Assisting scientist programmers in software engineering and development

MSc thesis Computing Science

Engin Kırmızıyüz

S1013839

Supervisor Computing Science: dr. D.G.F. Strüber

Supervisor Radboud Radio Lab: dr. S.T. Timmer

November 2, 2022

Radboud University

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my wife, Serdıl, as she continues to support me through thick and thin and has always encouraged and motivated me to challenge myself and keep on going.

I am very grateful to my supervisors, dr. Strüber and dr. Timmer, for their continued support and endless patience. Their kindness and critical thinking have been inspiring and have provided me more insight into my abilities as a researcher instead of a student.

This journey would not have been possible without PR^4 and I would like to express my deepest gratitude to each and every past and current member. I am also very thankful to the astrophysicists of Radboud Radio Lab, as their efforts provided me much insight into what difficulties they experience when developing software.

Last, but very much not least, I would like to thank my brother and my parents. My brother, who has always supported me and has given me much advice with his never ending wisdom. My mother, who provided us with her endless motherly love. My father, who came to the Netherlands as a young adolescent, as one of many children of immigrants, who worked years and years to support his wife and children, who did his best to encourage his sons to study as much as they could, hoping they would not face the same difficulties he did.

Therefore, I would like to dedicate this thesis to him, my father.

Abstract

This thesis examines common problems and difficulties experienced by scientist programmers and aims to find methods to assist scientist programmers in software engineering and development. The literature illustrates a wide spectrum of problems and difficulties. In particular, the most prevalent difficulties which persist to this day concern requirements engineering and maintaining. Additionally, the literature describes how the mindset of scientist programmers differs to those of software developers, resulting in the former seldom applying software practices.

By means of a questionnaire amongst the Astrophysicists of Radboud University Nijmegen and interviews with a team of Astrophysics and Mathematics students this thesis illustrates that the problems concerning requirements engineering and maintaining persist. In addition, by means of interviews and observations, this thesis evaluates the effect of several methods from the literature on a team of Astrophysics and Mathematics students, in an attempt to aid them in resolving the problems they experienced.

The results show that a session with developers and domain experts where the requirements are made explicit has a positive impact on the productivity but is not sufficient on its own. Deriving tasks from the requirements has proven more crucial in the increased productivity. Furthermore, the results on the impact of pair programming were inconclusive. However, creating specific user stories, which are clear and concise descriptive tasks, has on multiple occasions had a significant positive impact on the productivity of the student software team.

This thesis therefore concludes that frequently creating user stories and concrete tasks very likely have a significant positive impact on the productivity of scientists programmers, and thereby can assist scientist programmers in software engineering and development.

Contents

1 Introduction

The Radboud Radio Lab and the Electronic Systems group of Eindhoven University of Technology have started a joint student rocketry program. The initial project concerned the Payload for Radiation-measurement and Radio-interferometry in Rockets (PR^3). One goal of the project was to use radio-interferometry to track moving objects. The European Space Agency's REXUS program has given students across Europe the opportunity to add their payload to a sub-orbital rocket. The first rendition of the PR^3 payload was successfully launched on a REXUS rocket. However, while the launch was successful, some components and ground stations failed during the experiment, resulting in the lack of redundancy in the gathered data. This meant that the quality of the data could not be assured.

For the second rendition of this program, PR^4, the software team (based in Nijmegen) aims to build a modular data analysis and simulation framework for the radio-interferometry part of the project. However, this task is complicated by the fact that most members of the software team are astrophysicists or mathematicians with little to no professional experience in software engineering and development.

Related work on scientist programmer productivity and experienced difficulties show that scientists experience many difficulties which impede their productivity and motivation. Several surveys, namely those by Wiese et al.[2] and Nguyen-Hoan et al.[11], show that a large portion of difficulties concern requirements engineering and maintaining. Additionally, Nguyen-Hoan et al. and Sanders and Kelly[12] illustrate that scientists focus more on showing that their theory is correct instead of their software, meaning testing their software is mostly neglected.

Moreover, concerning the mindset of scientist programmers, Kelly and Sanders[13] and Nguyen-Hoan et al. observe a tendency to view correctness as the single most important thing in software, as described by their interviews and survey respectively. Additionally, Sanders and Kelly[13], and Segal[14] describe how software is not as important as the scientific models. The latter even mentions that software development is seen as *"a very secondary activity to their main work"*.

To combat these difficulties, Wilson et al.[3] describe twenty-four best practices to aid in a structured way of working. In addition, Sletholt et al.[10] conducted a literature review of agile practices and their effects in scientific software development and concluded that there is a positive impact on testing and requirements activities.

However, the literature also shows that scientific programmers experience that most software best practices are *"not addressing their needs"*[5]. They believe most solutions to be too domain-independent, which leaves the scientific programmer to brush them aside. I believe it is very likely that the problems and difficulties experienced in scientific software development are the result of this.

To provide further insight into the problems and difficulties, this thesis aims to analyse the difficulties experienced by scientists programmers in the literature and to build further on the literature by analysing difficulties experienced by Astrophysicists and Mathematics and Astrophysics students affiliated with Radboud University Nijmegen and the PR^4 project.

In addition, this thesis aims to combine the best practices described by Wilson et al. and the agile practices described by Sletholt et al. to aid scientist programmers when engineering and developing software, and to determine the effect of several practices from both studies on the productivity of a team of Mathematics and Astrophysics students, with little to no prior experience in professional software development.

The following research question has been formulated:

"How can scientist programmers be assisted in software engineering and development?".

To answer the research question, the following sub-questions have been formulated:

- RQ_1 What are common problems/difficulties scientist programmers experience when writing software?
- RQ_2 Which methods exist in the literature to combat these difficulties?
- RQ_3 What is the effect of using these methods on the productivity of Mathematics and Astrophysics students?

Section 2 discusses existing literature on the difficulties scientist programmers experience when writing software and on known methods to combat these difficulties. This section also discusses how the literature on both topics correlates with each other, and the justification for this research.

The PR^4 project and the software team are discussed in Section 3. Section 4 discusses the research methodology. This includes a detailed description of the Action Research methodology[1], and how it has been applied to this research.

Section 5 describes the results of this research, divided per sub-question. Then, Section 6 discusses the results and the implication of the results and their relation to the existing literature. Finally, Section 7 concludes the thesis.

2 Literature review

In this section, the known literature related to scientific programming and difficulties experienced by scientist programmers will be discussed. Scientific programming in this context relates to the act of software development by scientist without a background in computer or software science.

2.1 Common problems in scientific software development

Concerning the pain in developing scientific software, Wiese et al.[2] and Nguyen-Hoan et al.[11] conducted extensive surveys amongst scientist programmers. Wiese et al. give great insight in the problems reported, which are categorized in three main categories, each with their sub-categories: technical problems (70.8%), social problems (23.9%), and scientific-related problems (5.3%).

In the next paragraphs, the difficulties concerning requirements and testing are described. In addition to the notable size of the difficulties in the literature, these two subjects were chosen based on initial observations of the PR^4 software team and the initial interviews, which are described in Section 4, where these subjects were also prevalent.

2.1.1 Requirements

An interesting statistic is the notable size of the reported problems relating software requirements and management (23.2% of the technical problems) and 'communication and collaboration' (19.8% of social problems). This could signal that there is a lack of software engineering practices amongst the scientific programmers, where the two metrics could concern making requirements explicit and maintaining these requirements.

Changing requirements. Wiese et al. mention that the respondents attribute these problems to the volatile and evolutionary nature of functional requirements, citing *"the objective changes after having worked on it for months"*. Nguyen-Hoan et al. notice, amongst a multitude of characteristics, that requirements documentation is the least commonly produced type of documentation. Reasons given by their respondents against documentation include the abundance of time and effort required and *"requirements constantly changing or not specified up front."*

Up-front requirements. In addition to the observations made by Nguyen-Hoan et al., Sanders and Kelly [12] noticed in their research that none of their interviewees created an up-front formal requirements specification, stating that they wrote it when the software was almost complete, only if regulations mandated a requirements document.

Amongst scientific programmers, requirements engineering is deemed very difficult, especially since requirements specifications mostly cannot be specified up-front [3, 5]. Segal and Morris even state that full up-front requirement

specifications are impossible [6], which is reflected in the observations of Nguyen-Hoan et al. and Sanders and Kelly. Perhaps the ever evolving nature of scientific software in its respective research could go hand in hand with agile software practices.

2.1.2 Testing

Nguyen-Hoan et al. moreover found that concerning the types of testing, verification against specified requirements and design are the least common. They also state that verification and peer review were amongst the least performed testing and verification activities. Sanders and Kelly [12] mention that scientists use testing to show that their theory is correct, instead of whether their software works or not. They mention that some of their interviewees, whose testing was unsystematic, seemed unconcerned nevertheless, possibly due to their focus on theory. Additionally, Sanders and Kelly [12, 13], and Segal and Morris [6] note that there is a problem concerning verifying whether a test is successful. They mention that testing requires data to compare against test results, which can be very limited (or not available). In addition, they mention that in the case of a mismatch between the data and test results, the problem might be caused by a set of factors; either the theory, its implementation, the input data, or the verification data might cause the problem.

Sanders and Kelly name different techniques; e.g. creating a regression test suite and the organization of version control, and they mention how these techniques seem to be missing in scientific software development. Nguyen-Hoan et al. support this claim by mentioning the apparent lack of *"bug/change tracking or testing tools"*. Segal takes this one step further and observed that there is a *"lack of any disciplined testing procedure."*[14]

2.2 Relating the problems to software practices

Wiese et al., in resolving the illustrated pains, refer to Wilson et al.[3], who describe twenty-four best practices to aid scientists in writing maintainable code and to improve scientist programmer efficiency. In a more recent paper, Wilson et al.[4] name *"good enough practices"*, as an easier stepping stone to software practices for scientists which had less computational responsibilities. Most of the best practices incline towards a structured way of working; *"Write programs for people, not computers"*, *"Make incremental changes"*, *"Don't repeat yourself"*, and *"Collaborate"*.

While these practices could prove extremely useful in aiding the aforementioned pains, applying them unfortunately is not as straightforward. Milewicz and Rodeghero describe in their position paper the essence of usability in scientific software [7] and how it is mostly ignored according to Ahmed et al. [8]. Kelly solidifies this by discussing a *"software chasm"* between scientific programmers and software engineers [9], signalling that the scientific programmers are not applying the current solutions offered by software engineers, mostly due to them being too domain-independent.

An example Kelly gives of the software chasm is that of an overwhelmingly negative response to a required software engineering course for third-year electrical- and computer-engineers. Kelly names two major causes; the first being that the students do not acknowledge that there is a high probability that software will be part of their jobs. The second major cause is described to be the curriculum itself, which fails to offer solutions geared towards the type of work the students might undertake. This is a recurring subject in the research done by Faulk et al.[5], where they describe that scientific programmers overwhelmingly favor handcrafted solutions because *"the computer scientists don't address our needs"*. They signal that this causes the scientific programmers to become isolated from the software engineers' help. In their research, they have also noticed that *"issues important to software engineers (e.g. maintainable code, robust programming languages and practices) receive almost no attention from scientific programmers"*, which correlates with the findings of Milewicz and Rodeghero and those of Kelly. Faulk et al. also name several skill sets they deem as essential: domain science, scientific programming, scaling, and management, crucially noting that these are only useful when they synchronize through communication and collaboration. This might signal that there is a tremendous need for certain practices which could increase amongst others communication and collaboration within a development team. As mentioned before, perhaps agile software practices could prove extremely useful.

Sletholt et al. have conducted a literature review of agile practices and their effects in scientific software development [10]. Here, they focus specifically on *"the impact on testing and requirements activities in projects with agile practices"*. They concluded that their findings show that projects using agile practices indeed have a better handling of testing and requirements activities.

Therefore, in answer to RQ_2, the literature provides ample methods in favor of resolving the problems reported by the literature. For example, the best and good enough practices reported by Wilson et al.[3, 4] and the agile practices reported by Sletholt et al.[10] encompass a broad range of solutions.

2.3 Possible misalignment of mindsets

Up until this point, the literature clearly illustrated numerous practical difficulties impeding the scientist's productivity in software development.

However, from the results of the survey conducted by Nguyen-Hoan et al.[11] and the findings in existing literature, the assumption arises whether there is a deeper rooted, less easily noticed problem; namely the misalignment of the mindset of scientific programmers versus the mindset of software engineers.

2.3.1 Correctness above all

Kelly and Sanders [13] interestingly, having interviewed scientists who develop or use scientific software, observed the following; *"If the software does not return correct answers, then it is useless. Being on-time, under budget, maintainable, or highly usable all take a backseat to correctness."* This can also be seen in the results of the survey Nguyen-Hoan et al.[11] conducted amongst scientific software developers. Here, their respondents were asked to rate non-functional requirements on a Likert scale. The results confirm the importance of correctness, since all respondents adamantly voted Reliability (summarised by Nguyen-Hoan et al. as the ability to perform with correct, consistent results) to be either important or very important, resulting in it to be the only non-functional requirement to receive a 100% importance rating. Kelly and Sanders even observed that scientists prefer the system completely crashing, rather than it possibly giving false results.

2.3.2 The importance of software

The previous observation might signal how scientists perceive the software they develop, as Kelly and Sanders note the following: *"Scientists see the software code as an inseparable entity from their models. They assess the models. They normally do not assess the software as a separate thing that needs attention."* [13]

Interestingly, an observation made by Segal [14] describes that software development is perceived as *"a very secondary activity to their main work"*, where some interviewees mentioned they see developing software as *"just a tool"*. Segal strikingly observed the lack of recognition of the skills and knowledge required to develop software, where she among others quotes a line manager; *"everybody knows how to do [software development](...) It's assumed that everybody knows what to do."* [14]

Possibly, this view of software development as just a tool can be linked to the low importance of the bottom ranked three non-functional requirements as reported by Nguyen-Hoan et al. These are Reusability with 62% (described as *"the ability to be used in multiple applications"*), Traceability with 54% (described as *"the ability to link the knowledge used to create the application through to the code and the output"*), and Portability with 52% (described as *"the ability to be easily modified for a new software/computing environment"*). Nguyen-Hoan et al. namely mention that scientific developers do not anticipate reusing parts of their system [11].

2.4 This thesis

What stands out from the literature is that while there are best practices to aid not only scientists, but any software developer, scientific programmers nevertheless experience that these are *"not addressing their needs"*[5]. The scientific programmers perceive most software engineering solutions to be too domain-independent, resulting in these solutions being brushed aside, or in extreme cases, completely ignored. I believe that most of the reported problems and difficulties experienced by scientific software development are very likely the result of the isolation of scientific programmers.

To chip away at this firmly rooted tree of scientific software isolation, this thesis aims to combine the best practices described by Wilson et al. and the agile practices described by Sletholt et al. and to determine the impact of both studies on the productivity of a group of student scientific programmers. By means of the Action Research methodology [1], which is further described in Section 4, diagnosing a specific problem before each cycle and planning actions accordingly, this could possibly result in different practices being more effective than others. This thesis could then contribute to the ongoing research on assisting scientist programmers in software engineering and development, possibly having an effect on future academic courses aimed towards scientists who are highly likely to develop scientific software.

3 Project and software team

This section describes the PR^4 project context and the software team. Here, the broader context of the project is outlined, followed by the task the software team has had at hand. Furthermore, the software team and the software process and structure is described.

3.1 Project

The PR^4 project (Payload for Radiation and Radio-Interferometry on Rockets Revisited) is a joint student project of TU/e and the Radboud Radio Lab which focuses on designing and building a payload for rockets. The payload consists of two experiments; one for radio-interferometry, and one for radiation. The latter aims to *"characterize the arrival direction distribution of cosmic rays at altitudes between 20 and 80 km"*[15], which the team believes might support existing monitoring stations and cosmic-ray satellites with additional information. To achieve this, the team designed a CubeSat cosmic-ray detector which scans over the Earth in orbit.[15]

The former focuses on radio-interferometry to track moving objects. To do this, a payload on the moving object sends out a radio signal at a constant frequency. On the ground, separated by hundreds of meters, different ground stations are placed, each with three antennas to receive the radio signal. This is then transformed to phase data. The exact coordinates of the moving object should then be able to be computed using one or multiple reconstruction algorithms.

The software team has been tasked with developing a modular data analysis and simulation framework. The framework was supposed to consist of three layers, each with their own responsibilities. The first layer concerned the phase data, which is either simulated, replayed (from field experiments), or live data from the ground stations. The second layer included the reconstruction algorithms. The final layer consisted of the graphical user interface, which was intended for the configuration of the simulations and for the graphical representation of, among others, received data, coordinates, and trajectory.

What makes this project especially difficult is the lack of software engineering experience within the software team. As has been made clear from the literature review, starting a project and making the requirements clear without being able to plan too far ahead has been a prevalent difficulty for scientist programmers. This challenge, therefore, has been paramount for the software team, as designing each previously described layer and their respective requirements were the first things on their list of duties.

3.2 Software team

The software team has consisted entirely of students from Radboud University Nijmegen and Eindhoven University of Technology. The size of the team has been very variable. Since the project is an extracurricular activity for most students, there have been new students joining and old students leaving due to, most predominantly, a shift in their schedule. The number of active software team members has fluctuated between two and five, with the later stages of the team consisting of three members.

All but one student were either Mathematics or Astrophysics students, whereas the remaining student studied Artificial Intelligence.

The software team engaged in weekly meetings to keep each other up to date on the progress and to manage any ongoing or new problems in development. In the beginning of this thesis project, these meetings were part of a grander general meeting, which also included the organization of the field experiments and other organizational matters concerning the project.

Because of this structure, the software meetings were either very short, or not held at all. This gave incentive to organize a separate weekly software meeting, which were much more detailed and which gave the software team members leverage on their current and upcoming tasks.

This structure fits in very well with the research methodology of this thesis project, which is described in the next section.

4 Methodology

In this section, the research methodology is described. The first part outlines the Action Research methodology [1] and how this methodology was used in this project. The next section describes the initial interviews and the subsequent questionnaire study. The Action Research methodology also includes certain types of interventions to take, in order to tackle a specific problem. The subsequent section describes these interventions.

4.1 Action Research

The research method used in this project was based on the Action Research methodology as described by Staron [1]. This methodology allows us to diagnose problems by cycling through the following five phases:

1. **Diagnosing**. This focuses on the researcher *"collecting opinions and symptoms which they need to explore in order to decide which challenge to address during the action research cycle"*.

2. **Action planning**. Here, the researcher and the practitioners (in this case the software team, which was described in Section 3) plan what actions to take and how to take them. In this phase, planning which data is collected and how this should be done is also determined.

3. **Action taking**. The planned actions are executed and its effect is observed.

4. **Evaluation**. The collected data is statistically analysed. This analysis should show whether the actions taken do indeed solve the diagnosed problem. If this is not the case, either the researcher collects additional data, or re-diagnoses the problem based on the collected data.

5. **Learning**. The results of the cycle are specified as *"practical guidelines for the involved organizations and contexts and as theory-building for the research community"*.

In order to refine the research questions, we needed a deeper understanding of the problems the software team was facing. Therefore, a method was needed to diagnose these problems, to take steps to remedy or solve these problems, and to evaluate these steps.

Staron [1] describes how action research is *"a method for co-development of research results, where academia and industry can work together."* Staron also argues that this results in both parties learning from each other and developing results which contribute to both industry and academia.

The iterative nature of the methodology itself was deemed fitting for the software project, as this gave the opportunity to map action research cycles to different phases of the software project and to clarify, extend, and modify

the research question and sub questions of this thesis project. At the same time, the actions taken during each cycle would benefit the software team in overcoming their problems in software engineering and development. This was done by experimenting with different software practices and by evaluating the previous cycle and bringing to light any shortcomings during the cycle.

4.1.1 Phases and cycles within project

This project consisted of three stages, each with one or two cycles of variable length. Each cycle contained the phases described above. However, each phase was not strictly followed according to the points described. For example, during evaluation, no statistical analysis was carried out, and the impact of the action was determined by means of observation and acknowledgement of the software team members.

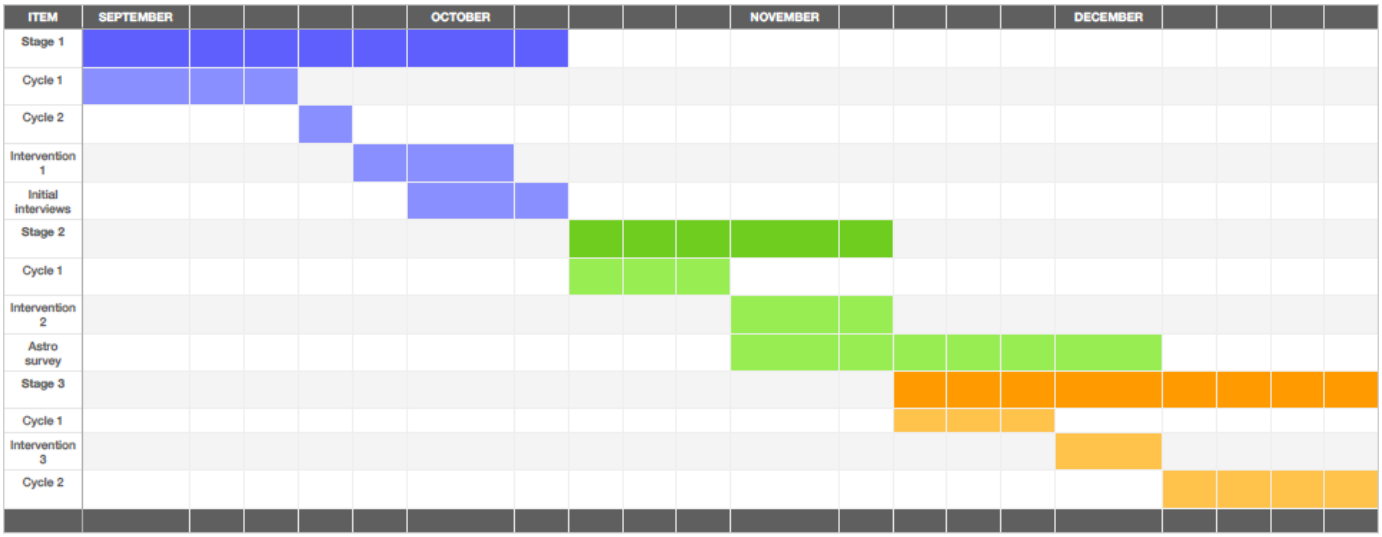


Figure 1: Stages and cycles within project, September-December 2020

In figure 1, the stages and cycles within this project are illustrated. The stages were defined by the state of the software team and the intervention of each stage during the project. For example, the first stage ended with an intervention with regard to requirements engineering and the initial interviews.

The interventions are described in detail in Section 4.3.
Stage 1 consisted mostly of exploring the project and known literature to determine the research questions, hence two short cycles, where the second cycle revised the diagnosis of the first one.

Following the first intervention and the initial interviews, further literature review was conducted in stage 2, combined with the preparation and execution of the questionnaire study. Distinguishing this stage from the first was the state of the software team, as some members were leaving the project, this resulted in the productivity of the team reaching near zero. Therefore, the lone cycle of this stage focused on diagnosing and resolving this problem, combined with the tasks described above.

The questionnaire accepted answers far into the final stage and in combination with its first cycle provided an excellent basis for the final intervention, which concerned the scrum methodology. The second and last cycle focused on working with the software team using scrum.

To determine the impact of each intervention, final interviews were conducted amongst the team members. The details of these interviews follow in the next subsection.

4.2 Data collection

Data collection was planned and carried out in several different ways. First, the literature review and the initial surveying of the state of affairs within the software team are discussed. Subsequently, the initial interviews are described. Finally, the resulting questionnaire and its details are presented.

4.2.1 Literature review

Initial surveying and literature review was carried out to get acquainted with the project and to define the initial problem definition of this thesis.

The literature review also served as a basis for RQ_1 and as crucial to RQ_2, namely by exploring existing difficulties experienced by scientist programmers and by describing existing methods to combat said difficulties respectively. Additionally, the literature review granted more insight into defining a questionnaire for further data collection, described in Section 4.2.3.

Literature review was persistent throughout the first two stages of the project and has provided solid bases for the initial interviews, the questionnaire, and the first and third interventions.

For the initial interviews, the questionnaire, and the first intervention, findings from the literature shifted the focus primarily on requirements engineering, as it was evident that a significant portion of scientist programmers experienced difficulties on this subject. This was also noticed in the initial surveying in the first cycle, as observations showed that the software team struggled with engineering and maintaining requirements, leading to the first intervention.

As for the third intervention, literature review on existing methods to combat the experienced difficulties was fundamental in its preparation. Sletholt et al. [10] found that agile practices have a positive impact on testing and requirements activities and the best practices given by Wilson et al. [3] were deemed fitting within the agile methodology. This provided ample basis for the intervention, which is described in Section 4.3.3.

4.2.2 Initial interviews

The initial interviews were held to get acquainted with the project and to gain insight into the difficulties the software team experienced at the time. Additionally, this granted more insight into finding similarities between the current difficulties and those laid out in the literature, further building upon answering RQ_1.

Four interviews were conducted. Two interviewees were software team members at the time, and the other two were domain experts.

All interviewees were asked five open questions:

1. Could you describe the term software development and your view on it in your own words?
2. Can you name the most profound difficulties you experience when writing software?
3. What are your thoughts on the current process within the project?
4. What aspects of this project would you change if you could?
5. How (if applicable) would you like to receive help in resolving the aforementioned difficulties?

Subsequently, thematic analysis was carried out, as described by Staron [1], to recognise recurring themes and to identify their importance. As described above, this provided a basis for the questionnaire that followed.

4.2.3 Radboud University Astrophysics questionnaire

The aforementioned initial interviews and additional literature research in conjunction with initial observations in the PR^4 project laid the foundation of the questionnaire that was conducted amongst different researchers of the Astrophysics department of Radboud University Nijmegen. The aim of this questionnaire was to build upon the existing literature by analysing what difficulties the astrophysicists experience when developing software, therefore granting additional information towards answering RQ_1.

This questionnaire has had 33 respondents with zero to 40 years of professional experience in scientific software development. Of those 33 researchers, 30 signal that their software development skills have been self taught, while, with overlap, 26 signal that they acquired their skills from programming courses during their studies. When asked which programming languages have been used to develop non-trivial projects, the grand majority signal either Python or C/C++ (93.8% and 59.4% respectively).

This questionnaire also had a secondary objective, which was unrelated to this thesis. Therefore, not all questions are discussed. The questionnaire can be found in Appendix A in its entirety.

Initially, as can be seen in the characteristics of the respondents above, this questionnaire starts with exploratory questions to explore different characteristics of the respondents; e.g. how many years of experience in scientific software development, how the current software development skills were acquired, and in which language(s) non-trivial projects they have developed.

Subsequently, the respondents were given a set of statements to which they could signal their level of agreement in the form of a Likert scale, with 1 denoting 'strongly disagree', and 5 denoting 'strongly agree'. The crucial statements mostly concerned requirements engineering and overall software development

habits. Their importance resulted from the initial interviews. The results of these interviews are discussed in Section 5.1.1. The statements are as follows:

- Using third party libraries is difficult.
- Writing tests efficiently is difficult.
- I prefer using a version control system, even when working alone.
- I make sure that I can reuse my code in the future when writing software.
- I find it easy to keep track of the software's requirements.
- Making software requirements explicit is easy.
- Making software requirements explicit is worth the effort.
- I always formally document software requirements.
- I find it demoralising when requirements change over time.

Then, the respondents were given a set of open questions. These questions gave them the opportunity to voice their most profound difficulties in writing software and third party library usage:

- What parts of writing software do you find (most) difficult?
- What is most important to you when choosing a third party library to use in your software?

Finally, the respondents were given a list to choose between subjects they might want to receive help in. This list mostly contained subjects that were relevant to the aforementioned secondary objective. The crucial subjects included 'Requirements engineering' and 'Agile software development'.

4.3 Interventions

Within this project, there were three interventions, as can be seen in figure 1. The interventions were held to help combat the difficulties experienced by the PR^4 software team. The first and third interventions were related to the findings from the literature, which is described accordingly in the following descriptions. Their descriptions follow the order as seen in the figure.

4.3.1 Requirements engineering

This intervention was held during the exploratory stage of the thesis project. After initial literature review and observations of the software team, the hypothesis was that the productivity of the team had halted due to the lack of concrete software requirements. It was evident that not all members had the same vision of the software that was to be written.

After discussing this with the team, the idea was to hold a requirements engineering session with the software team members and the domain experts. This included creating a document with functional and non-functional requirements, and discussing their respective priorities using the MoSCoW method.

Because the literature review was still in a very exploratory phase, the intervention was mostly based on the intuition and previous experience of the researcher and the software team. However, from the literature at the time, specifically Wiese et al., Wilson et al., Faulk et al., and Segal and Morris [2, 3, 5, 6], it was evident that scientists experienced difficulties in up-front requirements engineering and the volatile evolutionary nature of requirements in scientific software. The team was therefore instructed not to think too far ahead and focus on taking small steps towards a first working version. They were also informed of the high probability of the requirements changing over time.

4.3.2 Acquisition of new team members

A short while after the requirements engineering session, some members of the software team mentioned that they would not have any time to spare on the project anymore, and would be leaving. This resulted in the software team consisting of only two students.

This was reason for the researcher to repetitively inquire the PR[4] project leaders and domain experts for new software team members. This was done on multiple occasions, mostly during the weekly team meetings. The researcher reasoned that, ideally, the new members should work on the project as part of a thesis or internship project to reduce the impact of courses and deadlines on their productivity. However, at that stage of the project, there was not yet a possibility of defining such thesis or internship projects. This meant that the ideal measures of recruitment were not yet possible, and had to be let loose until further notice. Therefore, recruitment focused on students of Mathematics, Astrophysics, and Computer Science. Ultimately, four new members were recruited, of which three became active software team members. All three were students of the aforementioned fields respectively.

4.3.3 Scrum training

The final intervention was inspired by the findings of Sletholt et al.[10] and by the best practices of Wiese et al.[3] as mentioned in Section 2.2. The software team was already using a very rudimentary agile structure by having weekly software meetings where the progress and where new goals were discussed. However, the new team members were not aware of any structure and felt quite lost. The idea was, therefore, to present the Scrum methodology and to link it to the current structure of the project.

This intervention included a training session with all software team members. They were first introduced to the Scrum methodology and the agile ideology. The subjects of sprints, roles, and the backlog and its user stories were presented.

The focus was mostly set on requirements refinement and user stories, as the team has had most difficulties with those subjects.

The presentation also discussed several best practices mentioned by Wilson et al.; *"Make incremental changes: Frequently ask for feedback"*, *"Don't repeat yourself: Modularize your code! Make it reusable"*, and *"Collaborate: Use pair programming; Use Git"*. In addition, concrete examples were given, e.g. on modularising code and using Git. The latter was supported by a document outlining the different applicable commands, e.g. *git commit*, *git pull*, and *git push*, outlining when to use which, how merging and merge conflicts work, and how pull requests work.

Finally, the training ended with a collaborative effort on clarifying the requirements of the first intervention and on deriving clear and small user stories from them.

4.4 Evaluation of interventions

Since this thesis project concerned qualitative research, measuring and evaluating the effects of the different interventions to answer RQ_3 was done by means of final interviews with the software team members. Six people were interviewed. Two were domain experts and the rest were active members of the software team. The interview guide can be found in Appendix B in its entirety.

The interviewees were asked several open questions concerning their opinion on and description of each phase. For example: *How would you describe the state of affairs before intervention X?*, *How would you describe the state of affairs after intervention X?*, and *How would you describe the effect of the intervention on the project?* Intermittently, to quantify the results of the interventions, the interviewees were given the following statements to rate between one and ten:

1. How productive do you rate the software team?
2. How productive do you rate yourself?
3. The software team members were aware of what task everyone had.
4. I knew exactly what to do.
5. The requirements of the software were clear.
6. I as a developer am confident in writing software.

Three of the six joined the team after the second intervention, which meant that the first phase was only relevant to the original members. Two of the interviewees were domain experts, who only had a supporting role and were not actively involved in developing the software. Therefore, the statements 2, 4, and 6 did not apply to them, as these focused on the software developer's view of themselves.

Subsequently, the interviewees were asked to give their opinions on each intervention and whether there could be any improvements for the future.

As described before, regarding the initial interviews, thematic analysis was carried out on the open questions of these final interviews. This was done to recognise any recurring themes and to identify their importance. This gave solid insight into additional items of importance, concerning the possible effects of the interventions.

The justification behind the emphasis on solely the project process of the questions and statements lies in the nature of the software project at the time. Since the software project was in a very experimental and exploratory phase, no emphasis could be put on specific topics regarding software quality and stakeholder interests.

Nevertheless, this thesis serves as an extension on the existing literature by means of theory testing. The thesis project aimed at finding the difficulties scientist programmers endure and analysing the effect of best practices and agile practices provided by the literature.

As the literature showed that there exists vast amounts of difficulties experienced by scientific software developers, this thesis builds on this by providing relevant and topical difficulties that were experienced by the PR^4 software team and the Astrophysics department of Radboud University Nijmegen. Additionally, the impact of the best practices and agile practices provided by the literature on the difficulties experienced by the PR^4 software team was determined by means of the interventions, observations, and final interviews.

5 Results

This section describes the results of the research. First, the results of the initial interviews are discussed, followed by results of the questionnaire study.

Subsequently, the project specific difficulties are illustrated. This subsection focuses primarily on the observations of the software team during the thesis project. These observations are described in chronological order, as was done in the methodology section.

Finally, the results of the interventions and their effects on the student software team are described.

5.1 Common problems and difficulties

As discussed in Section 2 and Section 4, the literature evidently shows that a significant portion of the difficulties scientist programmers experience concerns requirements engineering and maintaining. Additionally, the manner in which these findings have shaped the initial interviews and questionnaire has been made clear.

Towards answering RQ_1, the following paragraphs present the results of these interviews and the questionnaire study.

5.1.1 Initial interviews with the software team

After the initial interviews were conducted with the PR[4] software team, the most notable difficulties mentioned concerned requirements engineering and maintaining. Interviewees mentioned that requirements changing in the middle of a project was very frustrating. Another interviewee notes that his lack of software development experience restricts his abilities in contributing much to the project, mentioning: *"No one has said 'This is what you must do exactly', which leaves me empty handed, because I have no idea what to do."* The same interviewee notes: *"I also feel like I cannot contribute to requirements engineering, since I seem to lack the complete vision of the software."* This shows that there is a recurring problem in up-front requirements engineering, and the distribution of the overall vision of the software.

5.1.2 Surveying Radboud University's Astrophysics department

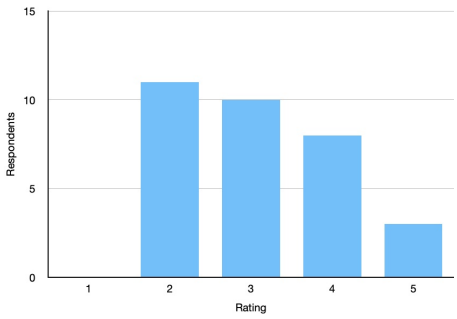
Following the initial interviews, a questionnaire was conducted amongst different researchers of the Astrophysics department of Radboud University Nijmegen, as described in Section 4.

The respondents were given a set of statements to which they could signal their level of agreement in the form of a Likert scale, with 1 denoting 'strongly disagree', and 5 denoting 'strongly agree'. As was described in Section 4, 33 responses were received.

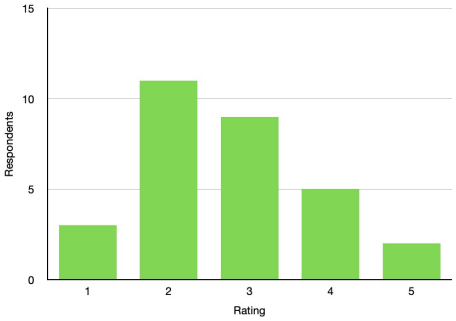
Requirements. When asked about requirements, the results seem to be very interesting. The statements and results concerning requirements can be seen in Figure 2. On whether it is easy to keep track of the software's requirements (Figure 2a), the opinions are balanced. Eleven respondents agree and the same amount disagrees.

Almost a half of the respondents (fourteen) disagree on the notion that making software requirements explicit is easy (Figure 2b). This shows that requirements engineering is still deemed difficult. The results also show that requirements changing over time is experienced as demoralising (Figure 2e), as fifteen respondents agreed with that statement. This correlates with findings in the literature, as problems reported by Wiese et al. were attributed to the volatile and evolutionary nature of functional requirements [2].

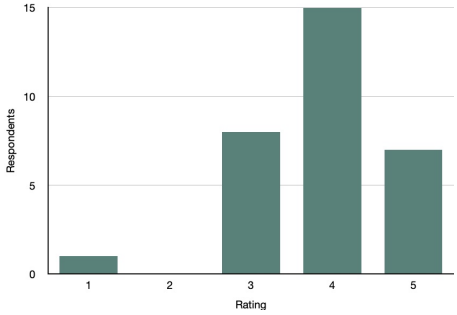
Interestingly, a majority of the respondents, 22 out of 33, agree that making requirements explicit is worth the effort (Figure 2c), while only seven respondents signal that they always formally document software requirements (Figure 2d). These responses seem to mostly correlate with the amount of problems reported by Wiese et al. on software requirements and management. While it is evident that requirements engineering is especially difficult in scientific software development [2, 3, 5, 6], only six respondents signal that they would like to receive help in requirements engineering.



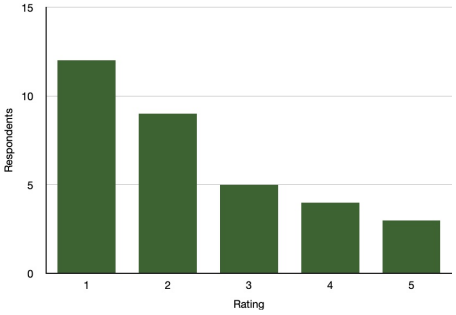
(a) I find it easy to keep track of the software's requirements.



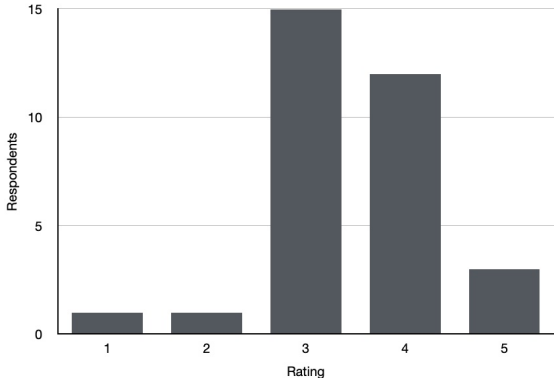
(b) Making software requirements explicit is easy.



(c) Making software requirements explicit is worth the effort.



(d) I always formally document software requirements.



(e) I find it demoralising when requirements change over time.

Figure 2: Five questions on requirements and their answers
(1 stands for 'strongly disagree', 5 stands for 'strongly agree')

Third party libraries. When asked whether using third party libraries is difficult, nine respondents disagreed and eight strongly disagreed, while ten did not agree nor disagree. The respondents were also asked what is most important to them when choosing a third party library to use in their software. While only nineteen responses were given, fifteen of them mentioned documentation, and six mentioned maintenance.

Faulk et al. reported that frequent complaints about third-party libraries and tools are that they are hard to learn and poorly supported. However, Wiese et al. reported only a tiny fraction of all problems to be related to third-party libraries [5, 2], which corresponds more with the responses mentioned above.

Starting a project/Initial design. On the question *"What parts of writing software do you find (most) difficult?"*, 26 responses were given. Most notably, ten of the responses mention that starting a project and the initial design of the software is most difficult. Interestingly, this coincides with the aforementioned difficulty of requirements engineering in scientific software development. One of the respondents even noted the following: *"The software is usually developed at the same time we understand a scientific problem. So, the most difficult part is actually understanding which kind of software we need for solving a specific problem."*. This signals that defining requirements up-front is indeed not viable and this might even signal that there could be more focus on changing the initial mindset of the scientific software developers concerning requirements over time.

About version control, the respondents do mention that they prefer using a version control system. On the statement *"I prefer using a version control system, even when working alone"*, twenty respondents either agree (six) or strongly agree (fourteen), possibly showing a positive change regarding software development habits.

Earlier, in the literature review, Nguyen-Hoan et al. mention that scientific developers do not anticipate reusing parts of their system. The respondents of the Radboud questionnaire, however, think differently. On the statement *"I make sure that I can reuse my code in the future when writing software"*, 24 respondents either agree (fourteen) or strongly agree (ten). Showing that there might be a shift in the importance of reusable software.

5.2 Project specific difficulties before interventions

In this subsection, the project specific difficulties are described. Here, the focus lies on the observations made of the software team during the thesis project, illustrating the circumstances which lead to each respective intervention. These observations are supported by the interviews described in Section 4.4, as each interviewee was asked to describe each phase in their own words.

5.2.1 Overall vision/lack of concrete requirements

The first stage of the thesis project revolved around getting acquainted with the project and the software team. An evident lack of productivity was observed, but the cause was yet to be determined. Developers were tasked with choosing relevant technologies for different parts of the software by means of experimentation. They were able to present some alternatives and their own preferences regarding the technologies. However, these preferences were not based on any specific requirements of the software, as the overall vision was not clear to all members of the software team.

The interviewees solidify the lack of productivity, as one interviewee mentioned: *"We were not concretely working towards measurable goals. We were just doing something."* In support of the software team, one of the domain experts mentioned: *"We were at the brink of starting the project with a new team. They just did not know what to do at the time."*

One of the developers noted: *"In that period, there was a long time where I had no clear task. (...) We did not get much done. (...) Our team was undermanned, and we had no idea what we were doing."*

From these observations, the hypothesis has been that the productivity of the software team had halted due to a lack of concrete software requirements. As mentioned above, there were no clear goals, and the team had no idea what to do. Therefore, the decision was that there should be an intervention concerning requirements engineering.

5.2.2 Available time

Remarkable unforeseen difficulties about the productivity of the software team that have been observed concerned the amount of time each developer had to spare on the project.

Since this has been an extracurricular student project since its conception, the students working on it have done so predominantly on a voluntary basis. This meant that the productivity came to a grinding halt as soon as exam periods or miscellaneous deadlines approached. For some developers, finishing the semester meant leaving the project altogether, resulting in the already undermanned software team consisting of only two students at one point. Discussions on the subject resulted in more effort being shifted to recruiting new team members.

5.2.3 Lack of unanimous vision/lack of concrete requirements II

After new team members were acquired, they were similarly tasked with experimenting with different libraries and developing small proofs of concept.

When evaluating the progress of the experiments during one of the weekly meetings, one of the new software team members explicitly noted its uncertainty on how a specific component should exactly look, since the vision of the initial members has not been made entirely explicit. The developers started indicating that they found the existing documentation on the requirements difficult to understand and difficult to translate into *"bite-sized chunks"* for them to pick up. This showed that a one-time requirements session is not sufficient.

The software team members solidify the observations. One developer states: *"When I first started, the task that was given to me was a bit vague. (...) I think we had a lot of different tools that we were using, so it was a bit unclear which one I should look at. I think that affected our productivity; It was so open that we had to decide what to do and that made it vague and unclear."*

Another developer's view is in line with the previous: *"I didn't have the idea that there were any deadlines. I don't think anyone had, which meant none of us really had an idea what we were doing exactly. That negatively impacted my productivity. I didn't have the idea we were working towards something together. That was the biggest drawback."*

While the other interviewees share the same experiences, there was also a positive note. A third developer notes: *"The first weeks were mainly to get acquainted with the project. I believe the time we took was really needed to start up well. Even though we were not writing software, us getting acquainted should still be considered productive."*

Concerning the software development method of the team, the structure observed had similarities with an agile method, like Scrum. For example, there were weekly software meetings where the team discussed the progress of each developer, eventual blockades/pitfalls, and where new goals were set for the next meeting.

However, it was presumed that the team was not aware of any structured method, and that the team was not aware of what they were achieving as a collective.

5.3 Summarising experienced problems

To answer RQ_1, we must first acknowledge that there is much correlation between the literature and the parties observed during this thesis project. Therefore, in addition to the scientist programmers in the literature, astrophysicists and astrophysics and mathematics students affiliated with Radboud University and the PR^4 project respectively experience similar problems.

Most notably, difficulties concerning requirement engineering and refinement seem to persist, most likely due to the evolutionary nature of scientific software development.

The astrophysicists seem to acknowledge the importance of making requirements explicit, but also agree on the demoralising nature of volatile requirements.

Very little respondents agreed on whether using third party libraries is difficult, which is in line with the literature.

Starting a project and the initial design of it seems to be difficult for several astrophysicists. The same was observed with the PR^4 software team, as they had difficulties with grasping the overall vision and with the requirements of the project, which seemed to be the most prevalent problems.

5.4 Interventions and their effects

This subsection describes the reasoning behind the interventions and the effect the interventions have had on the software team. Where the previous subsection described the circumstances and the context of the team which led to the interventions, this subsection describes the motivation behind each intervention and the effect it has had on the software team, therefore providing an answer to RQ_3.

5.4.1 Requirements engineering

This intervention took place in the first stage of the thesis project. At the time, only exploratory literature review was carried out. The software team lacked a unanimous vision of the software and there was no clear list of software requirements, which meant that these problems were to be tackled first. It was duly noted that these requirements were to be preliminary, as the literature clearly shows that up-front requirements in scientific software development is impossible.

Effect. As mentioned in Section 4.4, only three of the six interviewees were present at this stage of the project. Only one interviewee was a developer at the time, as the other two were domain experts with solely a supporting role.

Thematic analysis has shown recurring themes in the opinions of the three interviewees. The main items concerned the slow but positive effect of the intervention on the productivity of the team, which was attributed to initial persistent uncertainty amongst the team.

The interviewees mentioned that it took some time before the team picked up the pace, as not everyone was aware of what to do next. One domain expert noted: *"There was a wait-and-see attitude. (...) I think people needed something else, in addition to the requirements, namely a feeling for how it is all going to work. Understanding the essence (of the software) took a while."*

The eventual increase in productivity is attributed to the tasks that were derived from the requirements. One interviewee noted: *"There were very big blocks of things that had to be done, which in turn had to be split up in small blocks. To me, it was pleasant to have one simple thing to work on."*

Another domain expert noted: *"Initially, not everyone quite knew what they wanted and what they could work on. Shortly after, everyone had concrete things they could work on and they had enough knowledge to do so. That moment was a turning point where productivity greatly increased."*

The interviewees were also asked to rate statements describing different aspects before and each intervention, as was described in Section 4.4. The results can be seen in Table 1 and Table 2 respectively.

Interviewee	Team Productivity	Individual Productivity	Team Task Awareness	Individual Awareness	Requirement Clarity	Developer Confidence
Domain Expert 1	1	-	1	-	1	-
Domain Expert 2	5	-	1	-	1	-
Developer 1	2	1	1	1	1	1

Table 1: Statements before intervention.
Ratings are from 1 (lowest) to 10 (highest)

Interviewee	Team Productivity	Individual Productivity	Team Task Awareness	Individual Awareness	Requirement Clarity	Developer Confidence
Domain Expert 1	1 (+0)	-	6 (+5)	-	8 (+7)	-
Domain Expert 2	7 (+2)	-	10 (+9)	-	8 (+7)	-
Developer 1	3 (+1)	5 (+4)	1 (+0)	5 (+4)	4 (+3)	4 (+3)

Table 2: Statements after intervention.
Ratings are from 1 (lowest) to 10 (highest)

The results show some discrepancies between the domain experts and the developer. For example, Domain Expert 2 signals that the productivity increased very much, but noted that it took a while until it did so. The others agree on the same, but have given a low rating as they referenced the short term after the intervention. The results also show an increase in requirements clarity and team task awareness. However, the developer believes there was no increase in general awareness, noting: *"We were already a small team and I was starting to become alone, as the rest of the developers were phasing out"*.

Concerning RQ_3, several conclusions can be drawn based on these results. The requirements have served as a good basis for creating a clearer vision of the software that was to be developed.

They were, however, not sufficient on their own, as it was clear that it were the tasks derived from the requirements which served as the turning point in productivity.

The emphasis should therefore not be solely on requirements engineering, but on deriving said small tasks from the requirements. This enables the team to obtain a better understanding of the software, while providing them tasks to further shape and mold the software through experimentation.

5.4.2 New team members

As described previously, remarkable difficulties concerned the amount of time each developer had to spare on the project. Eventually, the software team was very undermanned and was in dire need of additional developers as little to no tasks were getting done. After repeated inquiries at the PR[4] project leaders, four students were recruited, of which three became active developers.

Effect. All six interviewees were present at this stage of the project. This means that four developers and two domain experts were interviewed about this intervention.

The new developers' view on their first period with the project has many similarities with the previous interviews. They experienced a lot of uncertainty and were not aware of much structure in the development methodology. Some even showed concern about themselves, which they attributed to the amount of freedom and openness they were given on getting acquainted with the project. As mentioned before, one of the developers noted: *"It was so open that we had to decide what to do and that made it vague and unclear."*

To help get the developers acquainted with the project, the researcher and domain experts discussed whether pair programming would have a positive effect. This discussion stemmed from the best practices described by Wilson et al.[3] This was difficult to put into practice, as the developers' schedules did not always align well enough. Eventually, a more rudimentary approach was taken, as pairs of developers were given the same task to work on together.

There has not been any conclusive evidence on whether it had a positive effect on the productivity of the team and on them getting acquainted with the project. The domain experts mention that the new developers were quick to understand the project and pick up tasks, but none attribute this to pairs working on the same task.

In contrast, one developer has had a very negative experience; *"I found it very difficult to work on the same task with my teammate. I could not keep up with his pace, which resulted in my falling behind very much. Ultimately, this was one of the reasons for me leaving the software team altogether, as I did not feel useful anymore."* This shows that working in pairs should be done with caution, as situations where one is outpaced by another can have detrimental effect on the motivation and productivity of developers.

The interviewees were again asked to rate the previously mentioned statements. For the first three interviewees, Table 2 serves as the status quo before this intervention and Table 3 as after this intervention.

Interviewee	Team Productivity	Individual Productivity	Team Task Awareness	Individual Awareness	Requirement Clarity	Developer Confidence
Domain Expert 1	1 (+0)	-	6 (+0)	-	8 (+0)	-
Domain Expert 2	5 (-2)	-	10 (+0)	-	8 (+0)	-
Developer 1	5 (+2)	5 (+0)	5 (+4)	5 (+0)	6 (+2)	3 (-1)
Developer 2	6	6	8	6	4	4
Developer 3	5	3	6	4	5	6
Developer 4	6	7	6	7	6	7

Table 3: Statements after new developers.
Ratings are from 1 (lowest) to 10 (highest)

Table 3 shows that the new developers were quite positive about their starting period. They mentioned that starting up was difficult, but that the team was nevertheless productive after a short while.

The domain experts, despite their positive opinion on the team starting up quickly, were not yet willing to change their ratings, as they believed the new team was still starting up at the time.

Nevertheless, the intervention proved itself beneficial for the team. As more developers were acquainted with the project quickly, the overall productivity improved.

However, in answer to RQ_3, the effect of pair programming, which was influenced by the best practices illustrated by Wilson et al.[3], is inconclusive.

5.4.3 Scrum training

The initial vagueness and uncertainty was attributed to the software requirements. These were not relayed sufficiently as the new developers were tasked with exploring different libraries to use, which sparked confusion, as one developer had noted: *"We have the requirements document, but I have no idea what to do exactly."*

This evidently shows that a one-time requirements session is not sufficient. As mentioned in Section 2.2, Sletholt et al. illustrate that projects using agile practices have a better handling of testing and requirements activities. Section 4.3.3 described how the software team was already using a rudimentary agile approach by having weekly software meetings where the progress was reported and new goals were set.

Since the new developers, were not aware of any structured method or of any deadlines or goals, a presentation was given concerning the Scrum methodology. During the presentation, the team was also shown several best practices mentioned by Wilson et al., as mentioned in Section 4.3.3. Subsequently, a tutorial on Git was given and a textual version was provided for later use.

After the presentation, the present developers and domain experts collaborated on creating user stories and deriving small, concrete tasks.

Effect. All interviewees agree on the great increase of productivity amongst the team. However, most interviewees attributed this increase to the user stories and concrete tasks, instead of on the Scrum methodology itself, as one developer stated: *"I never had the idea that we were really using Scrum, so I wouldn't say the presentation was very useful. However, there was a steep increase in the team's productivity, so from that point of view I would say it was very useful. I believe the user stories and tasks were everything we needed. It provided us with a clear list of concrete tasks and that was very pleasant."*

One domain expert noted: *"I am not convinced that one must use Scrum to run a project. (...) In this case, I do believe the intervention was needed, as it stimulated the team. (...) In the future, I would focus mostly on the user stories. In hindsight that was the part that had proven itself most useful."*

One of the developers had a different opinion on the cause: *"The most important thing for the team, what mostly helped raise the productivity of the team, did not have to do with the intervention or the documentation, but was the presence of a software team leader. For me, the biggest change was that there was someone who had an idea of what everyone was doing, had experience of leading a project, and could reason with the individuals, support them and keep order."*

Concerning the best practices mentioned during the presentation, the developers did not have a strong opinion in favor of or against them. One developer mentioned that modularising the code has resulted in the code being more readable and maintainable. The others mentioned that the best practices were useful, but quite straightforward and some even trivial.

They did however have a strong opinion in favor of the Git tutorial and document. One developer mentioned: *"I didn't have a lot of experience with Git. I actually benefited a lot from the PDF you made. (...) Sometimes when you look for information it gets too complicated. It helped me a lot during my studies now."* The other developers also noted that they use the document as reference during their studies. When asked whether they believe using Git is worth the effort, all responses were strongly in favor.

Table 4 shows the statements after the Scrum intervention.

Interviewee	Team Productivity	Individual Productivity	Team Task Awareness	Individual Awareness	Requirement Clarity	Developer Confidence
Domain Expert 1	8 (+7)	-	8 (+2)	-	8 (+0)	-
Domain Expert 2	8 (+3)	-	10 (+0)	-	8 (+0)	-
Developer 1	7 (+2)	3 (-2)	7 (+2)	5 (+0)	7 (+1)	2 (-1)
Developer 2	8 (+2)	8 (+2)	8 (+0)	9 (+3)	9 (+5)	9 (+5)
Developer 3	8 (+3)	7 (+4)	8 (+2)	7 (+3)	8 (+3)	7 (+1)
Developer 4	8 (+2)	7 (+0)	9 (+3)	9 (+2)	8 (+2)	8 (+1)

Table 4: Statements rated after scrum intervention.
Ratings are from 1 (lowest) to 10 (highest)

The results show improvements almost exclusively. The developers mentioned that the user stories and tasks made everything much more clear. They noted that they were setting concrete goals, were aware of what everyone was doing, and were aware what they were involved in together.

The domain experts specified that the productivity increased due to the concreteness of the tasks, which in turn had the developers more engaged in their responsibilities.

With regards to answering RQ_3, this has been the most influential intervention of the three, as the productivity and motivation of the team steeply increased. Several important remarks were made by the developers on points that increased the productivity, such as having a figure with knowledge and experience that leads the team, and such as modularising the code to make it more readable and maintainable.

However, as mentioned, the crucial part of the intervention was the collaboration on the user stories and the tasks derived from them. Repeating what one domain expert mentioned above, much emphasis should be laid on the user stories in the future. A critical note in addition, which one developer rightfully mentioned, is the importance of revising the user stories frequently, as that makes sure all tasks can streamlined and requirements can be refined where needed.

6 Discussion

This thesis project was carried out to answer the following question: *"How can scientist programmers be assisted in software engineering and development?"*

In answer to RQ_1, the literature has shown that requirements engineering and refinement has been the most prevalent difficulty that is experienced by scientist programmers, due to the evolutionary nature of the projects. The Radboud Astrophysics questionnaire and the observations on and interviews with the PR^4 software team support this claim.

The research methodology proved itself useful, allowing this research to be refined and extended based on observations of the PR^4 software team.

Three interventions were carried out based on these observations and were supported by methods illustrated in the literature as part of RQ_2.

To answer RQ_3, the effect of the interventions were measured by observations after the interventions and interviews at the final stage of the project.

The first intervention showed that a session with developers and domain experts where the requirements were made explicit served as a good basis for creating a clearer vision of the software. It was however concluded that the requirements themselves were not sufficient on their own, as the tasks derived from the requirements were more crucial in the increased productivity.

The second intervention was very beneficial for the team, as the overall productivity improved. However, whether pair programming had a notable effect on this was inconclusive.

The most notable was the final intervention concerning Scrum. Observations determined that productivity of the team increased steeply. Thematic analysis done on interviews conducted with each software team member solidify the observations, as there were strong positive opinions concerning the team's productivity.

However, this has been attributed not to Scrum, but to the user stories and the small, concrete tasks derived from the user stories, which correlates with the findings from the first intervention.

Therefore, based on the experience of the PR^4 software team consisting of Astrophysics and Mathematics students, this thesis has shown that frequently creating user stories and concrete tasks very likely has a significant positive impact on the productivity of scientist programmers, and thereby can assist scientist programmers in software engineering and development.

This thesis project did have its limitations. As the main research question concerns scientist programmers in general, the generalisability of the results is limited by the specific fields of the respondents and the software team, as they are astrophysicists and Astrophysics and Mathematics students respectively. Nevertheless, their views and input are crucial, as their problems and difficulties illustrate that action still must be taken to assist the broader group of scientist programmers.

Another aspect to consider is the small size of the software team and the fact that only one team was observed. This leaves to speculation any additional problems and difficulties that are experienced to this day. Furthermore, the effect of the methods could be determined more reliably with more and/or larger groups.

However, as the team was variable, which meant new members joined throughout the project, the results have shown that the newer members experienced similar problems to those experienced by the older ones. At the same time, the effects of the first and third intervention, which concerned requirements engineering with the older and newer members respectively, were also very similar.

It was unforeseen that the software team members had very limited time to spare, which resulted in the developers working around each other's timescales. This meant that not all methods from RQ_2 were applicable. Still, the results show clearly whether the methods that were applicable were effective or not.

Future work should consider the positive effect of user stories and concrete tasks on the productivity of scientist programmers. In doing so, these methods could possibly result in the difficulties concerning requirements engineering and maintaining to be a considerably smaller burden for scientist programmers.

In addition, future work could further consider the effect of including these methods in the curricula of students in scientific fields.

7 Conclusion

In light of assisting scientist programmers in software engineering and development, this thesis project has illustrated common problems and difficulties experienced by scientist programmers as described by the literature, supported by a questionnaire carried out amongst the Astrophysics department of Radboud University Nijmegen, and supported by observations and interviews on the PR^4 software team consisting of Astrophysics and Mathematics students.

In addition, this thesis project has highlighted existing methods and practices to aid in resolving the problems and difficulties.

Finally, the effect of several of the methods on the PR^4 software team has been determined based on observations and interviews regarding three interventions where the methods were applied. From this, the most effective method has been to create user stories and derive tasks from them.

The results of RQ_1 show that there is significant correlation between problems and difficulties reported by the literature, those reported by the astrophysicists of Radboud University, and those observed by the researcher on the PR^4 software team. This shows that those problems and difficulties still persist amongst scientist programmers.

In particular, difficulties concerning requirement engineering and refinement continue to bug the scientist programmers. The literature illustrated how requirements frequently changing and up-front seem to be frequently reported difficulties and how it contributes to scientist programmers seldom producing requirements documentation.[2, 11]

The findings of this thesis project support this, as a significant portion of the astrophysicists signalled that requirements changing over time works demoralising and, at the same time, a small fraction signalled that they always formally document software requirements, while two thirds of the respondents agree that making requirements explicit is worth the effort.

This means that finding methods to help the scientist programmers is crucial to this day, as requirements are the foundation to each and any software project.

This thesis project described several available methods provided in the literature in answering RQ_2[3, 4, 10] and has illustrated the effect of several of these methods on a team of Astrophysics and Mathematics students who experienced similar difficulties as described in RQ_1 in answer to RQ_3.

In doing so, this thesis project concludes, in support of the findings of Sletholt et al.[10], that sessions where user stories are created and tasks are derived have a significant positive impact on scientist programmer productivity, thereby aiding scientist programmers in difficulties concerning requirements engineering and maintaining. It must be noted that this should not be a one-time session, but a recurring event throughout the project.

References

[1] Staron, M. (2020). *Action Research in Software Engineering.* Springer, Cham. doi: https://doi.org/10.1007/978-3-030-32610-4_2

[2] Wiese, I., Polato, I., & Pinto, G. (2020). *Naming the Pain in Developing Scientific Software* in IEEE Software, vol. 37, no. 4, (pp. 75-82), July-Aug. 2020, doi: https://doi.org/10.1109/MS.2019.2899838

[3] Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., et al. (2014) *Best Practices for Scientific Computing.* PLOS Biology 12(1): e1001745., doi: https://doi.org/10.1371/journal.pbio.1001745

[4] Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T.K. (2017) *Good enough practices in scientific computing.* PLoS Comput Biol 13(6): e1005510., doi: https://doi.org/10.1371/journal.pcbi.1005510

[5] Faulk, S., Loh, E., Vanter, M. L. V. D., Squires, S., & Votta, L. G. (Nov.-Dec. 2009) *Scientific Computing's Productivity Gridlock: How Software Engineering Can Help* in Computing in Science & Engineering, vol. 11, no. 6, (pp. 30-39), doi: https://doi.org/10.1109/MCSE.2009.205.

[6] Segal, J., & Morris, C. (July-Aug. 2008) *Developing Scientific Software* in IEEE Software, vol. 25, no. 4, (pp. 18-20), doi: https://doi.org/10.1109/MS.2008.85.

[7] Milewicz, R. & Rodeghero, P. (2019) *Position Paper: Towards Usability as a First-Class Quality of HPC Scientific Software* in 2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science), Montreal, QC, Canada, (pp. 41-42), doi: https://doi.org/10.1109/SE4Science.2019.00012.

[8] Ahmed, Z., Zeeshan, S., & Dandekar, T. (2014) *Developing sustainable software solutions for bioinformatics by the "Butterfly" paradigm.* [version 1; peer review: 2 approved with reservations]. F1000Research, doi: https://doi.org/10.12688/f1000research.3681.1.

[9] Kelly, D.F. (Nov.-Dec. 2007) *A Software Chasm: Software Engineering and Scientific Computin.* In IEEE Software, vol. 24, no. 6, (pp. 120-119), doi: https://doi.org/10.1109/MS.2007.155.

[10] Sletholt, M.T., Hannay, J., Pfahl, D., Benestad, H.C., & Langtangen, H.P. (2011, May). *A Literature Review of Agile Practices and Their Effects in Scientific Software Development.* In Proceedings of the 4th international workshop on software engineering for computational science and engineering (pp. 1-9), doi: https://doi.org/10.1145/1985782.1985784.

[11] Nguyen-Hoan, L., Flint, S., & Sankaranarayana, R. (2010, September). *A Survey of Scientific Software Development.* In Proceedings of the 2010

ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 1-10), doi: https://doi.org/10.1145/1852786.1852802.

[12] Sanders, R., & Kelly, D. (2008). *Dealing with risk in scientific software development*. IEEE software, 25(4), (pp. 21-28), doi: https://doi.org/10.1109/MS.2008.84.

[13] Kelly, D., & Sanders, R. (2008). *Assessing the quality of scientific software*. In Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering.

[14] Segal, J. (2007, September). *Some problems of professional end user developers*. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007) (pp. 111-118), doi: https://doi.org/10.1109/VLHCC.2007.17.

[15] PR4 Space - Science, Technology and Engineering. https://pr4.space. Online; accessed October 4, 2022.

A Radboud University Astrophysics Questionnaire

Astrophysics questionnaire

Your experience in scientific software development

The goal of this questionnaire is to gain insight in the difficulties scientist programmers endure when writing software. Additionally, with the results, the aim is to find possible methods to assist scientist programmers in writing readable, reusable, and modular software.

What is your age?

_____

What is your gender?

- ○ Male
- ○ Female
- ○ Other: _____

For how many years have you developed software for scientific purposes?

_____

How many hours per day do you write software for scientific purposes?

- ○ 0 to 1 hour
- ○ 1 to 2 hours
- ○ 2 to 3 hours
- ○ 3 to 4 hours
- ○ 4+ hours

How many hours per day do you write software for fun?

- ○ 0 to 1 hour
- ○ 1 to 2 hours
- ○ 2 to 3 hours
- ○ 3 to 4 hours
- ○ 4+ hours

How experienced do you consider yourself in writing software?

Very inexperienced *1* *2* *3* *4* *5* *Very experienced*

1

How did you acquire your current software development skills?
Multiple answers possible

- ○ Programming courses during studies
- ○ Programming courses during school
- ○ Programming courses in free time
- ○ Professional training
- ○ Self taught
- ○ Other: _____

In which language(s) have you developed non-trivial projects in the past?
Multiple answers possible

- ○ Python
- ○ Java
- ○ R
- ○ C/C++
- ○ Haskell
- ○ JavaScript
- ○ HTML/CSS
- ○ Other: _____

2

Difficulties in software development

Please rate the following expressions from 1 being ‘strongly disagree’ to 5 being ‘strongly agree’. If you do not know the answer to a question or if the question does not apply to you, please leave it empty.

Writing object-oriented code is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Using third party libraries is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Working with old code is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Implementing multi-threading is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Connecting my application to a database is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Working with memory and pointers in C/C++ is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Writing tests efficiently is difficult.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Preferences in software development

Please rate the following expressions from 1 being ‘strongly disagree’ to 5 being ‘strongly agree’. If you do not know the answer to a question or if the question does not apply to you, please leave it empty.

I prefer collaborating with colleagues on a software project to working alone.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

I prefer using a version control system, even when working alone.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

I make sure that I can reuse my code in the future when writing software.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

I find it easy to keep track of the software’s requirements.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Making software requirements explicit is easy.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Making software requirements explicit is worth the effort.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

I always formally document software requirements.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

I find it demoralising when requirements change over time.

Strongly disagree *1* *2* *3* *4* *5* *Strongly agree*

Open questions

Please answer the following open questions as thoroughly as you can.

What parts of writing software do you find (most) difficult?
E.g. requirements engineering, working in a group, working with one or more unfamiliar language(s)

_____

Did you ever struggle with learning a new language? If so, what language was that and what parts of learning that was (most) difficult?

_____

What is most important to you when choosing a third party library to use in your software?
E.g. readable manual, written in the same language

_____

Do you feel that there are generally enough libraries that meet your criteria? If not, what parts would you like to change about the available libraries?

_____

5

Your preferences in receiving help

Would you like to receive help in any of the subjects you find difficult when writing software? If so, please choose one or multiple subjects from the table below.

- ○ Object-oriented programming
- ○ Learning a new language
- ○ Advancing knowledge on a language (e.g. Advanced Python, C/C++, Java)
- ○ Multi-threading/concurrency
- ○ Working with databases
- ○ Memory and pointers in C/C++
- ○ Basics of testing
- ○ Advanced testing
- ○ How to collaborate on a project (via Git or other VCS)
- ○ Learning how to write clean code (e.g. SOLID principles)
- ○ Requirements engineering
- ○ Agile software development
- ○ Creating websites
- ○ How to work with TeX/LaTeX
- ○ Networking
- ○ How to work with the Linux terminal
- ○ Other: _____

How would you like to receive help in these subjects?

- ○ Lunch lecture (short one-time lecture)
- ○ Mini course (multiple lectures on a subject
- ○ Hands-on exercise sessions
- ○ Other: _____

Do you have any further comments on the subject of software development or this questionnaire?

_____

B Interview guide RQ_3

Monday, 17 May 2021

Interview guide RQ3

What is the effect of using these methods on scientist programmer productivity?

Context

In the context of my thesis, RQ_3 comes after a section on project-specific difficulties, namely the situation before the interventions, and after a section on the interventions. The answer to RQ_3 should detail the situation after the interventions, mostly by means of observations made, which should heavily be supported by qualitative and quantitative data acquired from the software team.

Questions

I would like to present the following questions to be asked during the interviews, along with the rationale behind them.

To gain insight into the current state of affairs and how the interviewee experiences the project currently, I would like to ask the following questions (Note that the additional questions with Roman numerals serve as a guide, and are only asked to gather further context when needed):

1. How would you describe how things are going currently?
    I. With regards to overall productivity
    II. With regards to progress made
    III. With regards to the requirements of the software

1

For the interviewees which were active in the project before the new team members joined, I would like to ask the following questions to gain insight into the state of affairs before and after the interventions concerning requirements and the new team members respectively. I would also like to discover what effect these interventions have had on them and the project.

2. How would you describe the state of affairs before the intervention on making the software requirements explicit?
3. How would you describe it after the intervention?
4. How would you describe it before the new members joined?
5. Idem after the new members joined?
6. How would you describe the effect of these interventions on the project?
    I. Idem on yourself?

To gain insight into the impact of the Scrum intervention, I would like to ask all interviewees their opinion on the training. I would also like to ask how they experienced the project before and after the intervention to be able to determine its effect.

7. How would you describe the state of affairs before the intervention?
    I. For the newer members when they first joined.
8. How would you describe it after the intervention?
9. How would you describe the effect of the intervention on the project?
    I. With regards to the working structure
    II. With regards to overall productivity
    III. With regards to the user stories
    IV. With regards to the requirements of the software
    V. With regards to the best practices mentioned
10. What is your opinion on the intervention concerning Scrum?

2

Additionally, I would like to ask similar questions concerning the short tutorial I produced on using Git to help the new developers get a feel for working together on a project with version control. Ideally, I would like to get to know what elements they found useful and not useful.

11. What is your opinion on the Git tutorial?
    I. With regards to working structure
    II. With regards to productivity
    III. With regards to clarity
    IV. With regards to ease of use
12. How would you describe the effect the tutorial had on you?
    I. Idem 13.I
    II. Idem 13.II
    III. With regards to short-term usability
    IV. With regards to long-term usability

I would also like to ask the interviewees if they missed any possible new interventions, or if they found anything lacking in the interventions themselves. This could bring to light improvements or new methods that might be useful for future work.

13. If possible, how would you like to have improved the interventions yourself?
14. What kind of intervention(s) would you have liked to have seen/experienced?

To be able to quantify the effects of the interventions, I would like to have the interviews rate several statements (in italic) and questions on a scale of one to ten (lowest to highest respectively) before and after each intervention.

1. How productive do you rate the software team?
2. How productive do you rate yourself?
3. *The software team members were aware of what task everyone had.*
4. *I knew exactly what to do.*
5. *The requirements of the software were clear.*
6. *I as a developer am confident in writing software.*

3