Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Rapid Solo Software Development (RSSD) Methodology based on Agile

Kristo Radion Purba (kr.purba@soton.ac.uk)
University of Southampton Malaysia
Rusyaizila Ramli
University of Southampton Malaysia

Research Article

Keywords: Software development, Methodology, Agile methodology, Solo software development, Software Development Life Cycle (SDLC)

Posted Date: August 26th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1985368/v1

License: 🅭 🅯 This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Abstract

Existing software development methodologies were mostly focused on team-based development. Solo development presents its own challenges due to the lack of collaboration and resources. Existing solo development methodologies lacked efficiency, modularity, and revisitability, which become the values for the proposed RSSD (Rapid Solo Software Development) methodology. RSSD adopts the same main phases from the Agile methodology (meet, plan, design, develop, test, and evaluate), but with different subphases, values, and optimization measures. The *efficiency* value focuses on streamlining the planning process, including code structure planning and optimization strategies. *Modularity* focuses on dividing the codes into smaller functions and reducing dependencies. Lastly, *revisitability* focuses on improving the clarity of the diagrams, documentation, and code structure for easy revisits. The RSSD methodology was tested on 10 total projects, and respondents agreed that it was helpful to streamline the development process, with an average score of 4.19 out of 5.0. This study contributes to making a streamlined yet effective methodology that helps solo software developers to plan a project in a more structured manner.

1. Introduction

Many methodologies have been developed for software development, but most of them are for team-based development instead of for solo developers. Solo developers include freelancers, developers in small companies, and students on final-year projects. Many solo developers prefer freelance work due to the flexibility of timing and project cost; however, self-managing the project structure is challenging (White, 2015). Students on final year projects have a similar challenge in doing the project systematically (Hassani et al., 2018). Furthermore, a survey showed that 60% of software companies started their activities with a solo developer (Pagotto et al., 2016). Thus, a practical, systematic, and rapid solo software development methodology is required.

Various solo software development methodologies have been proposed, such as Personal Extreme Programming (XP), Scrum Solo, and a more recent Secure-SSDM **(Moyo & Mnkandla, 2019).** However, they lacked important aspects, namely efficiency, modularity, and revisitability. There are methodologies for team-based development, such as Agile and Waterfall, but they have complexities and collaboration elements which are not applicable for solo development.

This paper aims to build the RSSD (Rapid Solo Software Development) methodology, with three main values, i.e., *efficient*, *modular*, and *revisitable*. The following research questions were studied, i.e., (Q1) What are the phases, subphases, values, and optimization measures of the RSSD methodology? (Q2) How does RSSD help solo developers during the development process? (Q3) What are the challenges faced by the respondents during the development using RSSD?

The rest of the paper is organized as follows; literature review on common software development methodologies, research methodology, phases and values of RSSD, evaluation, and conclusion.

2. Literature Review

This section discusses the common team-based and solo software development methodologies, including Agile, Waterfall, and their derivatives.

2.1 Agile Methodology and Its Derivatives

Agile-based Software development methodologies are rapidly expanding (Flora et al., 2014), with many derivations such as XP, Scrum, Kanban, Lean and Crystal (Kumar et al., 2019). These methodologies were focused on team-based development. Agile is also well-known for mobile software development due to its iterations and ability to adapt to challenges such as input technology, usability, user interface design, portability and security (Shaydulin & Sybrandt, 2017).

Extreme programming (XP) allows developers to work closely with customers and works best on short-period projects. Customers define their initial requirement through an index card and then see the system in progress. The Scrum methodology focuses on the empirical processes to allow the team to respond rapidly whenever should there is any change needed (Hron & Obwegeser, 2022). Scrum also applies transparency, where the changes will be seen by every team member and stakeholders. However, such transparency increases the burden on the developers.

In Kanban, the team will manage the flow of work using a visual signal to show the continuous flow of improvement and deliveries (Zayat & Senvar, 2020). However, as it relies on visual signals, there is a possibility that team members might interpret tasks wrongly. Another popular Agile methodology is Lean Development. It allows the team to move any tasks that they think are not applicable anymore as they go along the project to save time and money (Kišš & Rossi, 2018). However, this requires experienced team members, and there is a risk involved in removing items. Lastly, the Crystal methodology gives freedom for developers to develop their preferred process as they go (Chaudhari & Joshi, 2021). It focuses on members' interaction.

2.2 Waterfall Methodology

The Waterfall methodology consists of linear, non-circular phases, where each phase is taken seriously to avoid issues in the later phases. It consists of five basic phases which are requirement, design, coding, testing and operations (Bogdan-Alexandru et al., 2019). It assumes that there will be no issues throughout the development thus there is no need to return to previous phases. This is very unlikely to happen because clients usually change their requirements throughout the development even though initial agreements were made.

2.3 Solo Development Methodologies

One of the latest methodologies for solo development is Secure Solo Software Development Methodology (Secure-SSDM) (Moyo & Mnkandla, 2019), which is focused on security and is Agile-based. They also listed existing SSDMs including Freelance as a Team (FAAT), Personal Extreme Programming (PXP1 and PXP2), Go-Scrum, Scrum solo, DeSoftIn and Initial Software Development Method (MIDS) Adaptation. Those SSDMs have common phases, namely Planning, Development and Evaluation (Moyo & Mnkandla, 2019). Personal XP assumes all of the requirements are detailed during the planning and does not expect them to be changed during the project. Other SSDMs are more flexible in editing the requirements during the development. Such flexibility can be a problem for solo development due to the lack of resources.

2.4 Comparison of Methodologies

Overall, the waterfall methodology is more strict compared to the Agile-based methodologies. Agile and its derivatives are more realistic, streamlined, and flexible compared to other methodologies. Team-based methodologies have collaboration elements which are not applicable for solo projects. There are few Agile methodologies for solo development. Personal XP has a detailed planning process, and Secure-SSDM includes security aspects. However, existing solo methodologies lacked efficiency (such as code optimization and streamlined planning process), modularity (making modular codes), and revisitability (ability to revisit the codes).

3. Research Methodology

There were four phases of this research, i.e. preparation, data collection, aggregation, and finalization, as seen in Fig. 1. The data was collected from 7 (seven) expert solo software developers which are described in Table 1. In total, 10 projects were developed using RSSD.

Table 1
Respondents (Developers)

#	Country	Type of Developed Software
D1	Australia	Information systems
D2	Indonesia	Information systems
D3	Taiwan	Mobile games
D4	Malaysia	Games
D5	Indonesia	Information systems
D6	Netherland	Mobile applications
D7	Malaysia	Mobile applications

During the *preparation phase*, the RSSD was designed, and we explained the methodology to professional solo software developers in an open-ended discussion. In the *data collection phase*, the developers were asked to use RSSD in their upcoming projects, and they will record their timelines accordingly. This phase was carried out for 11 months. After that, we collected the feedback from the developers in the *aggregation phase*, which was then used to finalize the RSSD methodology.

4. Phases Of Rssd Methodology

The Rapid Solo Software Development (RSSD) Methodology is summarized in Fig. 2. While it uses the same phases as Agile (*Meet*, *Plan*, *Design*, *Develop*, *Test*, *Evaluate*), RSSD has different values and subphases. RSSD focuses on the following three values: *Efficient*, *modular* and *revisitable*. The phases and subphases of RSSD are explained below, while the values are explained in Section 5.

4.1. Meet Phase

The *Meet Phase* is the initial meeting between the developer and the stakeholders (or client), and it consists of the following subphases, i.e., *discuss*, *stickman*, and *list and decide*. Here, the developer should create *stickman diagrams* (simple drawings) that illustrate the features requested by the client to confirm the final scope. This is to avoid communication and cultural barriers (Zarewa, 2019). The developer should then write down the features and confirm them before the meeting ends (*list and decide subphase*).

4.2. Plan and Pre-Evaluation Phase

The *Plan Phase* starts with *re-evaluating* (think, visualize, imagine) the features to decide on the feasibility of the project. Together with that, he/she can do the *chunk subphase*, which is to create a proposal that consists of the big and small chunks of the project, and the cost breakdown. The proposal will then be proposed to the stakeholders in the *pre-evaluation phase*. The developer can attach a copy of the draft list and diagrams from the initial meeting as an appendix when necessary.

4.3. Design Phase

The *design phase* consists of *diagram* and *structure* subphases. The developer should only draw some diagrams, especially the global diagram(s) and the important procedures. For example, the diagram can be a class diagram for OOP (Object Oriented Programming) projects, ERD (Entity Relationship Diagram) if it has a database, and Use Case Diagram for information systems. Examples of the important procedures include the main form, user input form, enemy AI or player controller (for a game), etc. In the *structure subphase*, developers need to create the functions and class declarations, with some comments on how they intend to use them. The structures should be kept in a separate file, which serves as documentation before the bodies are written.

4.4. Develop Phase

In this phase, the developer can create the *code body* based on the *structure* that was made in the previous phase. Developers can cycle through the *subphases* when necessary, i.e., *code body*, *integrate*, and *verify*. The *integrate* subphase is to connect different functions and classes, where one has to ensure that the output of a function is compatible with the input of another related function. Since RSSD adopts *modular* as a value, the functions should be able to be verified (*verify subphase*) individually, with as few dependencies as possible.

4.5. Test Phase

The *test phase* includes *test* and *optimize subphases*. Developers should *test* each of the functions using different test cases, as bugs usually happen when a function doesn't produce an expected result for certain inputs. Testing a function extensively using the *black box* method (using test cases without looking at the internal structure of the unit) is a prefered choice compared to using mathematical proof which is impractical in many cases (Clermont & Parnas, 2005). In addition, developers should *optimize* the codes to ensure the longevity of the program. For example, optimizations of SQL queries (Győrödi et al., 2021), arrays and loops (Sarma, 2015), have been proven to significantly improve the application's performance as the number of users grows.

4.6. Evaluate and Maintenance Phase

The *Evaluate Phase* consists of three subphases, i.e. *discuss* with stakeholders, *demonstrate* the software, and *collect* feedback, which can happen in multiple meetings. The developer also needs to do the *Maintenance phase* at the same time to *fix* bugs and *change* features as needed. In these *phases*, the developer needs to maintain a good relationship with the stakeholders by setting a suitable *threshold* of the willingness to do reworks on features, especially for minor changes. However, major changes should be discussed as they should incur additional costs. As for *fixing* bugs, the costs should not be borne by the client. The developer needs to put up around 50–75% of the total costs for maintenance (Koskinen et al., 2003).

5. Values And Optimization Measures Of Rssd Methodology

Cycles during the development process are always inevitable, such as fixing bugs and changing features. Skipping proper planning can lead to disagreements and more cycles. They can include wrong scope interpretations and conflicting contract clauses, which in turn cause stringent relationships with stakeholders (Butt et al., 2016). The examples of project changes and causes in software development are summarized in Table 2. The RSSD methodology, which focuses on being rapid, aims to minimize these cycles by making sure that each phase of the development is anchored to the three RSSD values; *efficient, modular, revisitable*. In each value, multiple optimization measures can be implemented to further enhance the software development process.

5.1. Efficient

In the *Meet Phase*, anchoring the initial agreement on both visuals (*stickman diagram*) and text (draft list of features) help to reduce potential disagreements. When writing a proposal in the *Plan Phase*, developers need to understand the difficulty of the project, which translates to costs breakdown, by identifying the big and small chunks of the program. Ideally, this has to be recorded in a formal contract form, or at least a non-erasable written conversation.

The *Design Phase* includes designing UML diagrams to help developers to develop the software in a systematic way (Hafeez et al., 2020), and the same goes for the code structure. This means making the declaration of classes and functions before doing the code body, as much as possible. Developers can do this partially while consecutively doing the *Develop Phase* because sometimes it's hard to visualize the whole idea before doing an implementation. Similarly, integrating codes can be done partially after the code structure and body.

Table 2
Examples of Project Changes in Software Development

Change Cause	Example	Direct Impacts	Indirect Impacts
Overestimation	Development turns out to be much simpler	Developer gains time and money benefit	It can cause trust issues. Developers can offer a compensation.
Underestimation	• Development turns out to be too complicated • The system lags because there are more users than expected • Scope was not defined clearly. During evaluation, the stakeholders expect more.	Developer needs to spend more time, while the initial budget might be underestimated.	• Deadline missed • Requesting for additional budget is not recommended unless there's a clear justification.
Emergence of new technology	Developer decides to make X from scratch because such free library did not exist; it exists then.	Developer has wasted the time on development.	Stakeholders feel that the development cost should've been lower.
Unawareness of technology	Developer does not realize that there's an easier way to develop X, he/she uses a complicated one.	Developer has wasted the time on development.	Stakeholders feel that the development cost should've been lower.
Unwillingness to adopt new technology	Developer decides to use an existing technology that he/she has been using for many years, despite being aware of the problems of it.	In the long run, vulnerability or stability issues can rise.	Not communicating such issue during the initial stage can cause trust issues.
Changes of features during development	Stakeholders want to add/remove features due to new policies (Cáliz et al., 2016) or ideas (Assi et al., 2021). Ideally, the features have to be detailed during initial meeting.	Developer spends more time to adapt the changes. Additional cost must be discussed.	Being flexible as a developer helps him/her to gain more trust from existing and new clients.
Unexpected events	Hard drive corruptions, stability issues or wrong calculation results during live testing (Coulon et al., 2013), laptop got stolen.	• Need to fix the bugs quickly. • Need to redo some works.	Can cause trust issues, especially when stakeholders feel like it's just an excuse.

Solo developers don't typically need a lot of diagrams, and some UML diagrams might overlap in functionality with each other. A flowchart is also a good option to briefly sketch out a program (Winnie, 2021), but not very helpful for complicated codes. Here is the list of UML diagrams, ordered from the most important (Reggio et al., 2014): Sequence, class, use case, state machine, activity.

In the *Test phase*, developers should *optimize* the code as much as possible. Practically, this includes the following optimizations (but not limited to), i.e., time complexity (loops, recursions, number of instructions), space complexity (number of variables, arrays, constant folding) (Sarma, 2015), and SQL query optimizations (when applicable). Time complexity optimization is especially important if the code complexity is N^2 or above. Lastly, in the *Evaluate Phase*, the developer needs to refer to the original contract when demonstrating the software.

5.2. Modular

The final product should be modular, i.e., divided into small functions as much as possible. Skipping careful planning can lead to messy codes after several updates or revisits, especially when adding and removing features (Hare & Kaplan, 2017).

In the *Plan phase*, developers can start to think of ways to split the code into small chunks. In the *Develop* (and *Design*) phase, the following aspects have to be taken care of when designing the code *structure* to ensure modularity (Litzsinger & Riddle, 2002) (Sankaranarayanan & Kulkarni, 2013) (Sierra & Bates, 2014):

1. Reduce complexity: Break down complex problems. Avoid deeply nested and complex logic; code should be separated into functions.

2. Increase flexibility: Make a flexible structure, such as using a global variable or database value for elements that need to be adjusted frequently.
3. Eliminate repetition: Do not repeat codes, such as by grouping codes into functions.
4. Reduce dependency: Make functions less dependent on global variables.
In the *Test phase*, each function should be tested using multiple test cases to prevent bugs. Each function should be able to be tested individually as much as possible, without having to preset any variables.

5.3. Revisitable

A software project can typically last for 3–12 months (Aguilar et al., 2014), and even up to 2–3 years for a solo development. Revisiting the code after months can present a challenge to understanding the code. In the *Plan* and *Pre-Evaluation Phase*, the final list of features and budget breakdown have to be made and understandable by both parties, which includes the project timeline, architectural design, requirement specification, and testing process (Debbiche et al., 2019). Some technical aspects have to be included, which helps in solving possible disagreements, especially the types of algorithms and tools. Stakeholders might expect newer algorithms (encryption method, compression algorithm, machine learning or AI method, etc.) or newer tools (library version, language version, database technology, protocols, API version, etc.).

In the *Develop* (and *Design*) phase, codes should be easily revisited by considering the following aspects:

1. Comments Optimization: Add proper code comments to ensure comprehension (Rani et al., 2021). In the early stages of development, developers typically rush to complete the task, with no time to write proper comments. They can revisit the codes as early as possible to correct the comments, including documentation comments for languages like Java.
2. Readability Optimization: To ensure that codes are understandable (Holzmann, 2016) and self-explanatory (Lawrie et al., 2006). Using English language for the identifier namings and comments is preferable, to easily transfer the code to another developer or to upload the code to code-sharing platforms.
3. OOP Optimization: Important points for object-oriented programming (OOP) include keeping variable scope as small as possible, avoiding designing class with no methods, making use of overloading, avoiding long arguments list, using getters/setters (Sierra & Bates, 2014)
In the *Test, Evaluate* and *Maintenance* phases, any logic changes have to be documented in the code comments. For example, the initial code has a complexity of N^2, and this code has been released. The developer then realized that it can be reduced, so some logic was changed later. The comments on what was changed and when are helpful to identify potential failures in future test cases. Additionally, a Version Control system with an appropriate interval between commits helps developers to learn from mistakes. Lastly, in the *Evaluate phase*, the developer needs to create a checklist based on the initial contract. The final agreement has to be recorded in written form.

6. Evaluation For Rssd Development Phases

The RSSD methodology was tested by the respondents (software developers) in 10 projects. The project timeline is shown in

Table 3. On average, the proportion of each phase of RSSD is shown in Fig. 3. The *evaluation* and *maintenance* phase was the largest part of the project, reaching 42.1% of the total project, while the *development phase* was the second largest (35.09%).

Table 3
RSSD Implementation Result on the Software Development Timeline

Case #	Dev #	Software Name	Language	Total Days	Development Duration (% of total days)					
					Meet *	Plan & Pre-Ev	Design	Develop	Test	Eval. & Maint.**
C1	D1	Academic information system	PHP + JS	194	7.7 (2)	4.6	5.2	35.1	5.7	41.8 (6)
C2	D1	Data analytics for eCommerce	PHP + JS	202	8.9 (3)	5	6.4	30.2	4	45.5 (4)
C3	D2	Clinic information system	PHP + JS (Laravel)	129	5.4(2)	6.2	5.4	34.9	3.9	44.2 (5)
C4	D2	Dental clinic system	PHP + JS (Laravel)	115	8.7 (2)	7	5.2	29.6	5.2	44.3 (4)
C5	D3	Casual game	Unity C#	130	5.4(2)	6.9	5.4	26.9	9.2	46.2 (3)
C6	D4	Online poker game	Unity C#	256	5.5 (2)	4.3	2.7	31.3	6.3	50 (10)
C7	D5	Point of sales (metal industry)	PHP + JS	175	0.6(1)	4.6	4.6	38.9	5.1	46.3 (6)
C8	D5	Point of sales (cafe)	PHP + JS	154	4.5(2)	5.2	5.8	46.8	6.5	31.2 (3)
C9	D6	Health mobile app	HTML + JS (Cordova)	199	10.6 (5)	6	5.5	40.7	5	32.2 (4)
C10	D7	Attendance system	Android & Web	178	9 (3)	4.5	5.6	36.5	5.1	39.3 (7)
				Average	**173.2**	**6.63%**	**5.43%**	**5.18%**	**35.09%**	**5.6% 42.10%**

* *Meet phase*: Duration (number of meetings). The *Meet* and *Plan phases* were sometimes prolonged due to the stakeholder's availability.

** *Evaluation* and *Maintenance* phase: Duration (number of meetings):

- The *evaluation phase* starts from the first time the final draft was handed over to the stakeholder
- The *maintenance* phase is based on 30 days cut-off period, where if there is no updates from the stakeholders, the (major) maintenance period is considered over.

In addition, a post-interview was also conducted to measure the success of RSSD. The survey questions are listed in Table 4, while the result is listed in Table 5. Each question was answered using the Likert scale (1–5). The overall score of the post-interview was 4.19/5.0, which means that the respondents agreed that RSSD was helpful. The question *Po1* scores 4.30/5.0, which means RSSD helped in streamlining the development process. The results for each phase are discussed below.

Table 4
Post-Interview Questions (Quantitative)

#	Dev. Phase	Interview Question	Scale*
Po1	All	RSSD methodology helps in streamlining the development process, while maintaining the robustness and scalability of the software	1–5
Po2	Meet	Drawing sketches and showing them to the stakeholders were useful	1–5
Po3	Meet	The initial meeting went smoothly	1–5
Po4	Plan and Pre-Evaluation	The plan phase went smoothly as all information was collected thoroughly during the initial meeting	1–5
Po5	Plan and Pre-Evaluation	Identifying the small and big chunk of codes was useful	1–5
Po6	Design	Drawing only several diagrams were useful for better planning	1–5
Po7	Design	Making code structures before the development were useful	1–5
Po8	Develop	The development went smoothly because there was a careful planning	1–5
Po9	Develop	The difficulties during the development can be handled well	1–5
Po10	Test	Testing the codes with different test cases was useful to ensure stability	1–5
Po11	Test	Optimizing codes, such as minimizing loops and arrays, was useful	1–5
Po12	Evaluate	The presentation to the stakeholders went smoothly	1–5
Po13	Evaluate	The evaluate phase went smoothly, with minimal reworks needed	1–5
Po14	Maintenance	Changes during the maintenance phase can be handled well as there were careful plannings beforehand	1–5
* Likert Scale 1–5: 1 = strongly disagree, 2 = disagree, 3 = neutral, 4 = agree, 5 = strongly agree			

Table 5. Post-Interview Results

Question	Dev. Phase	Project (Case #)										Average
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Score
Po1	All	5	4	4	5	4	3	5	5	4	4	4.30
Po2	Meet	4	4	4	4	4	5	5	4	4	3	4.10
Po3	Meet	4	5	5	4	4	5	5	5	5	4	4.60
Po4	Plan	5	5	4	4	4	3	4	4	4	5	4.20
Po5	Plan	4	3	4	4	4	3	4	5	4	5	4.00
Po6	Design	5	5	3	4	3	4	5	5	4	4	4.20
Po7	Design	5	5	4	4	3	5	5	4	4	4	4.30
Po8	Develop	5	4	4	5	5	5	4	4	4	4	4.40
Po9	Develop	4	5	4	5	4	3	4	4	4	4	4.10
Po10	Test	4	4	3	4	4	5	4	4	4	4	4.00
Po11	Test	5	5	4	4	3	5	3	4	3	4	4.00
Po12	Evaluate	5	4	4	5	4	3	4	5	4	4	4.20
Po13	Evaluate	3	4	4	4	5	3	5	5	5	4	4.20
Po14	Maintenance	3	4	5	5	5	3	4	4	4	4	4.10
Average		4.36	4.36	4.00	4.36	4.00	3.93	4.36	4.43	4.07	4.07	4.19

6.1. Meet Phase

Questions *Po2* and *Po3* are related to the *Meet* phase, which produced scores of 4.1 and 4.6 out of 5.0, respectively. Respondent *D7* stated that drawing sketches (*Po2*) were sometimes not doable when the meeting time is short.

6.2. Plan and Pre-Evaluation Phase

The questions *Po4* and *Po5* are related to the *Plan Phase*, which produced scores of 4.2 and 4.0 out of 5.0, respectively. Based on *Po4*, the respondents agreed that the *Plan phase* was smooth. Respondent *D7* highlighted that identifying the big and small chunks during this phase (*Po5*) is useful. Respondent *D4* stated that project *C6* was smooth during the initial meeting. However, there were disagreements about the online system during the *Plan* phase as there were overlooked aspects during the meeting. It should've been better planned.

6.3. Design Phase

For the *design phase*, based on *Po6* and *Po7*, the respondents generally agreed that creating diagrams and code structures was helpful in the later phases. As stated by *D1*:

"...For developing information systems, especially, I found it useful to create the code structure before implementing it, because big projects tend to be messier the longer it goes..." (D1)

Respondents *D2* and *D3* highlighted that creating diagrams and structure was a bit complicated at first, especially for a medium project for solo developers; however, they were eventually useful:

"...As a solo developer, I don't usually create diagrams. However, making some important diagrams (sequence, in my case) helped me to plan my app more systematically, which in turn decreases complications in the end..." (D2)

"...The systematic subphases in RSSD really helped me to develop my game more quickly. The initial subphases, such as creating structure and diagrams look complicated for solo development, but it helped me to focus on the codes..." (D3)

6.4. Develop Phase

Based on questions *Po8* and *Po9*, respondents generally agreed that the *Develop phase* was smooth and difficulties can be handled well. Respondent *D4* highlighted that the code structure that was made during the *Design Phase* was helpful, especially in difficult projects:

"…The meeting was smooth, even though I encountered a lot of difficulties during the implementation, especially on the synchronization processes. During the design phase, I planned by making codes structure as suggested. The structure and the diagrams serve as guides as the codes grow tremendously…" (D4)

Respondents *D5* and *D6* also highlighted that the *Develop phase* was smooth, particularly to develop the software systematically:

"…During the development, creating the code structure helps during the integration…" (D5)

"…As mainly information systems developers, the detailed subphases of RSSD are very practical. It prevents hassle towards the end of the project. The project is my master thesis, and RSSD has helped me develop the app in a systematic way…" (D6)

6.5. Test Phase

The respondents generally agreed that the *Test Phase* was smooth, based on *Po10* and *Po11.* However, the average score for the *Test Phase* is the lowest among all the phases, which is 4.0. Some respondents find it useful to do code testing and optimization to prevent mess during the maintenance:

"…The methodology has made me realize the importance of code optimization before the amount of data in the system becomes much larger. I found an inefficient loop in my system that causes the student's grade saving process to be 70% slower. I wish I knew this earlier…" (D1)

"…Testing each method individually helped me to quickly identify wrong loops, excessive array usage, inefficient SQL queries, vulnerabilities, etc…" (D7)

"…I also constantly made code optimizations as the number of users in the game grows, especially for the socket system and queries, which reduced the database and server load significantly…" (D4)

In contrast, respondent *D3* stated that code optimization might not be needed for projects that have to be completed quickly, and respondent *D5* prefer not to optimize the code because there were only a few users using the software:

"…As for the code optimization part, I did that for the enemy AI code that slightly reduced the CPU usage. Even though such optimization might be more critical for information systems or bigger games, as opposed to small games with a quick development cycle, I think it helps to reduce the complexity of codes, especially for revisits…" (D3)

"…During the test phase, I successfully tested the methods using different test cases. However, for optimization, I only did some SQL query optimization, because I think the app will still work fine with just a few concurrent users…" (D5)

6.6. Evaluate and Maintenance Phase

Based on *Po12*, *Po13*, and *Po14*, the *Evaluate* and *Maintenance* phases went smoothly with minimal re-works needed due to the careful planning in the earlier phases. Project *C6* was a difficult project with the longest project duration (256 days) and the longest *Evaluate and Maintenance Phase*s (50% of the total project). However, RSSD helped to easier project navigation:

"…The evaluate and maintenance phases were still difficult simply because the app is huge with a lot of users involved, which produced a lot of varieties. Still, the plans from the earlier phases have helped me to navigate the project…" (D4)

7. Evaluation For Rssd Values And Optimization Measures

RSSD aims to minimize project cycles by doing optimization measures based on the three RSSD values, as discussed in Section 5. The testing result for the optimization measures is shown in Fig. 4, separated by each value and each optimization measure. Overall, the average *implementation* score is lower than *important* (4.52 vs. 4.32, respectively). While these measures are important, sometimes budget and time become the main constraints.

In the *Meet* phase, both the stickman diagram and draft feature list were mostly implemented, even though the stickman diagram was considered as not as important as the draft list. This was due to, in some meetings, the developers didn't have a large screen to project the diagram and the list. In the *Plan and Pre-Evaluation* phases, developers generally agreed that the measures are important and have been implemented, except for item *Vq4*. The developers argued that for the *Plan* phase, listing the big chunks is enough to start the project and estimate the budget.

In the *Design* phase, *Vq7* (UML diagram) fell into the range of 3.0–4.0 for the *implementation* score, which is very different from the *importance* score (4.13). As solo developers, the respondents feel that they can directly start a project without having any diagrams, even though the RSSD document already mentioned that only some necessary UML diagrams are needed. Despite so, the average *importance* of UML diagrams is higher for information systems (project C1, C3, C7, C8, C10) compared to other types of projects, which is 4.4 vs. 3.67, respectively. Two respondents (projects C4 and C5) answered n.a. for *Vq7*, and these two projects were the shortest in duration among others. These facts indicate that UML diagrams are still necessary for longer projects, especially for information systems.

The importance of making functions structure (*Vq8*) was also rated as very important, and it was very well implemented. However, the class structure (*Vq9*) fell into the range of 3.0–4.0 for *implementation* even though it was considered very important. This was because expert developers can start the development without much planning. For *Vq9*, since most projects are not OOP-based, six respondents answered n.a.

In the *Develop* phase, all the optimization measures were considered very important. Interestingly, some time-consuming measures (*Vq12*, *Vq14*, *Vq15*) fell into the range of 3.0–4.0 for *implementation*. These measures are important for the longevity of a project, especially when there will be many changes. However, most projects were limited by time and budget constraints. Developer D1 also mentioned that these measures, especially to increase flexibility, are useful for adapting previous projects to a new one.

In the *Test* phase, the optimization measures are aimed at reducing time and space complexity. All measures (*Vq18* to *Vq27*) were rated as very important, except for constant folding (*Vq23*). Compared to other optimization measures, constant folding is a less popular technique and might not be available on all compilers. The optimization of the number of variables and arrays (*Vq21* and *Vq22*) was rated the highest among others. SQL query optimization (*Vq24*), on the other hand, has a high *importance* score (4.88) but is slightly lower in *implementation* score (4.38). According to the respondents, this is because optimizing queries is trickier compared to optimizing other aspects.

Lastly, in the *Evaluation and Maintenance* phases, the optimizations measures are rated as very important. However, while documenting code changes (*Vq28*) is very important, some developers were reluctant to do it because, by the time they reach the *maintenance phase*, they were usually busy with new projects, with not much time to focus on previous projects.

8. Conclusion

Overall, the proposed RSSD methodology was well accepted by the respondents. Compared to Agile, it has different subphases to cater for the need for rapid solo development. Additionally, RSSD follows the following values: *Efficient*, *modular*, and *revisitable*. Each value has its own optimization measures. According to the testing results, important optimization measures include reducing complexity and eliminating repetition during the *Develop* phase, optimizing variables and arrays in the *Test* phase, testing with test cases during both the *Develop* and *Test* phases, etc.

The developers agreed that following the whole RSSD process is useful to prevent various future problems. However, there were constraints, such as (1) Limited time in the initial meeting, which leads to some overlooked aspects; (2) Some projects have a limited timeframe, thus creating diagrams and structure during the *design phase* is challenging; (3) Some developers decided not to do a thorough code optimization because the project is small or have a limited budget; and (4) Maintenance *phase* can be challenging because the developers will have new projects by that time.

In future works, researchers can contribute by enriching the values and the phases in software development. Certain aspects of the methodology can be categorized, such as different phases for different software types and scopes.

Declarations

CONFLICT OF INTEREST

The authors declared that they have no conflict of interest.

Author Contributions

All authors were involved in the whole discussion, initiative, and final review process. Kristo Radion Purba wrote the main manuscript text, except for Chapter 2. Rusyaizila Ramli wrote Chapter 2 of the manuscript.

FUNDING ACKNOWLEDGEMENTS

The authors received no financial support for the research, authorship, and/or publication of this article.

References

1. Aguilar, J., Sanchez, M., Fernandez-y-Fernandez, C., Rocha, E., Martinez, D., & Figueroa, J. (2014). The size of software projects developed by Mexican companies. *ArXiv Preprint ArXiv:1408.1068*.
2. Assi, I., Tailakh, R., & Sayyad, A. (2021). Survey on Software Changes: Reasons and Remedies. International Arab Journal of Information Technology, *18*, 248. https://doi.org/10.34028/iajit/18/2/14
3. Bogdan-Alexandru, A., Andrei-Cosmin, C.-P., Sorin-Catalin, G., & Costin-Anton,, BOIANGIU. (2019). A Study On Using Waterfall And Agile Methods In Software Project Management. *JOURNAL OF INFORMATION SYSTEMS & OPERATIONS MANAGEMENT* .
4. Butt, A., Naaranoja, M., & Savolainen, J. (2016). Project change stakeholder communication. International Journal of Project Management, *34*, 1579–1595. https://doi.org/10.1016/j.ijproman.2016.08.010
5. Cáliz, D., Samaniego, G., & Cáliz, R. (2016). Methodological Proposal of Policies and Procedures for Quality Assurance in Information Systems for Software Development Companies Based on CMMI. Journal of Software, *11*, 230–241. https://doi.org/10.17706/jsw.11.3.230-241
6. Chaudhari, A. R., & Joshi, S. D. (2021). Study of effect of Agile software development Methodology on Software Development Process. *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*, 1–4. https://doi.org/10.1109/ICESC51422.2021.9532842
7. Clermont, M., & Parnas, D. (2005). Using information about functions in selecting test cases. *Proceedings of the 1st International Workshop on Advances in Model-Based Testing, A-MOST '05*, *30*. https://doi.org/10.1145/1083274.1083276
8. Coulon, T., Barki, H., & Pare, G. (2013). Conceptualizing unexpected events in IT projects. *International Conference on Information Systems (ICIS 2013): Reshaping Society Through Information Systems Design*, *1*.
9. Debbiche, F., Wrang, M., & Sinkala, K. (2019). *Accelerating software delivery in the context of requirements analysis and breakdown for devops: A multiple-case study*.
10. Flora, H., Wang, X., & Chande, S. (2014). Adopting an Agile Approach for the Development of Mobile Applications. International Journal of Computer Applications, *94*, 43–50. https://doi.org/10.5120/16454-6199
11. Győrödi, C., Dumşe-Burescu, D., Gyorodi, R., Zmaranda, D., Livia, B., & Popescu, D. (2021). Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases. Applied Sciences, *11*, 6794.

https://doi.org/10.3390/app11156794

12. Hafeez, A., Khuhro, M., Furqan, M., & Husain, I. (2020). *Importance and Impact of Class Diagram in Software Development*.
13. Hare, E., & Kaplan, A. (2017). Designing modular software: a case study in introductory statistics. Journal of Computational and Graphical Statistics, *26*(3), 493–500.
14. Hassani, H., Kadir, G., Al-Salihi, N., Monnet, W., Ali-Yahiya, T., & Alizadeh, F. (2018). *Supervision of Undergraduate Final Year Projects in Computing: A Case Study*. *8*, 210. https://doi.org/10.3390/educsci8040210
15. Holzmann, G. (2016). Code Clarity. IEEE Software, *33*, 22–25. https://doi.org/10.1109/MS.2016.44
16. Hron, M., & Obwegeser, N. (2022). Why and how is Scrum being adapted in practice: A systematic review. Journal of Systems and Software, *183*, 111110. https://doi.org/10.1016/J.JSS.2021.111110
17. Kišš, F., & Rossi, B. (2018). Agile to Lean Software Development Transformation: A Systematic Literature Review. *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 969–973.
18. Koskinen, J., Lahtonen, H., & Tilus, T. (2003). *Software Maintenance Cost Estimation and Modernization Support 6 Approaches for Software Modernization and Its Support*.
19. Kumar, R., Maheswary, P., & Malche, T. (2019). Inside Agile Family Software Development Methodologies. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING*.
20. Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006). What's in a Name? A Study of Identifiers. *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 3–12.
21. Litzsinger, M. A., & Riddle, M. A. (2002). A Modular Approach to Portable Programming. *SAS Conference Proceedings: SAS Users Group International 27 (SUGI 27)*.
22. Moyo, S., & Mnkandla, E. (2019). A Metasynthesis of Solo Software Development Methodologies. *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, 1–8. https://doi.org/10.1109/IMITEC45504.2019.9015867
23. Pagotto, T., Fabri, J. A., Lerario, A., & Goncalves, J. (2016). *Scrum solo: Software process for individual development*. 1–6. https://doi.org/10.1109/CISTI.2016.7521555
24. Rani, P., Birrer, M., Panichella, S., Ghafari, M., & Nierstrasz, O. (2021). What do developers discuss about code comments? *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 153–164.
25. Reggio, G., Leotta, M., & Ricca, F. (2014). Who knows/uses what of the UML: A personal opinion survey. *International Conference on Model Driven Engineering Languages and Systems*, 149–165.
26. Sankaranarayanan, H., & Kulkarni, P. (2013). Source-to-Source Refactoring and Elimination of Global Variables in C Programs. Journal of Software Engineering and Applications, *06*, 264–273. https://doi.org/10.4236/jsea.2013.65033
27. Sarma, A. (2015). New trends and Challenges in Source Code Optimization. International Journal of Computer Applications, *131*, 975–8887. https://doi.org/10.5120/ijca2015907609
28. Shaydulin, R., & Sybrandt, J. (2017). To Agile, or not to Agile: A Comparison of Software Development Methodologies. *ArXiv*, *abs/1704.07469*.
29. Sierra, K., & Bates, B. (2014). *OCA/OCP Java SE 7 Programmer I & II Study Guide, Exams 1Z0-803 & 1Z0-804*. McGraw-Hill Education Group.
30. White, K. (2015). Freelancing – Are you ready to go solo? Medical Writing, *24*, 140–144. https://doi.org/10.1179/2047480615Z.000000000309
31. Winnie, D. (2021). Flowcharting. In *Essential Java for AP CompSci: From Programming to Computer Science* (pp. 23–27). Apress. https://doi.org/10.1007/978-1-4842-6183-5_7
32. Zarewa, G. (2019). Barriers to Effective Stakeholder Management in the Delivery of Multifarious Infrastructure Projects (MIPs). Journal of Engineering, Project, and Production Management, *9*, 85–96. https://doi.org/10.2478/jeppm-2019-0010
33. Zayat, W., & Senvar, O. (2020). Framework Study for Agile Software Development Via Scrum and Kanban. International Journal of Innovation and Technology Management, *17*(04), 2030002. https://doi.org/10.1142/S0219877020300025

Figures

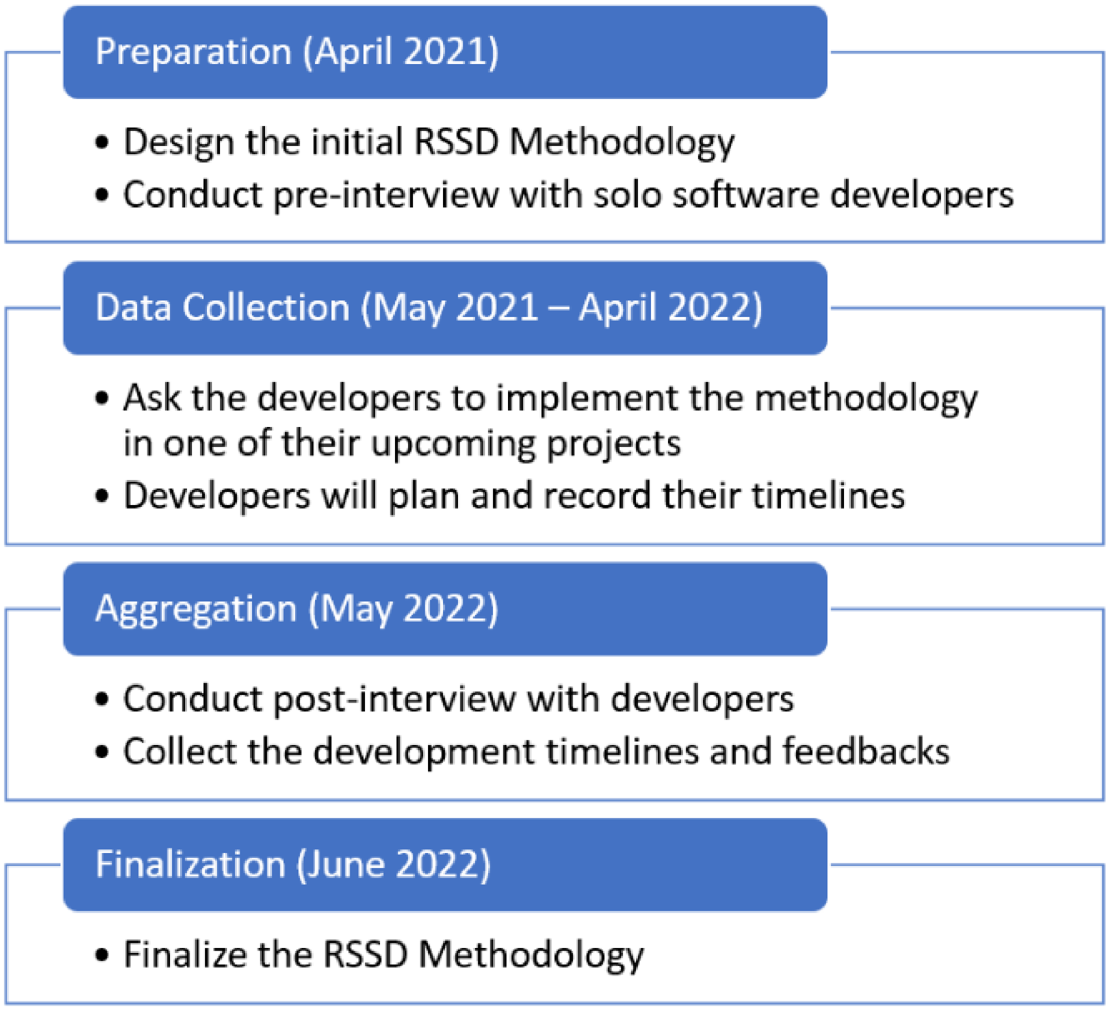

Preparation (April 2021)
• Design the initial RSSD Methodology
• Conduct pre-interview with solo software developers
Data Collection (May 2021 – April 2022)
• Ask the developers to implement the methodology in one of their upcoming projects
• Developers will plan and record their timelines
Aggregation (May 2022)
• Conduct post-interview with developers
• Collect the development timelines and feedbacks
Finalization (June 2022)
• Finalize the RSSD Methodology

Figure 1

Research Methodology

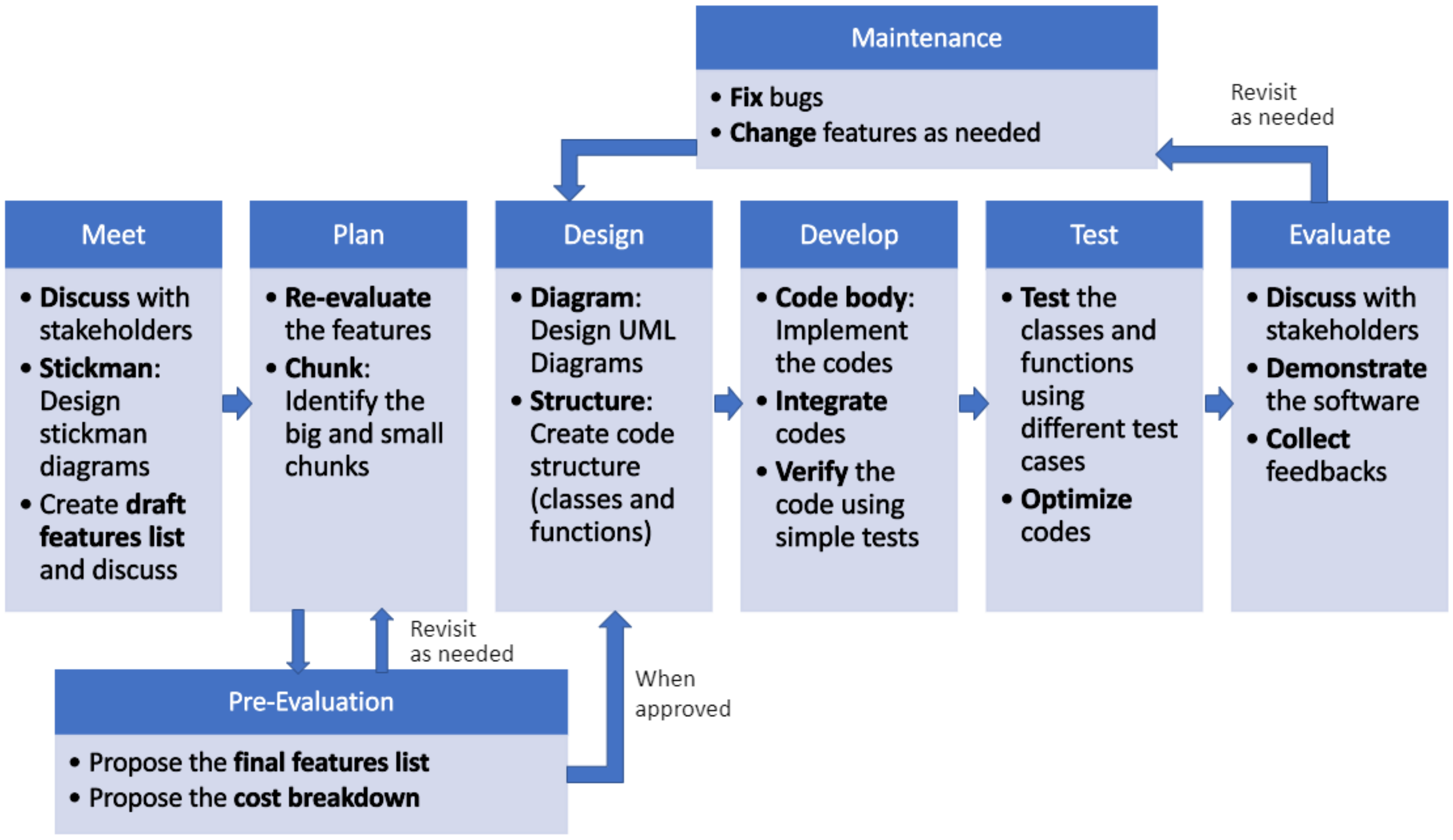

Maintenance
• Fix bugs
• Change features as needed
Revisit as needed
Meet
• Discuss with stakeholders
• Stickman: Design stickman diagrams
• Create draft features list and discuss
Plan
• Re-evaluate the features
• Chunk: Identify the big and small chunks
Design
• Diagram: Design UML Diagrams
• Structure: Create code structure (classes and functions)
Develop
• Code body: Implement the codes
• Integrate codes
• Verify the code using simple tests
Test
• Test the classes and functions using different test cases
• Optimize codes
Evaluate
• Discuss with stakeholders
• Demonstrate the software
• Collect feedbacks
Revisit as needed
When approved
Pre-Evaluation
• Propose the final features list
• Propose the cost breakdown

Figure 2

Rapid Solo Software Development (RSSD) Methodology

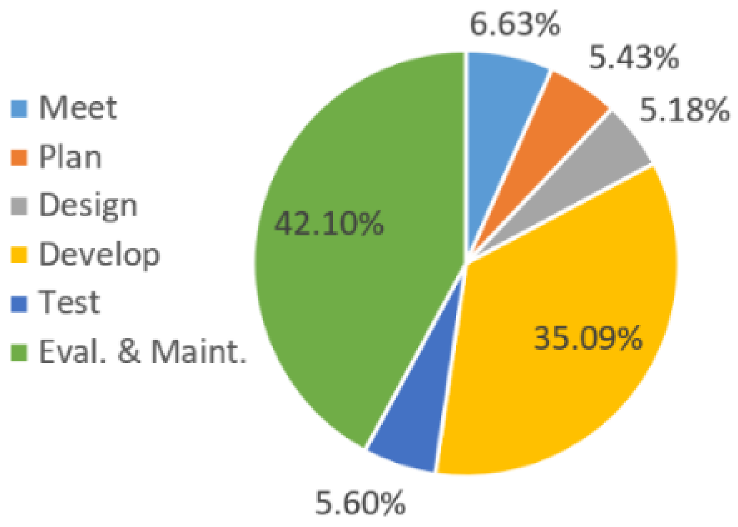
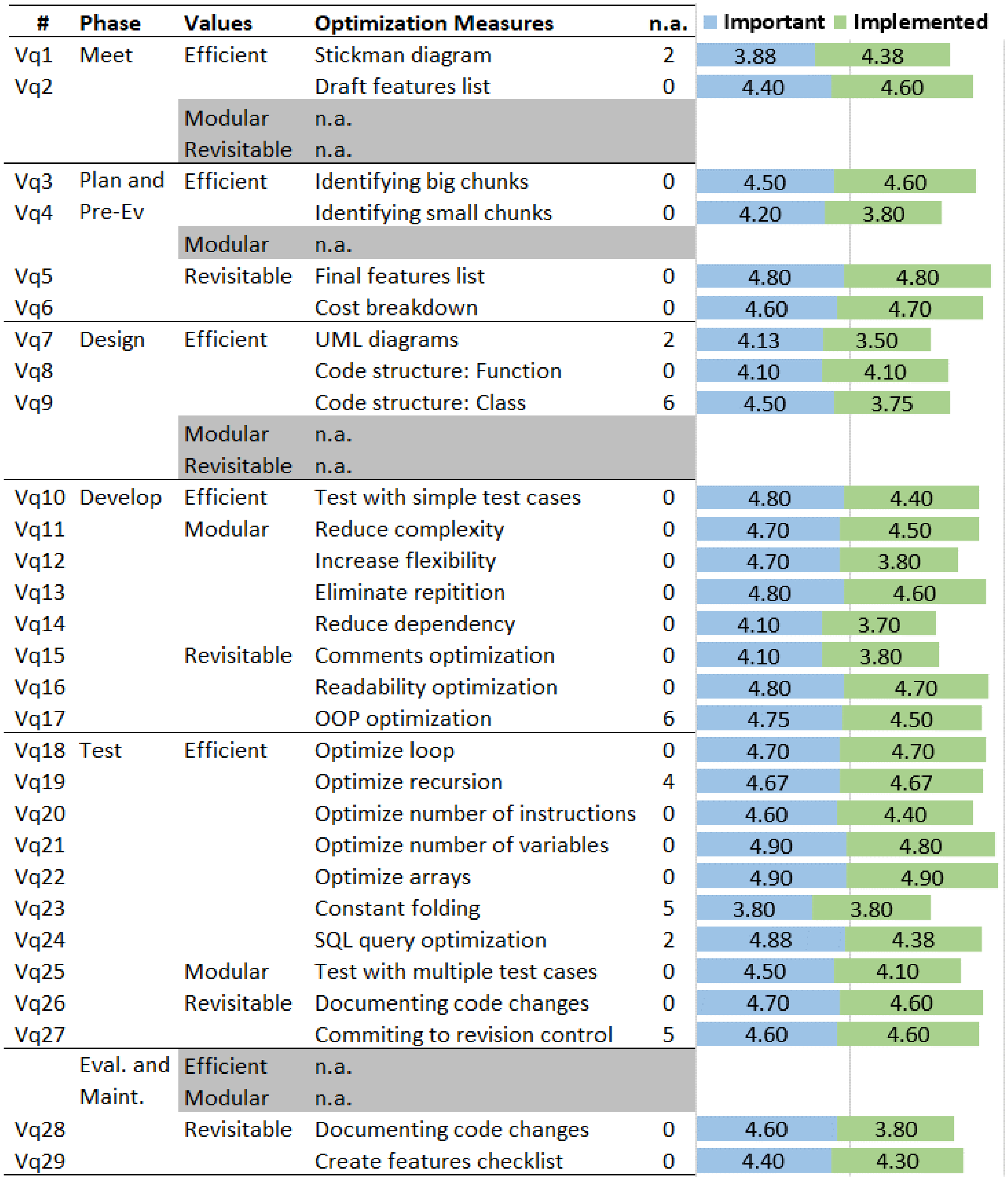


Figure 3

Average Proportion of Each Phase of RSSD



#	Phase	Values	Optimization Measures	n.a.	🟦 Important	🟩 Implemented
Vq1	Meet	Efficient	Stickman diagram	2	3.88	4.38
Vq2			Draft features list	0	4.40	4.60
		Modular	n.a.			
		Revisitable	n.a.			
Vq3	Plan and	Efficient	Identifying big chunks	0	4.50	4.60
Vq4	Pre-Ev		Identifying small chunks	0	4.20	3.80
		Modular	n.a.			
Vq5		Revisitable	Final features list	0	4.80	4.80
Vq6			Cost breakdown	0	4.60	4.70
Vq7	Design	Efficient	UML diagrams	2	4.13	3.50
Vq8			Code structure: Function	0	4.10	4.10
Vq9			Code structure: Class	6	4.50	3.75
		Modular	n.a.			
		Revisitable	n.a.			
Vq10	Develop	Efficient	Test with simple test cases	0	4.80	4.40
Vq11		Modular	Reduce complexity	0	4.70	4.50
Vq12			Increase flexibility	0	4.70	3.80
Vq13			Eliminate repitition	0	4.80	4.60
Vq14			Reduce dependency	0	4.10	3.70
Vq15		Revisitable	Comments optimization	0	4.10	3.80
Vq16			Readability optimization	0	4.80	4.70
Vq17			OOP optimization	6	4.75	4.50
Vq18	Test	Efficient	Optimize loop	0	4.70	4.70
Vq19			Optimize recursion	4	4.67	4.67
Vq20			Optimize number of instructions	0	4.60	4.40
Vq21			Optimize number of variables	0	4.90	4.80
Vq22			Optimize arrays	0	4.90	4.90
Vq23			Constant folding	5	3.80	3.80
Vq24			SQL query optimization	2	4.88	4.38
Vq25		Modular	Test with multiple test cases	0	4.50	4.10
Vq26		Revisitable	Documenting code changes	0	4.70	4.60
Vq27			Commiting to revision control	5	4.60	4.60
	Eval. and	Efficient	n.a.			
	Maint.	Modular	n.a.			
Vq28		Revisitable	Documenting code changes	0	4.60	3.80
Vq29			Create features checklist	0	4.40	4.30

Figure 4

Evaluation for RSSD Values and Optimization Measures. The questions were answered for each project. For each question, the developer should answer whether it is not applicable (shown as n.a., and the number shows the count of it), how important it is in the project (shown as Likert scale 1-5; 1=not important, 2=less important, 3=neutral, 4=important, 5=very important), and how much it is implemented in the project (shown as Likert scale 1-5; 1=not implemented, 2=less implemented, 3=implemented, 4=well implemented, 5=very well implemented).