Heldal, R., Moreira, A., Oyedeji, S., Penzenstadler, B., Porras, J., and Seyff, N. (2021). Software Sustainability: Beyond the Tower of Babel. 2021 IEEE/ACM International Workshop on Body of Knowledge for Software Sustainability (BoKSS), 3-4.

Venters, C., Seyff, N., Becker, C., Betz, S., Chitchyan, R., Duboc, L., McIntyre, D., & Penzenstadler, B. (2017). Characterising Sustainability Requirements: A New Species, Red Herring, or Just an Odd Fish?. *39th International Conference on Software Engineering ICSE 17*, Buenos Aires, Argentina.

Venters, C.C., Lau, L., Griffiths, M.K., Holmes, V., Ward, R.R., Jay, C., Dibsdale, C.E. and Xu, J., 2014. The Blind Men and the Elephant: Towards an Empirical Evaluation Framework for Software Sustainability. Journal of Open Research Software, 2(1), p.e8. DOI: http://doi.org/10.5334/jors.ao

Wiese, I., Polato, I., and Pinto, G. (2020). Naming the Pain in Developing Scientific Software. IEEE Software, 75-82.

Winkler, T. (2018). Human Values as the Basis for Sustainable Software Development. 2018 IEEE International Symposium on Technology and Society (ISTAS), Washington DC, DC, USA, 1, 37-42.

Winters, T. (2018). Non-Atomic Refactoring and Software Sustainability. 2018 ACM/IEEE 2nd International Workshop on API Usage and Evolution, 2-5.

Appendix 1

A Study in Research Software Engineering Practice Interview Structure

Background

1. What is the subject area of your first degree?
2. What is your highest education level? (Bachelor, Masters, MPhil, PhD, other please state)
a. What was the subject area of your highest degree?
3. What is your official job title?
4. What is the main role and focus you undertake in your current job?
5. How many years have you worked in your current role? 0-1, 2-3, 4-5, 6-7, 8- 9, 10+
6. What domain do you primarily develop software for?
7. How many years of experience do you have in software engineering and development? 0-1, 2-3, 4-5, 6-7, 8- 9, 10+

Software Engineering Education and Training

1. Do you have formal training in software engineering and development? Yes/No
a. If yes, what level of formal training have you received in software engineering, e.g. undergraduate, postgraduate, non-certified courses (external and internal), certified courses etc?
b. If no, do you have informal training in software engineering and development, i.e. self-taught?
2. Do you feel that this training has helped you in your current role?
a. If yes, to what extent?
b. If no, why?
3. **On a scale of 1-6, with 1 being None at all and 6 being A Lot;** How much training in the following areas has there been in your formal education or professional development? Please answer:
   - Software requirements
   - Software design

- Software architecture
- Software coding
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering models and methods
- Software quality
- Software engineering professional practice
- Software engineering economics

4) Have you attended a Software Carpentry course?
If yes, which course(s)

5) Are you certified in a specialist area of software engineering such as requirements engineering, software architecture or software testing? Yes/No, If yes, What?

Software Development Lifecycles

1. On a typical project, which software development lifecycle model do you generally adopt?
   What influenced the adoption of a particular software development lifecycle model?
   Do you feel that this is the best suited to the projects that you undertake? (Why or why *not*?)

2. Out of 100%, approximately what percentage of your time do you spend on the following activities on a typical project:
   - Requirements elicitation
   - Documenting requirements
   - Software architecture
   - Software design
   - Coding
   - Software testing
   - Usability testing
   - Project management
   - Other, please specify.

Software Testing

Do you carry out software testing?
If yes, to what extent do you plan?
What type of software testing do you implement?
Have you had experience with the following types of testing?

1. Coverage-based testing. (Covering all or most possible scenarios)
2. Fault-based testing. (Exploiting frequently occurring faults)
3. Error-based testing. (Purposely trying to break the system by doing things they suspect will break the system, e.g. by entering wrong data or clicking the wrong buttons etc.)

If yes, what is the extent of your experience with each type of testing?

1. Coverage-based testing.
2. Fault-based testing.
3. Error-based testing.

Which, in your experience, is the most reliable type of testing? Why?
What are the challenges around software testing?

Software Quality

1. Do you consider software quality, i.e. non-functional requirements, in the development of software?
a. If yes, what qualities?
b. If no, why not?
2. Are you aware of ISO/IEC 9126 Yes or No?
3. Are you aware of ISO/IEC 25010:2011 Yes or No?
4. **On a scale of 1-6, with 1 being Not Important and 6 being Very Important;**
How important do you rate the following software qualities in the development of your code?

- Functional suitability: the degree to which a product or system provides functions that meet the stated or implicit requirements when used under specific conditions.
- Performance Efficiency: the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
- Compatibility. Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment.
- Usability - "A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users."
- Reliability - "A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time."

- Security: Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
- Maintainability: A set of attributes that bear on the effort needed to make specified modifications."
- Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another."

These definitions are taken from ISO/IEC 25010.

1. Do you test the quality of your software? Yes/No

a. If yes, how?

i. Do you use any specific software metrics?

b. If no, what are your reasons for this?

Software Sustainability

1. From your perspective, what is sustainable software?

2. Is sustainability of the software you develop an important aspect of your working practice?

a. If yes, why?

b. If no, why?

3. Software sustainability has been defined in the literature as a software quality, i.e. first-class, composite non-functional requirement. To what extent do you agree with this view?

4. From your perspective, what are the core software qualities of sustainable software?

5. **On a scale of 1-6, with 1 being DON'T AGREE and 6 being COMPLETELY AGREE;** To what extent do you agree that the following software qualities are the foundation of sustainable software?

- Extensibility: a measure of the software's ability to be extended and the level of effort required to implement the extension.
- Interoperability: the effort required to couple software systems together.
- Maintainability: the effort required to locate and fix an error in operational software.
- Portability: the effort required to port software from one hardware platform or software environment to another.
- Reusability: the extent to which software can be reused in other applications.

- Scalability: the extent to which software can accommodate horizontal or vertical growth.
- Usability: the extent to which a product can be used by specified users to achieve specific goals with effectiveness, efficiency, and satisfaction in a specified context of use.

(Venters et. al, 2014).

6. Do you measure the sustainability of your software? Yes/No
   a. If yes, how?
      i. Do you use any specific software metrics?
   b. If no, what are your reasons for this?

Sustainability Design

1. **On a scale of 1-6 with 1 being DON'T AGREE and 6 being COMPLETELY AGREE**; To what extent do you agree with the following statements:
   - Ecosystems are under stress and declining, and this is affecting human conditions and futures.
   - Sustainability is a reasonable approach to addressing this decline.
   - Sustainability is systemic.
   - Sustainability has multiple dimensions.
   - Sustainability transcends multiple disciplines.
   - Sustainability is a concern independent of the purpose of the system.
   - Sustainability applies to both a system and its wider contexts.
   - Sustainability requires action on multiple levels.
   - System visibility is a necessary precondition and enabler for sustainability design.
   - Sustainability requires long-term thinking.
   - Sustainability is the responsibility of everyone, during their whole lives, including at work
   - Acting as a sustainable practitioner means both reducing my footprint and increasing my handprint. i.e **reducing harm** i.e **actions towards sustainability**?
   - I am currently integrating sustainable practices into my work i.e. acting as a sustainable practitioner
   - Are you aware of the Karlskrona manifesto for Sustainability Design? Yes/No

The statements are all taken from the Karlskrona Manifesto for Sustainability Design (Becker et. al, 2015).