

Article

Software Development Process Considerations in GNSS-Denied Navigation Project for Drones

Sebastian Rutkowski and Cezary Szczepański * 

Lukasiewicz Research Network—Institute of Aviation, Al. Krakowska 110/114, 02-256 Warsaw, Poland

* Correspondence: cezary.szczepanski@ilot.lukasiewicz.gov.pl

Abstract: This article will discuss the software development process utilised in the GNSS-denied navigation project targeted for drones. The process was implemented in an environment where software developers were oriented to developing a source code only and who showed great reluctance to follow any formal process. The mentioned process was a lightweight AGILE-based process that, by assumption, minimised developers' engagement in activities other than those related to the source code development. The process described in this paper was designed to support and reflect product quality characteristics like functional completeness and correctness, time behaviour, resource utilisation, analysability, modifiability and testability in an "implicit" way. It allowed the developers to achieve those characteristics in an "invisible" manner. This allowed us to achieve acceptable product quality without needing to engage experienced architects. The described process needs improvements and extensions to satisfy certification needs. However, for the prototyping or research phases, it may be a perfect solution for facilitating future product modifications and saving costs. The process improves product quality by underlying the value of transparency, performing specific actions and incorporating specific attributes related to team tasks. This research also focused on finding a metric related to process acceptance in a given environment, resulting in a three-degree scale showing whether the process needed improvements or was accepted by the environment. Such a metric is a new finding and the metric implies that there are some factors that stand behind any process, which determine whether it will be friendly to the environment (people with their habits, work style, personal traits, etc.) or not. However, investigating factors that determine the process acceptance and that are connected to the environment traits were beyond the scope of this research and will be considered in the future.



Citation: Rutkowski, S.; Szczepański, C. Software Development Process Considerations in GNSS-Denied Navigation Project for Drones. *Appl. Sci.* **2024**, *14*, 10347. <https://doi.org/10.3390/app142210347>

Academic Editor: Yangquan Chen

Received: 30 September 2024

Revised: 28 October 2024

Accepted: 31 October 2024

Published: 11 November 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: software engineering; systems engineering; quality; development processes; drone-dedicated software development

1. Introduction

Has anyone ever mentioned what quality is actually about? Is it something that we see that looks good? Or is it soft and nice to touch? Or is it something that is not tangible and is invisible? If one ever heard such questions or wondered how to find an answer, one would probably also notice that the answer is tangible and invisible. The invisible part is usually related to the manufacturer or solution provider, but the tangible part is rather familiar to the end user.

Plato defined a quality as “some degree of perfection”. Studies on quality have unveiled the internal and external quality factors that significantly impact product quality. Internal factors are invisible to the end customer and are dependent on the overall work organisation by the development team. This work organisation is called “the process”.

As mentioned in [1,2], the development process is a key factor in achieving software product quality. The high-quality impact on cost savings is also mentioned in [2,3]. Our environment, however, dominated by software developers working with the source code

only, is characterised by the reluctance to follow any formal process. A process is important for product certification, though. The tendency to jump into design “right now” without obtaining greater knowledge about the problem to solve or the solution itself is mentioned in [4,5]. We encountered the same issue. We were then faced with a choice, either to impose any well-defined process and struggle with its execution by the developers or to design a lightweight process and progressively measure how much is acceptable and modify it to better match the environment. A lightweight process is more likely to be accepted in the beginning by developers than a fully-fledged one.

Analysability, modifiability and testability are important cost-saving characteristics that need to be ensured, especially when the product is going to be improved in the future or if there is a vague understanding of the solution at the beginning. A vague understanding and knowledge maturation during development may cause a lot of changes after product development starts. This is common to projects targeted to develop innovative solutions and is almost absent in typical ones. See Section 3 for considered factors in the case of this article.

Our environment, dominated by software developers oriented in source code development only, instead of being reluctant to formalise, is also characterised by the lack of software architects and overall experience in developing software architectures, which, as stated in [3,6,7], are important to ensure software product quality characteristics, especially like modifiability and analysability.

This research focused on supporting product quality characteristics without the need to put a big emphasis on product design, and to look for a metric that will allow us to assess process acceptance by the environment. Further investigations on factors deciding on acceptance or rejection will be provided in the future as this requires more data and experiments. Results of future research will enable designers to consider process traits while designing a development process in any project. This research found a solution how to measure whether the process described in the further text needs changes or is designed enough to be accepted by the given reluctant environment. These results will be taken into account in further research as evaluation criteria.

Comparing the proposed process with a standard AGILE process, like the one mentioned for instance in [8], it is worth highlighting that the proposed one does not foresee a strong interaction with the user. It is actually based on an assumption that the customer shows very weak engagement or that it is an internal customer (different division in the same company). The only customer involvement required while validating requirements is what happens before the process starts. Having this, customer needs are assumed to be known and stable for the rest of the development effort. This is also a good practice while having safety requirements to satisfy. In the case of the existence of safety requirements, as mentioned in [9], a more predictive (plan-driven) development approach will be suitable over an adaptive one. Safety requirements have their source in analysis, such as preliminary hazard analysis (PHA) [10], functional hazard analysis (FHA) [10] and so on, or failure mode and effect analysis (FMEA) [11] analysis, and very often safety requirements are implicit requirements (not stated clearly) that need to be uncovered by engineers. Safety requirements also need safety tests to be planned and performed, which requires more broad knowledge at the beginning to ensure adequate resources and prepare adequate test scenarios. Regarding the process mentioned in Section 3, the existence of safety requirements is not a factor eliminating the proposed process from use. Requirements specification, prepared by the requirements engineer, is the input for a given process. This means that safety requirements may still be satisfied with the proposed process as their identification takes place before the process begins, along with safety test scenarios. What is more, the proposed process helps ensure that any requirement, including safety requirements, is reflected in the final software product.

Compared with a standard AGILE process, it is also worth mentioning that the adaptation is achieved differently. Adaptation is understood as the ability to learn from experience. The standard AGILE approach assumes that adaptation is ensured by frequent interactions

with customers/end users, which is not a common situation in many research projects in which we have been involved. The proposed solution enables adaptation by ensuring transparency that supports inspection, as mentioned in Section 3.2.4. The adaptation is achieved by self-concluding all the struggles and obstacles, and understanding the problem and solution during the retrospective activity described in Section 3.2.4. The adaptation is supported by a specific sequence of activities, tools and attributes, as mentioned in Section 3. After each iteration, developers knew more and better understood the solution, which resulted in better decision making and choosing better solutions in the next iterations.

While comparing the mentioned process with the plan-driven approach, we need to highlight that the product backlog mentioned in Section 3 is a kind of a plan that drives further work.

In summary, the software development process proposed in Section 3 may be used to develop software, no matter whether a higher level process (project process, systems engineering process, etc.) is adaptive or plan-driven. The only difference will be visible in the product backlog and if it will be filled once in the project lifetime (when a higher level process is plan-driven, and assuming no changes are introduced during execution) or iteratively in small portions (when a higher level process is an adaptive one).

In Sections 2 and 3, we will discuss the general process, the basic concepts and the objectives of any process. Then, in Section 4, we will explain how the software development process was designed on the example of a particular drone-related project.

We will analyse the process, what problems were encountered and solved and what can be performed better to improve this process in the future. In Section 4, the standard and dedicated metrics used to assess the process quality will be described, followed by the conclusions.

2. Process Fundamentals

2.1. Generic Process Fundamentals

A generic process is defined in [12] as a “*set of interrelated or interacting activities, which transforms inputs into outputs*”. Activities mentioned in the process definition, as defined in [13], are a “*set of cohesive tasks of a process*”. Going a bit further and making the definition more “elastic”, Prince2® [14] clearly states that this set of activities is designed to meet a given objective. It gives some freedom in understanding the idea of the process and leads to thinking about it as something we can create or change depending on our needs. Tailoring, as addressed in [9,15], is a process that allows the adaptation of standards or somehow defined processes to make them more suitable for the environment and needs.

Each process can be measured as mentioned in [12,16] and has a set of basic metrics that constitute values for process assessment. Regarding ISO 9000 [12], there are two metrics as follows:

- Effectiveness of process: ability to achieve desired results;
- Efficiency of process: results achieved vs. resources used.

But, in contrast, the ECSS-Q-ST-80C [16], which is targeted to software product assurance in space engineering, defines the following two basic characteristics of a process utilised to develop and maintain software products in space applications:

- Duration: how phases and tasks are being completed versus the planned schedule;
- Effort: how much effort is consumed by the various phases and tasks compared with the plan.

We can see now that metrics used to assess the process may vary and may be derived from constraints such as standards to which we must adhere.

The general idea of a process is shown in Figure 1.

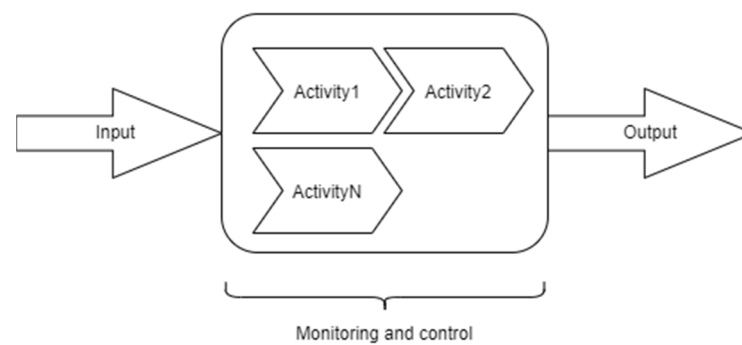


Figure 1. A general idea of a process.

As stated in ISO 9000 [12], inputs and outputs to/from the process may be tangible and intangible. It may be any formal or informal document, source code, executable software, materials, equipment, information, energy, etc.

That is the general idea of a process. More about the benefits of the process approach, linking and implementing processes, etc., may be found in [12]. Instead of a generic process, we can find a variety of process type forms. Each has its objectives, requirements, activities set and types of inputs and outputs. For example, project processes may be found in [9,17], processes related to software development in [13,18–21], processes related to systems engineering in [15,22] and quality management/assurance in [1,16].

2.2. Why Is Process So Important?

Martin Fowler, one of the authors of the AGILE manifesto [23], in his article published in 2019 titled “Is high quality software worth the cost?” [2], defined an internal and external dimension of the quality. He mentioned that internal quality is something that the customer cannot see but is visible and experienced by developers. Also, high internal quality brings many more benefits than external quality, no matter the initial slowdown at the beginning of the project. It is compensated later. Robert C. Martin, in his book [3], mentioned that software is not only about writing source code, and highlighted the big meaning of software architecture in ensuring software quality. In the foreword of quality standard [1], it is stated explicitly that quality management of the software process is crucial to achieving software quality, and such a process must be planned, controlled and improved. It is obligatory for the military drones’ software.

The Fowler’s text mentioned at the beginning of this chapter leads to the conclusion that process is the key to the success of a software development process. In 1992, NASA wrote in their recommended approach for software development [24], that one of the keys to success is to develop and adhere to the software development plan (SDP). The software development plan is addressed in [19,20,24–26] and is a document that describes the development environment and development activities and its relationship/process, etc. It addresses the plan for software development and its process description as its main part.

Another confirmation of the process meaning for the project can be found in the INCOSE SELI guide [27]. One of the leading indicators in systems engineering is “*process compliance trends*”. The explanation given to this indicator clearly states that poor or inconsistent systems engineering processes or weak adherence to defined processes increase project risk.

One more proof for the above claim is presented in the AGILE PM handbook [17], which states clearly that quality corresponds to both the range of features delivered and their technical quality. This is all sealed with the statement that the process is imperative to achieving quality.

Now, let us look closer at the software development process in the GNSS-denied navigation system development project.

2.3. Software Development Process

First, let us explain what the software development process means from the system engineering and software engineering point of view.

The software development process and its activities are documented in [13,18]. Some more specialised processes are targeted and tailored to the specific needs of a given domain. For instance, the considerations regarding the software development process for aviation are documented in [6]. They should be used for drones with a maximum take-off weight between 150 and 20,000 kg intended to operate in a non-segregated airspace [28]. Considerations for space applications can be found in [16,20,21,24].

We can sum up that any software development process usually concerns the following aspects of software engineering [21]:

- Requirements definition;
- Design;
- Production (implementation);
- Verification and validation;
- Transfer (installation);
- Operation and maintenance (utilisation).

Sometimes, other aspects are also considered during the software development process, for instance, as addressed in [6], software planning, certification, integration or disposal [13]. They are the key aspects of the certified drones applied in civil and military areas. Some processes, like quality management or configuration management [15], are also considered during software development. How much support processes are incorporated into the software development process designed for the project depends on the need (contract regulations, company rules, etc.) and tailoring [9,13,15] associated with it. Knowledge areas like quality management and assurance [16], version, configuration or change management [1,18,29] are also within the scope of software engineering [18] and, thus, shall not be omitted while considering a software development process. As mentioned in [6,13,15], they are crucial to achieving an acceptable quality of a product or service and then for certifying the software and the drone.

3. Software Development Process Given for Consideration in This Paper

The designed and described process analysed in this article is not a standard adaptive process but is tailored to the specific environment and needs. The “end user” of that software is an autonomous drone, capable of operating for hours in GNSS-denied conditions.

This was an AGILE [8,9,17,22,23] -based process, driven by three scrum pillars [30], as they were considered valuable and helpful in the target environment. The main factors that dictated the implementation of the identified process activities, which had a big impact on how the process looked, were cumulated around the following:

- Current work style;
- Company internal regulations;
- Team experience and attitude to adhering to established processes;
- Different locations of teams (teams were not working in one place);
- Research character of a project that was burdened with a dose of uncertainty regarding the direction of understanding the problem; product development started from Technology Readiness Level 4 (TRL 4) [31] and was targeted to achieve Technology Readiness Level 6 (TRL 6);
- Knowledge maturation regarding the problem to be solved and possible solutions that may significantly change current ideas and draft solutions that may generate frequent changes;
- Plans regarding project continuation in the future and development of the product to a higher Technology Readiness Level [31].

More about the presented and discussed project may be found in Borodacz et al. [32] and Pogorzelski et al. [33].

The factors mentioned above are justification for choosing an AGILE-based process over a plan-driven one as a base for development. As stated in [9], while selecting a development approach, *product*-, *project*- and *organization*-related factors should be taken into account.

The most important factors from those mentioned above that were decided in our case were as follows:

- A big dose of uncertainty related to the research character of the project; in fact, our knowledge greatly matured during development, which might cause a lot of unexpected changes;
- Low team experience in following established process and reluctant attitude to follow.

In fact, only two of the three groups of factors mentioned in [9] had a significant impact on the decision, as factors related to *the project* group were stable enough to put them in second place. To find the justification for using three scrum pillars, please see Section 3.2.4. In general, the decision that stands behind those pillars is cumulated around *adaptation*, which allows us to be smarter in the future by learning from the past.

3.1. Aspects Considered in the Process for the GNSS-Denied Navigation System Development Project for Drones

There were many aspects and issues to consider including in the drone-dedicated software development process chosen as an example project. These were as follows:

- General standard development approach to use as a base for tailoring: a good software engineering practice used in the company;
- Software requirements: elicitation/decomposition/definition;
- Quality characteristics of the desired product: functional completeness and correctness, time behaviour, resource utilisation, analysability, modifiability and testability, as described in ISO/IEC 25010 [34];
- Software design: a good software engineering practice used in the company;
- Software implementation: embedding that software into the onboard navigation computer;
- Software verification and testing: software-in-the-loop and hardware-in-the-loop laboratory tests;
- Standards to consider: DO-178C [6];
- Process metrics: individual velocity, team velocity, process suitability in a given environment;
- Tools that support activities: Gitlab and Enterprise Architect;
- Transform software requirements into “issues” [35] or tasks to ensure the product is complete and traceable;
- Defects and errors management to ensure every defect and error found is addressed in the software;
- Version and release management to enable control and product management.

The drone-dedicated software development process and its details were addressed in the software development plan (SDP). It was prepared before the development activities began in the project planning phase [9] to reduce the overall risk associated with the software development activities. Now, we will analyse the software development process in detail.

3.2. Software Development Process and Activities in the GNSS-Denied Navigation System Development Project for Drones

After considering, many factors like requirements certainty, scope stability, ease of change, etc., that helped choose the project approach mentioned in [9], the decision was made to apply an adaptive approach over a hybrid or plan-driven (sometimes called “predictive”) approach for software development, with some modifications that will be addressed further in this chapter. According to this paper’s title, we will discuss the process for drone-dedicated software development only, not for the entire project. The approach for

the whole project was the subject of a project management plan (PMP) or project plan [36] and is beyond the scope of this paper.

Regarding the software development process for drones, even though the software requirements [13,37] and scope were stable and were derived from the system level [15], the degree of innovation was high. Therefore, there needed to be a “place” left for changes as the developers’ knowledge matured with the product development and as the software was the main and key part of the product (the drone). The development team size was also considered, and it was below 10, which was suitable for an adaptive approach [9,30]. One of the objectives of the software development process was to reduce the risk associated with the development activities [27]. Another was to introduce resilience by providing the team’s ability to adapt and respond quickly to uncertainty related to high product innovation and new process approaches.

The general view of the software development process for drones implemented in the project is shown in Figure 2.

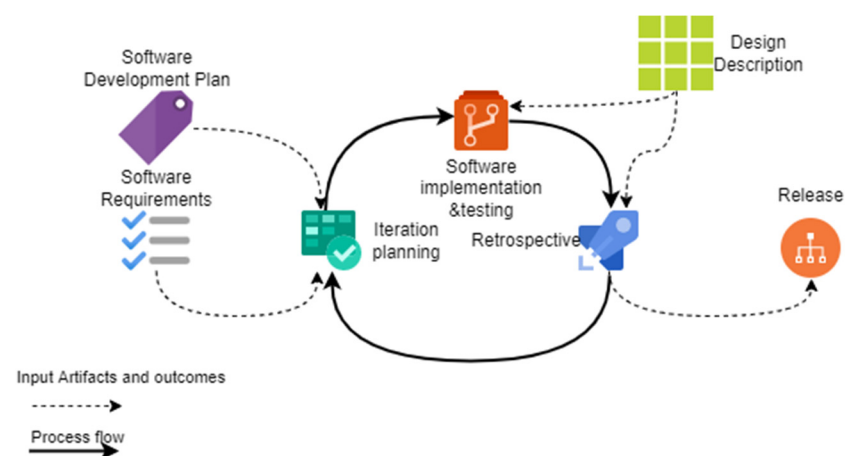


Figure 2. The software development process implemented in the project.

The process was focused on three basic activities: iteration planning, software implementation and testing (coding and tests) and retrospective [17,30] (a review and conclusion). Input data that were needed to perform the process activities were the software development plan (SDP), software requirements and design descriptions of components developed during the project as software solutions. The outcome of the process was the software release.

The applied process, in general, was an adaptive one, based on three important scrum pillars: transparency, inspection and adaptation [30]. These pillars formed the basis for every activity within this process.

3.2.1. Preparation Activity

Software requirements taken as input to the process were already prepared and somehow stable (at least 75% of the expected number) before the software development process activities began. That gave a better view of the development scope and allowed us to plan the tasks and support ensuring product completeness by providing traceability (task to requirements and vice versa). Software requirements were then transformed into “issues” (Issue definition may be found in [9,35]), which represented tasks to be completed. The term “issue” corresponds to the term used in a tool chosen to support the example process and will be explained later.

Functional requirements defined in [15,18,29,37] were first represented by issues whose names corresponded to the name of a function represented by such requirement and were given a special label, marking the issue as a function like “function 1, function 2, function n”. Labels were also a consequence of a chosen tool, with each issue having a requirement, as a parent gives a special link to the issue representing a function to ease the tracking of function development progress. In fact, in software development, we provide

functions, but performance requirements, as highlighted in [15], are a special kind of quality requirement that corresponds to the quality of these functions that deal with values like time, volume, frequency, etc. The number of functions we need to implement in software depends on how many functional requirements we have allocated to the software. The non-functional requirements [15] tell us about the quality and degree of complexity of implementing such functions, as pointed out in [38]. Non-functional requirements like quality requirements [29] may have their source, for example, in specialty engineering areas [15] like safety engineering, reliability engineering, etc., or in quality models and quality characteristics related to those models like those defined in [16] or [34].

ECSS-Q-ST-80C [16] clearly defines a “quality model” as a set of characteristics and relationships between them. A quality model is used as a basis for specifying quality requirements and evaluating quality.

Key software quality characteristics identified for software developed during the example project were based on ISO/IEC 25010 [34] and were mainly as follows:

- Functional completeness and correctness (from the functional suitability group);
- Time behaviour and resource utilisation (from the performance efficiency group);
- Analysability, modifiability and testability (from the maintainability group).

Each characteristic should be understood in terms of a product under development and its context (for context definition see [29]) or intended future use. It should be transformed into requirements and then reflected in the system/software design. It is a common practice to use some pre-defined patterns that support different characteristics related to software architecture, as mentioned in [7]. **The process described in this paper goes further. It was designed to ensure these characteristics are reflected in the final product.**

This was a useful consequence of the three valuable scrum pillars mentioned in [30]. The process supported achieving these characteristics by providing means of facilitating transparency and inspection. Adaptation was achieved by self-concluding the results. It mainly took place in the retrospective activity (described further). The means provided by the process to support product quality in terms of mentioned quality characteristics were as follows:

- Task traceability to requirements and vice versa (to support product functional completeness concerning software requirements specification);
- Deriving task acceptance criteria from parent requirement (to support performance efficiency assessment and make sure that the final product reflects its specification, which supports a partial software verification [15,39]);
- Provide unique identifiers related to software products [13], their version, branch, iteration and documentation set to support error and defect management and project completion and to ease analysis while looking for errors or estimating the impact of proposed or required changes;
- Binding branch and task identifier to a specific part of source code developed during iteration to support analysability, which limits the inspection only to that part of the source code;
- Unifying work style and language usage (modifiability and analysability), in part not dependent on system/software design, supported by the process itself. Addressing coding standards should ensure that everyone in the team “speaks the same language” and can support product development or modifications of source code prepared by somebody else from the team without having doubts about the meaning of variables and so on.

Functional requirements may be represented as a use case if there is an interaction with an actor as in [40–42], user story as in [17,43–45] or traditional requirement as in [29,45]. User stories may contain some additional information about quality or acceptance criteria and are prepared following INVEST rules [17,43] rather than the quality criteria mentioned in [1,18,37,39,45] for traditional requirements. They shall be transformed, decomposed to other forms and supported by additional information to facilitate analysis or traceability as

they may not provide full or detailed specifications [45]. Some more discussion about the meaning of requirements in the project can be found in [15].

After this “one-time” activity (assuming all software requirements are prepared before the process starts; otherwise, this activity is repeated for the rest of the requirements) the list of tasks that constitute a product backlog [22,30] is prepared and iteration planning may now begin.

Although the process-assumed requirements are stable before entering the first activity in the process, the process may trigger a need to return and refine requirements on every level, as the knowledge about the problem and possible solutions mature during the project (as a tool of “adaptation” [30]). This is not marked explicitly in the software development process itself, as this is rather the matter for the configuration/change management process [1,6,15,29,46].

A software development plan is one of the inputs needed by the process to facilitate giving some additional (of little importance for this article matter) parameters/data to tasks selected in iteration planning. It is to simplify traceability, task management, results reference (for example, a reference to a file containing a comprehensive report or other information needed to complete the task, etc.) and tracking relationships between tasks (if any).

3.2.2. Iteration Planning

The duration of iterations in an example project was two weeks. This was enough time to see progress but not to allow the work to proceed in the wrong direction. The company work style and the need for project and software development monitoring were considered to set up iteration duration. Each task was created in a way that allowed work to be performed in these intervals. The tasks too big to be implemented in two weeks were split into smaller tasks. After preparing a product backlog that way, there was a better possibility of estimating the overall software development duration in iterations or weeks. This approach also facilitated progress monitoring with a burn-down chart [22] to correct estimates and determine the real velocity of the team in implementing issues. The velocity was determined by measuring the number of issues closed per iteration and was used to adjust the number of issues to be implemented in the following iterations. Productivity could be measured by dividing the velocity by the number of team members [22]. Productivity was out of our point of interest. More details on utilised process metrics can be found in Section 4.

During the iteration planning, each team member responsible for source code development chose tasks to perform during the iteration. That way, the team created an iteration backlog.

The number of tasks chosen by one person in the current iteration depended on the historical data (if some were already collected from previous iterations) and the estimated time needed to complete the task. Although tasks were prepared to last up to two weeks duration, some of them could be much simpler to complete and could take, for example, a day. Everything depended on the parent requirement’s complexity and understanding of the requirement by the developer during transformation. A few initial iterations may not be optimally planned due to a lack of historical data regarding the team (or person) velocity. This would be unveiled after a few iterations and corrections.

After selection, each selected task was given its acceptance criteria [43,44] or definition of completion [30] to facilitate effort assessment and provide a common understanding of what was the limit of the task and what results allowed the task to be marked as completed. It facilitated transparency, inspection and adaptation [30] regarding achieved results and related thoughts/conclusions.

The number of tasks selected by individual team members depended on the time declared for this iteration, the level of task complexity and the developer’s skills. The time available for the current iteration was determined before selecting tasks. Time expressed in hours during the iteration depended on the work week length and the engagement of individuals in other projects.

The iteration backlog was the output being entry data for software implementation and testing activity.

A “zoom-in” into iteration planning activity is shown in Figure 3.

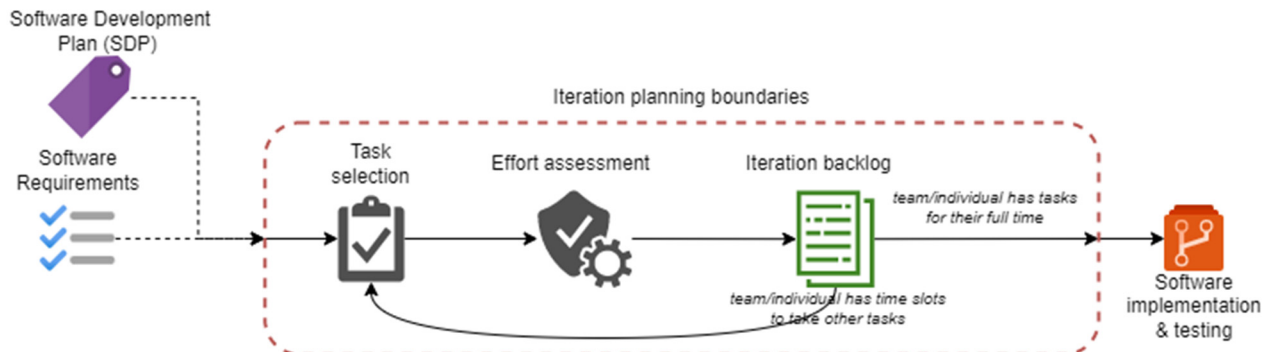


Figure 3. Iteration planning activity.

3.2.3. Software Implementation and Testing

Software implementation and testing activity was essential during the transformation from specification and design into a tangible item. It was composed of a few tasks as shown in Figure 4.

Note 1: Diagram was prepared assuming that TDD is not practiced.

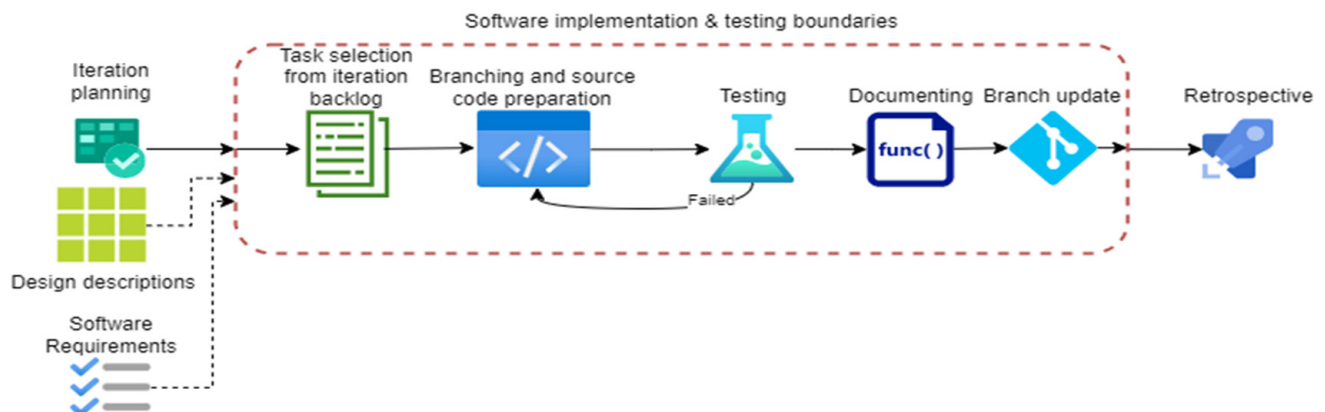


Figure 4. Software implementation and testing activity.

After initiating the activity, each developer chose a task from the iteration backlog. The software development plan (SDP) assumed that there should be only one task under implementation at a time for an individual to minimise the risk of errors implemented in the source code. This depended on how the tasks were created and if there were no related tasks that should be at least partially completed before to enable tests for the current task. Each developer followed the path on his own and independently from the others. Fortunately, in the example project, teams were organised so that one developer was responsible for one software component [16], for example, one software application.

After the developer decided which task from the iteration backlog to implement first, the branch (For a definition of branch see [47]) was created. The term “branch” is related to the git version control tool (<https://www.git-scm.com/>, accessed on 23 September 2024). The general idea of branch management in the project is shown in Figure 5.

Note1: "Tag:" is put in the diagram only to visualize next software release.

Note2: New branches within iteration are created from previously completed but not merged to "main" yet branch as show in the diagram.

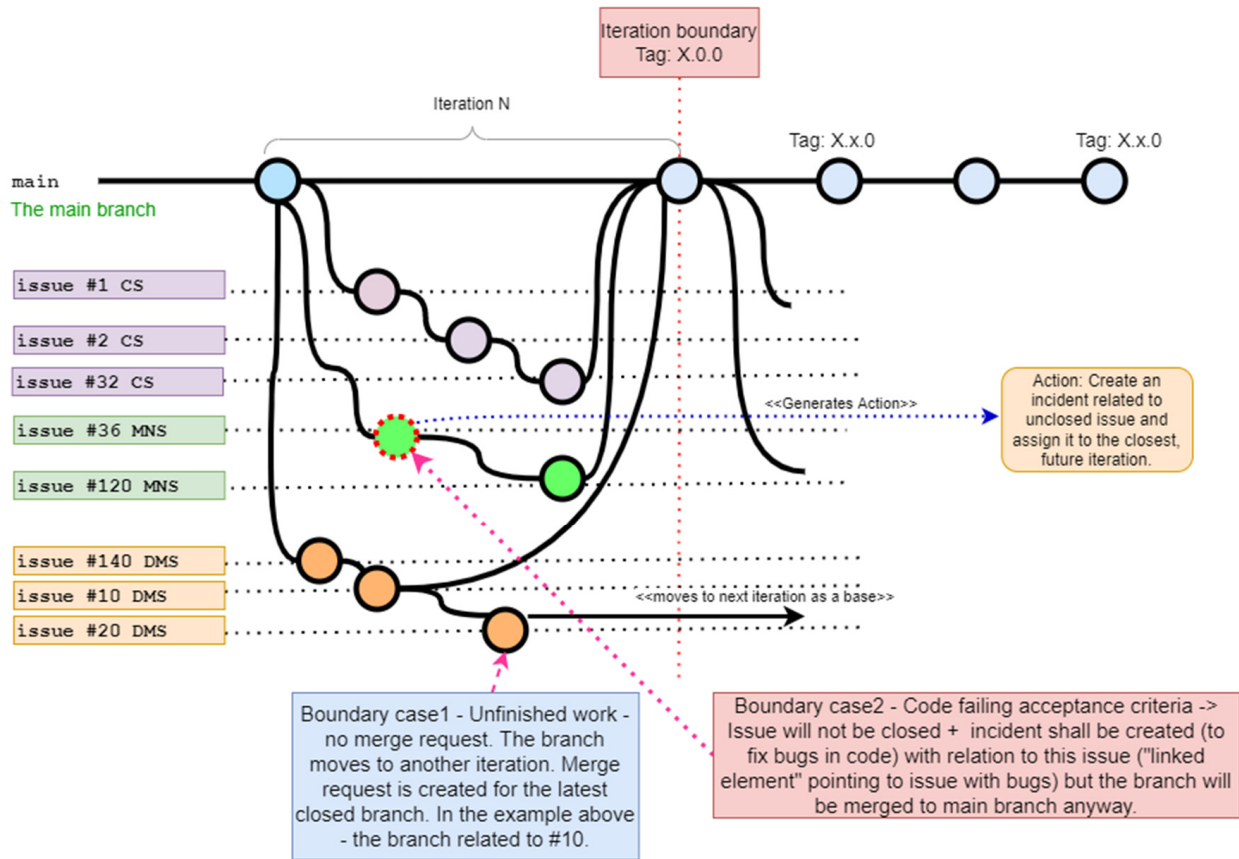


Figure 5. Branch management.

There was one main branch foreseen to store the source code. It was accepted in the retrospective after each iteration (the next activity in the process). Each developer responsible for the chosen task created a new branch from the main branch at the beginning of the iteration. Before the first iteration, the main branch stored an empty repository structure. Each developer needed to clone and organise a project in the integrated development environment (IDE) of choice to fit the repository structure. The repository structure is shown in Figure 6.

After preparing the project in the IDE, each developer had to create a branch with an identifier designed for error or defect tracking and task management. Identifier details have no meaning for this article, so they will not be discussed.

Each branch corresponded to a different issue. To facilitate repository and task/issue management, the GitLab tool (<https://about.gitlab.com/>, accessed on 23 September 2024) was selected. Selection criteria are beyond the scope of this paper, as the tool is only a means to facilitate activities but is not, and should not be, the core of the process. Tool selection criteria were also dependent on company factors. In the GitLab, issues are marked with “#” followed by a number pointing to a particular unique issue number. Corresponding with Figure 5, “issue #1 CS” and similar text in rectangles on the left mean different branches corresponding to all issues. “CS”, “MNS” and “DMS” that appear in this paper are acronyms of the full names of software components developed during the example project and have no meaning for this article.

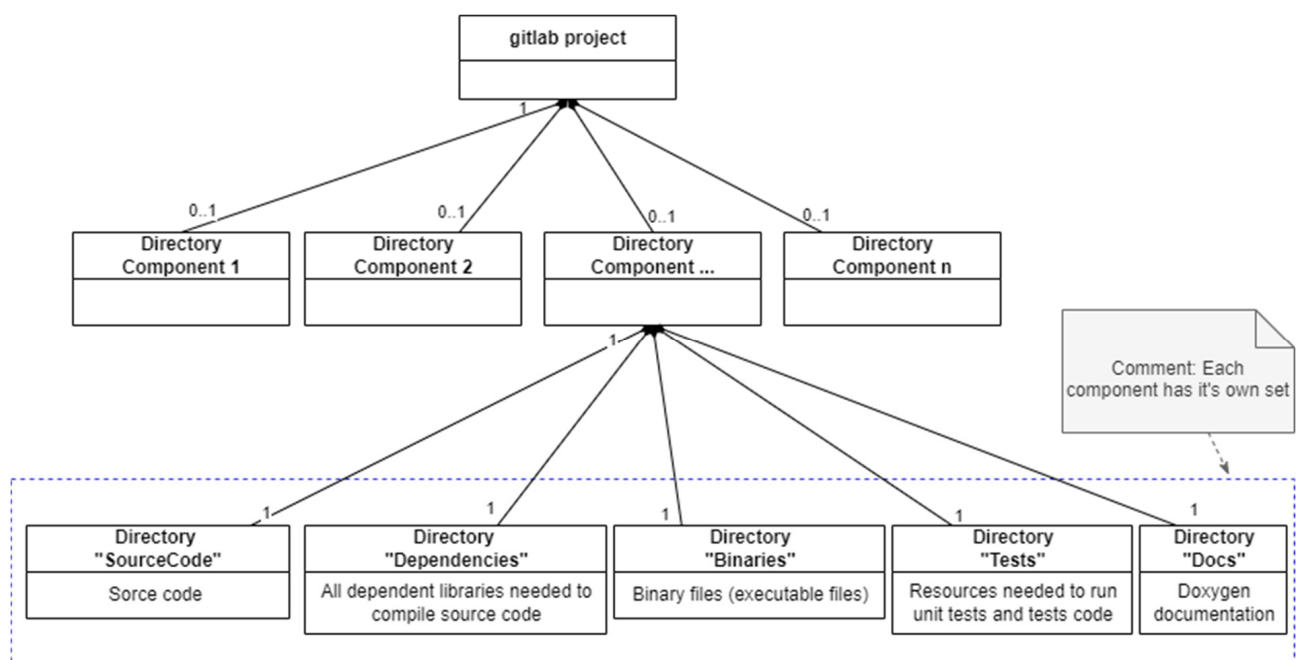


Figure 6. Repository structure.

After having work completed, or at the end of the iteration (during retrospective described further), the last created and completed branch (meaning that the corresponding task was also completed) was merged to the main branch that stored the up-to-date and working code. Splitting teams where only one team/individual (as a team) was responsible for their component and organising repository, as shown in Figure 6, allowed to minimise the number of merge conflicts depending on team size and internal communication. If the task was not completed, or its results did not fit the acceptance criteria as pointed out in the iteration planning activity, the branch was recognised as unsuccessful and was moved to another iteration. In such cases, the last successful branch was merged with the main branch to ensure that only the working code was stored. Before a developer can recognise the source code as the working code that meets acceptance criteria, unit tests [16,24,43,44] need to be run. For the first time, the process was in its draft release, a test-driven development (TDD) [48] was considered, but was finally rejected due to the current teamwork style and many other new challenges that the team needed to face. It required some new habits and energy to become used to thinking about testing code that did not already exist. There are some benefits of such an approach, as addressed in [8], and it should be considered in the future, especially when we need to have the “courage” to implement changes in the code.

Having tasks traced to requirements and having acceptance criteria derived from parent requirements, partial software verification testing [15,29,39] at the component level was executed simultaneously with unit testing.

Regarding the repository structure in Figure 6, the diagram was simplified, making it easier to understand for people unfamiliar with UML [40] or SySML [41]. In the plot, the composition relationship [40,41] is shown (an arrow with a filled diamond on one end). It means that a directory pointed by the diamond consists of directories on another end; zero/one or one that are visible on the arrow ends are the multiplicities that point that one upper-level directory can be composed from zero to one directory or exactly one lower-level directory (subdirectory). Describing the diagram, we can say that the “gitlab project” directory may have from zero to one subdirectories named Component 1, Component 2, etc. During the example project, Component 1, 2, etc., corresponded with the name of the software component that was to be developed during the project.

Coding standards like [49] were addressed in the SDP to facilitate further product development and modifications by different teams or individuals. The source code was documented using the Doxygen (<https://doxygen.nl/>, accessed on 23 September 2024) file generator and Doxygen comments in the code to allow automatic document generation.

3.2.4. Retrospective

Retrospective [9,30] meetings took place after each iteration. There were frequent meetings performed on the last day of ongoing iteration. Their goal was to conclude the job performed during iteration, to check together achieved results from their acceptance criteria or definition of completed point of view and to talk about the future improvements of the process or technical performance of the product. Additionally, the meeting helped to identify gaps and obstacles that stopped us from doing something (for example, IT environment limitations, missing tools, etc.), and to address proposed changes. The goal of retrospective meetings in the drone software development process was to achieve inspection and adaptation.

According to the SCRUM Guide [30], we should be transparent to ensure all team members understand the expectations, the goals, the processes we are going through and their role in this “journey”. The more the transparency, the better for inspection. Then, we can inspect. If we inspect, we can conclude. If we conclude, we can improve and learn. Then, we adapt. There is no adaptation without transparency and inspection.

The relationship between these pillars is visualised in Figure 7.

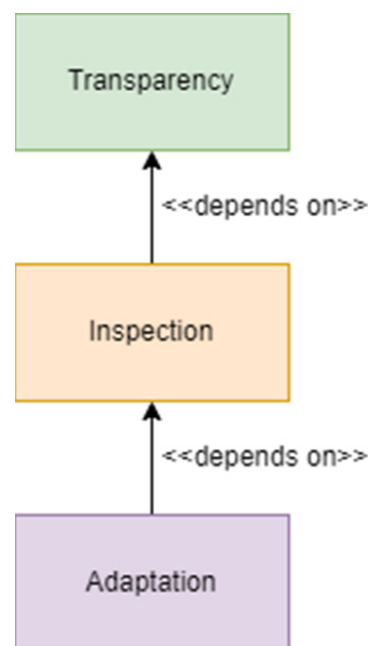


Figure 7. Relationships between three scrum pillars.

The relationships clearly pointed out that transparency was the key to achieving the other two values. The process defined in this paper supported transparency and inspection in some aspects.

Retrospective meetings, except for talking with the team, concluding together, discussing and resolving or addressing problems, were conducted as shown in Figure 8.

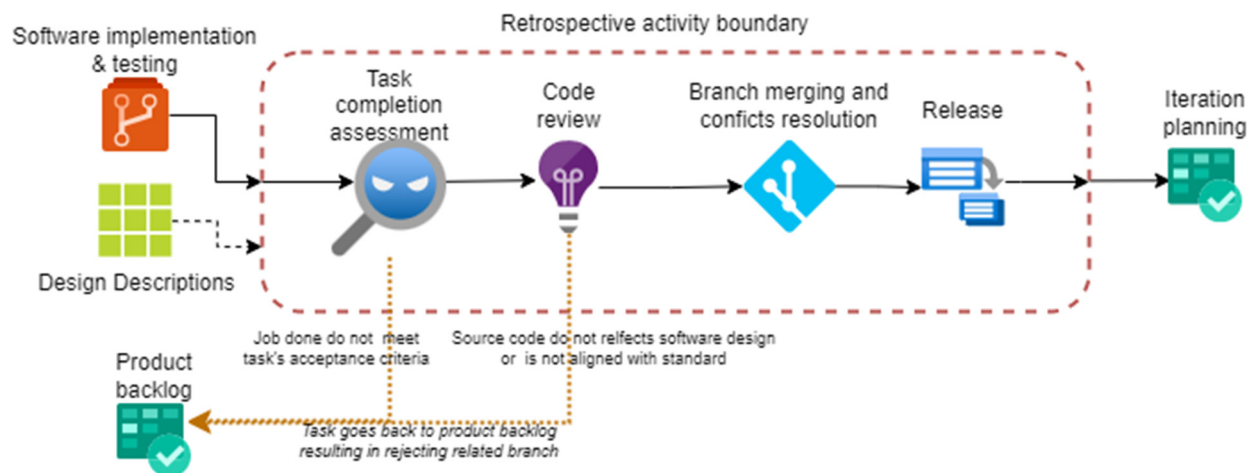


Figure 8. Retrospective activity.

The first task in the activity was the project's task completion assessment. This task was based on an interview with the developer responsible for the project's task and mutual verification of achieved results.

The general acceptance criteria for every projects' task, no matter if some of them were derived from the parent requirement or not, were as follows:

- Code is compiling and working;
- Code complies with coding standards;
- Tests are performed on the code and passed;
- Code is documented with Doxygen and the updated document is placed in a dedicated subdirectory in the repository;
- Code is aligned with software design; if not, the design is already updated before finishing a task.

If the job performed within the task met its acceptance criteria, the task was assumed to be completed and marked as finished. For the overall endeavour, not every task corresponded only with requirements or finished with a piece of source code being prepared. Some tasks depended on analysis that needed to be performed before source code development or were related to environment and hardware resource preparation. These tasks had additional acceptance criteria defined while taking the task to the iteration backlog. Such tasks did not generate a new branch but were bound to another task that ended up with the source code to facilitate relationships, work completion and error tracking. If any task was assessed as not meeting the acceptance criteria, it was moved to the product backlog and it was preferred to be finished in the very next iteration if possible (sometimes, to complete the task, we needed to perform other tasks first, but this relationship was invisible at a glance). In Figure 8, a task completion assessment is divided into two parts: task completion assessment and code review. If a task was not related to a source code development, the code review was not applied. The process assumed that the number of such tasks was limited only to a few. That is why it is not highlighted in Figure 8 with a separate path bypassing code review and other tasks in the activity.

If a task was completed, its branch may be merged into the main branch. If any conflict appeared during the merge, it was resolved by the developer or repository manager as fast as possible to enable the branch management strategy realisation, as shown in Figure 5. If branches were merged successfully, a software release with dedicated version tags was prepared and contained a working or accepted piece of software product [13] (code, related documents, etc.). The release may constitute a return point or be passed to further tests if eligible.

3.2.5. Problems Related to the Proposed Solution

There is no rose without thorns. Although the process was designed to support many aspects of drone software development in the target environment and for specific needs, the solution needed further improvement based on data and collected conclusions. During the drone software development process, we encountered a few problems or obstacles that influenced overall process performance and made it difficult to follow. Each identified problem was given an identifier P1, P2, etc., to support referencing in further text. Problems were mainly related to the following:

- P1: new task being created “on the fly” in an advanced development phase.
As the team consisted only of software developers, it was not easy to foresee and prepare correct and stable software requirements before development started. Of course, uncertainty is the source of changes. Uncertainty is related to the project goal and other factors described earlier in the paper. There are still functionality and quality elements we can identify and freeze earlier to minimise the risk and number of changes. This may be pointed out by project objectives [9,50] and key performance indicators (KPI) related to it [9,22] or measures of effectiveness (MOE) [15,51] or measures of suitability (MOS) [15,51]. As claimed in [52], developers often fall into an “implementation trap” [52], focusing on solution space rather than problem space, so requirements prepared by such individuals caught in such a trap are not abstract (free of implementation) [53] and they need to be changed later.
- P2: new tasks (incidents) being created due to a given software release failed tests.
This problem was related to handling new tasks (“incident” type in our version of GitLab) that were created due to failed tests performed on a given software release, not failed during unit tests conducted by the developer himself. The hard part was how to address changes in ongoing iteration if it appeared that correction was crucial for not delaying other tests planned for a given time and not to break the currently opened branch to avoid merge conflicts.
- P3: Applied branching strategy required discipline to be followed by individual developers. Not following this led to merge conflicts and missing or overwriting code problems.
- P4: Choosing tasks to complete in “free order” impacted the test plan and schedule slips. If the test plan for example foresaw that function A or function B was a subject of a particular test run [44], this could require removing that function from the run or changing the schedule if at least one task corresponding to a function (as described in Section 3.2.1) could not be completed on time. To avoid this, the process required providing a little bit of “control” in deciding which tasks were to be performed in the following iteration.

These problems were encountered during the project performing in a specific environment while developing software for the drone navigation system. Other issues may appear and disappear or their impact may be negligible in different environments.

4. Process Metrics

As mentioned in the introduction, one of the most important matters for us was to break a mental barrier characteristic of the target group (the software developers). They preferred to develop the software in a free, unstructured and uncontrolled way, with reluctance to follow any formal process. Because of that, we had to find how to break all the barriers. To be sure that our solution progressed in the correct direction, we worked out two process metrics to measure the degree of process suitability in our environment. Those metrics will be explained further in Section 4.2.

4.1. Standard Metrics Used to Evaluate Process

The standard metrics applied to evaluate the process focused on two values used to support future estimations and planning future iterations. These were as follows:

- Individual velocity (V_{Ind});
- Team velocity (V_T).

Individual velocity was calculated by dividing the number of closed tasks by one person in a given iteration and it was expressed in [tasks/iteration]. It was useful for estimating the time remaining to close all tasks related to the software components assigned to an individual and to correct time estimations needed to complete tasks in an iteration. It also helped to decide how many tasks an individual could take for one iteration concerning the declared time available, as mentioned in Section 3.2.2:

$$V_{Ind} = \frac{\sum_0^i T_{cInd}}{i}$$

where

i = number of closed iterations;

T_{cInd} = number of tasks closed by an individual in closed iterations;

V_{Ind} = individual velocity expressed in [tasks/iteration].

Individual velocity (V_{Ind}) was influenced by problems $P3$ and $P4$ mentioned in Section 3.2.5. Not following the branching strategy (problem $P3$) made it more difficult for developers to complete tasks, as they required more time to resolve conflicts or missing code. Choosing tasks to complete in free order (problem $P4$) resulted in work being blocked by the need for another task to be completed first (e.g., due to lack of data source or entry point).

The total number of closed tasks equalled the number of tasks closed by any individual team member. Team velocity was calculated by dividing the total number of closed tasks by the number of closed iterations:

$$T_{cTeam} = \sum_0^n T_{cInd}$$

where

n = number of individuals in the development team (team size);

T_{cInd} —as above;

T_{cTeam} = total number of tasks closed by the team in closed iterations.

Then, the team velocity (V_T) was computed with a given formula:

$$V_T = \frac{T_{cTeam}}{i}$$

where all symbols are explained above.

The number of iterations left (I_{left}) to complete the project was estimated as follows:

$$I_{left} = \frac{V_T}{T_o}$$

where

V_T is as above;

T_o is number of open tasks left.

The number of iterations left to complete the project was calculated as required or when significant changes in team velocity were noticed. Team velocity (V_T) helped estimate the overall time needed to complete the project. Problems $P1$ and $P2$ mentioned in Section 3.2.5, unfortunately, impacted estimations because the number of opened tasks was rising, making previous estimations invalid.

Low individual or team velocity values had an impact on estimations alerting possible schedule slips. We could calculate the required team velocity by dividing the number of

tasks at the beginning of the development by the time to complete them all expressed in iterations (if we needed to determine the required number of tasks completed per iteration). See Figure 9 for a visualisation of the actual team velocity impact on schedule. As the proposed process preparation of tasks to be completed was assumed to be a single-time activity performed before development began (see Section 3.2.1), it was easy to calculate the required team velocity, as the number of tasks to be completed was considered stable or stable in the most part (see Section 3.2.1 and assumption about % of stable requirements). Given that, problem P1 related to creating new tasks on the fly was a little bit of a challenge, as also mentioned in Section 4.2.

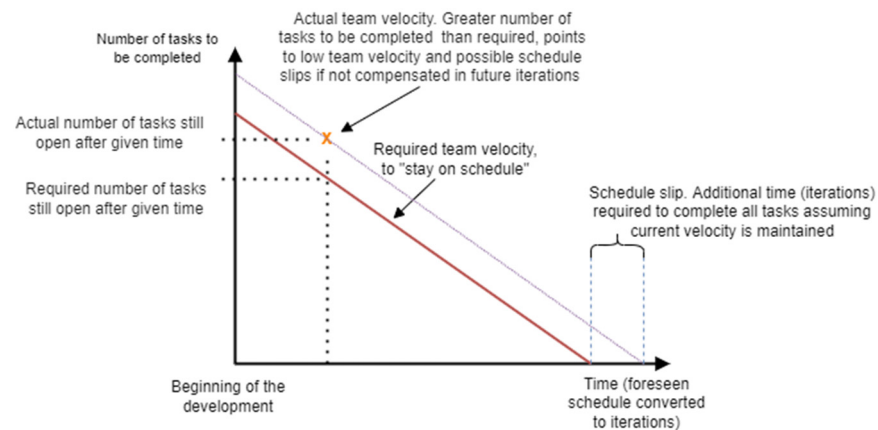


Figure 9. Required vs. actual team velocity. Impact on schedule.

Schedule slips in the mentioned GNSS-denied navigation project targeted for drones were a kind of a trap. As the developed solution was targeted at drones, it was also planned to be tested on a drone. This, however, was dependent on weather conditions. Moving a schedule further toward months characterized by rains and strong winds would result in a much bigger schedule slip than estimated.

Not only schedule slips were the result of low team velocity. As the scope of our project was the minimum one to satisfy the project goal, there was no possibility to cut tasks to stay on schedule. Developers were then becoming quite nervous. This phenomenon had an impact on the quality achieved during task completion, which, in many cases, resulted in problem P3 or additional correction needed before marking tasks as completed. To stay on track, and to increase team velocity, we introduced supporting activities mentioned in the Section 5. These activities also had a positive influence on the metrics proposed in Section 4.2.

4.2. Proposed Metrics to Evaluate Process Suitability in a Given Environment

For the software development process explained in this paper, we found a solution to assess whether the process was suitable for our environment. The process was executed in an environment characterised by its features mentioned at the beginning of this chapter. The project environment had a big impact on overall work organisation and success. Therefore, it was important to make sure the new approach was not a huge wave that would lead the team to a first struggle but that would help the team “surf on the waves”.

Our research on that topic led us to work out a metric that allowed us to address process suitability in our specific environment and to propose a three-degree scale to help us assess whether the process was friendly.

For the first, we needed to find a formula that helped us compute the degree of sticking to the process, because we noticed that not everyone was following the process exactly as planned. The formula is:

$$Stp = \left(1 - \frac{\sum_1^i e}{i * n}\right) * 100\%$$

where

Stp = degree of sticking to the process. Expressed in [%], rounded to integer type.

i = number of closed iterations.

e = number of individuals not sticking to the process in an iteration (estimated after iteration). If in a given iteration there are two individuals not following the process, the value is increased by two, if there are five individuals, the value is increased by five, and so on. For zero individuals, the value is not increased.

n = number of individuals in the development team (team size).

The numbers of individuals not sticking to the process were collected in each retrospective activity when it was found that some activities did not follow the process for any reason. It was summed up after a few iterations. This was not the number of mistakes made, but this value corresponded to the fact that an individual made any mistake or variation in a given iteration. Even if we observed that one individual made six variations or mistakes in a given iteration, the value increased by one.

There was no reason to calculate the degree of sticking to the process (*Stp*) in the few first iterations, as there should be some time to stabilise and become used to the new work style. We noticed that, in our case, we could achieve more reliable values of the degree of sticking to the process after three iterations (6 weeks). The information about the time required to stabilise and become used to the new work style may also be used to assess the learnability of the process and whether it needed any changes or improvements. This was, however, beyond the scope of our interest regarding the given process.

To support an assessment of process suitability in a given environment, we proposed a dedicated three-degree matrix regarding the computed *Stp* value, as shown in Table 1.

Table 1. Process suitability matrix.

1. <i>Stp</i> Value	2. $0 \leq Stp < 50\%$	3. $50\% \leq Stp < 75\%$	4. $75\% \leq Stp < 100\%$
5. <i>Sdeg</i> = Degree of process suitability 6. (short: “suitability degree”)	7. 1st degree—“Unsuitable”	8. 2nd degree—“Unresolved”	9. 3rd degree—“Suitable”
10. Value Interpretation	11. Process not friendly. Significant improvements or re-design required.	12. Further investigation is needed to find a reason or barrier that makes it hard to follow the process.	13. The process is suitable for a given environment. For values close to 75%, further investigation is required if following the formal process is very important for the project’s success or is required by the contract or any other formal agreement.

As a result of comparing the *Stp* value to given ranges, we obtained three possible **degrees of process suitability** (and a short name “suitability degree”): first, second and third degree.

The **degree of process suitability**, determined by the matrix presented in Table 1, helped us to improve our process and to take any supporting actions when needed. Supporting actions that impacted the *Stp* value are described in Section 5.

In our case, the value of *Stp* after three iterations (6 weeks), reached the third degree of suitability, but was close to 75%, which was finally increased by incorporating supporting actions described further.

The *Stp* value may vary during the project but should be considered stable if it stays within the given value range corresponding to a degree of process suitability (*Sdeg*).

The impact of the problems mentioned in Section 3.2.5 on *Stp* value was observed in a way that the more problems that developers encountered, the greater the reluctance there was. This resulted in workarounds that increased the number of individuals not sticking to the process in an iteration. The most significant impact observed was related to problem *P1*.

Creating many tasks “on the fly” very often made developers more tired of the work they did not want to do, which resulted in missing tasks and a greater list of new tasks added next time. In some cases, the task list was not updated for a few iterations, even until it turned out that the code performed functions that nobody ever talked about. This all made task creation activity not a “single-time” activity as assumed in Section 3.2.1, but forced developers to go through it many times, which increased their reluctance.

4.3. Tools Used to Collect Data

We needed to collect data required to use the formulas explained in Sections 4.1 and 4.2. Fortunately, the Gitlab tool helped us to determine the number of tasks closed and the number of closed iterations. Other data tools like MS Excel or Notepad were utilised to collect data related to observed activities pointing to any variations from the process.

5. Conclusions

The general idea of a process and its importance for the project was addressed in this article. The explanation of the drone software development process was presented and its implementation in the particular environment was comprehensively addressed. The pros and cons of the proposed approach were discussed. The addressed process was a unique approach performed during drone navigation software development in a specific environment. It grouped individuals reluctant to formalise any activity and without experience in following a planned process during software development.

Our environment, dominated by software developers oriented in source code development only, instead of the reluctance to formalisation, is also characterised by the lack of software architects and overall experience in developing software architectures. As a result, we found a solution how to ensure some important quality characteristics like functional completeness and correctness, time behaviour and resource utilisation, analysability, modifiability and testability by the process of design and tool utilisation. This enabled the product quality to be planned (to the mentioned extent), monitored and consciously controlled during product development without overwhelming team members. This was the reason why it was very important to us to make sure the process was accepted by our environment and, as a result, we found a metric that allowed us to assess this acceptance.

Traditional approaches, as documented in [6] or [13], assume that the verification process is the point where we verify whether we built the system right. This is usually performed after product realisation. The proposed approach moves the part of verification activities to the product realisation step, yet simultaneously ensures the traceability of tasks, branches and releases to source requirements. Deriving acceptance criteria from requirements to tasks acceptance criteria allowed marking a source requirement as “completed and verified” when all tasks related to a given requirement were closed. It resulted in much less time spent on system and components verification later.

Functional completeness and correctness, usually ensured by requirements themselves are ensured by the proposed solution by transforming lower-level requirements into tasks for the team. This ensures that no requirement will be missed or wrongly understood by the developer. Required time behaviour and resource utilisation is supported by deriving required values from lower-level requirements to task acceptance criteria. The task will then stay open until those criteria are satisfied. Modifiability and testability, although usually ensured by modular architecture and other architectural means, in the proposed solution are facilitated by ensuring traceability from releases, single branches, tasks, requirements and

related problems if any are reported. This approach significantly sped up error corrections or code modifications required, e.g., due to problem P1, creating tasks “on the fly”, as the developer responsible for a piece of source code was able to track all the way back to find an error or to assess modification impact.

The *Stp* and *Sdeg* values related to process acceptance helped us to assess the direction of process changes performed during the project. Those values are currently lagging indicators. The fact that the existence of a value shows if the process is more or less accepted, also implies that there are some environmental factors that decide whether a given process will be implemented in a given environment or not. These factors require further investigation and research and thus will be the subject of future work. Unveiling the factors will help design the process from the beginning to be easy to implement in a given environment so that the lagging indicator, uncovered in this paper, will only be a metric confirming the future design.

We observed that the significant positive impact on the final suitability degree (*Sdeg*) we achieved in our environment was made by supporting activities like:

- Increasing team awareness about the meaning of the process for the quality and highlighting the impact of popular mistakes or habits on overall work.
- Individual sessions or workshops after we observed any variation or mistakes;
- Big emphasis on leadership;
- Taking care and continuously improving transparency.

Concluding all the given knowledge stated in Section 2, and the experience described in Section 3, we may confirm that the designed process allowed us to achieve the desired results. At the same time, we can confirm that the process is imperative to enable and achieve quality. It addresses many issues and aspects that are usually forgotten and helps to provide resilience to disturbances before they happen.

The drone software development process also supported the development team in monitoring and controlling work and work items. It helped to improve developers’ activities by providing the means of monitoring and understanding relationships and consequences of activities or tasks. Also, certification of software developed like this is easier and takes less time for testing and verification.

The performed research and its practical implementation proved that, for certification-ready software development, it is worth establishing a process and adhering to it.

Author Contributions: Conceptualization, S.R. and C.S.; methodology, S.R.; formal analysis, S.R. and S.C.; investigation, S.R.; data curation, S.R.; writing—original draft preparation, S.R.; writing—review and editing, C.S.; visualization, S.R.; supervision, C.S.; project administration, S.R.; funding acquisition, C.S. All authors have read and agreed to the published version of the manuscript.

Funding: Lukasiewicz Research Network—Institute of Aviation, Al. Krakowska 110/114, 02-256 Warsaw, Poland—Internal funding [22165].

Institutional Review Board Statement: Not Applicable.

Informed Consent Statement: Not Applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy and internal funding restrictions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. AQAP 2210. *Wymagania Uzupełniające NATO do AQAP-2110 i AQAP-2310 Dotyczące Zapewnienia Jakości Oprogramowania*; Version 2, A ed.; ©Centrum Certyfikacji Jakości: Warsaw, Poland, 2015.
2. Fowler, M. Is High Quality Software Worth the Cost? Available online: <https://martinfowler.com/articles/is-quality-worth-cost.html> (accessed on 15 October 2024).
3. Martin, R.C. *Czysta Architektura. Struktura i Design Oprogramowania. Przewodnik dla profesjonalistów*; Helion SA: Gliwice, Poland, 2018; ISBN 978-83-283-4225-5.

4. Hooks, I. *Why Johnny Can't Write Requirements*; Compliance Automation Inc.: Aurora, CO, USA, 2010.
5. Hooks, I.; Wheatcraft, L. Scope—Magic. Alamo Chapter of the Project Management Institute (PMI): San Antonio, TX, USA, 2001.
6. RTCA-DO-178C; Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc.: Washington, DC, USA, 2011.
7. Richards, M.; Ford, N. *Podstawy Architektury Oprogramowania Dla Inżynierów*; Helion SA: Gliwice, Poland, 2021; ISBN 978-83-283-7027-2.
8. Martin, R.C. *Czysty Agile. Powrót do Podstaw*; Helion SA: Gliwice, Poland, 2020; ISBN 978-83-283-6304-5.
9. Project Management Institute. *The Standard for Project Management and a Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 7th ed.; Project Management Institute Inc.: Newton Square, PA, USA, 2021; ISBN 978-1-62825-664-2.
10. MIL-STD-882E; System Safety. Headquarters Air Force Materiel Command/SES (System Safety Office): Wright-Patterson Air Force Base, OH, USA, 2012.
11. O'Connor, P.; Kleyner, A. *Practical Reliability Engineering*, 5th ed.; John Wiley & Sons: Hoboken, NJ, USA, 2012; ISBN 978-0-470-97981-5.
12. ISO 9000; Introduction and Support Package: Guidance on the Concept and Use of the Process Approach for Management Systems. ISO: Geneva, Switzerland, 2008.
13. ISO/IEC 12207-2008; Systems and Software Engineering—Software Lifecycle Processes, 2nd ed. ISO/IEC-IEEE: Piscataway, NJ, USA, 2008; ISBN 0-7381-5664-7.
14. Axelos Global Best Practice. *PRINCE2®—Skuteczne Zarządzanie Projektami*; The Stationery Office: Norwich, UK, 2019; ISBN 9780113315543.
15. INCOSE. *Systems Engineering Handbook: A Guide for Systems Engineering Life Cycle Processes and Activities*; International Council on Systems Engineering (INCOSE): San Diego, CA, USA, 2015; ISBN 9781118999400.
16. ECSS-Q-ST-80C; Space Product Assurance: Software Product Assurance, Revision 1. ESA-ESTEC, Requirements & Standards Division: Noordwijk, The Netherlands, 2017.
17. AGILE Business Consortium. *AgilePM Agile Project Management Handbook V2*; Polish ed.; Buckland Media Group Limited: Dover, UK, 2019; ISBN 9781910961049.
18. Sommerville, I. *Inżynieria Oprogramowania*, X. ed.; Wydawnictwo Naukowe PWN SA: Warsaw, Poland, 2020; ISBN 978-83-01-21259-9.
19. MIL-STD-498; Software Development and Documentation. US Department of Defense: Fort Belvoir, VA, USA, 1994.
20. ECSS-E-HB-40A; Space Engineering: Software Engineering Handbook. ESA-ESTEC, Requirements & Standards Division: Noordwijk, The Netherlands, 2013.
21. ECSS-E-ST-40C; Space Engineering: Software. ESA-ESTEC, Requirements & Standards Division: Noordwijk, The Netherlands, 2009.
22. Douglass, B.P. *Agile Model-Based Systems Engineering Cookbook*, 2nd ed.; Packt Publishing Ltd.: Birmingham, UK, 2022; ISBN 978-1-80323-582-0.
23. Beck, K.; Grenning, J.; Martin, C.R.; Beedle, M.; Highsmith, J.; Mellor, S.; van Bennekum, A.; Hunt, A.; Schwaber, K.; Cockburn, A.; et al. Agile Manifesto. Manifesto for Agile Software Development. Available online: <https://agilemanifesto.org/> (accessed on 15 October 2024).
24. NASA Goddard Space Flight Center. *Recommended Approach to Software Development, Revision 3*; NASA Goddard Space Flight Center: Greenbelt, MD, USA, 1992.
25. Software Development Plan. Available online: <https://acqnotes.com/acqnote/careerfields/software-development-plan> (accessed on 15 October 2024).
26. United States Government US Army. *Defense Acquisition Guidebook*; CreateSpace Independent Publishing Platform: Scotts Valley, CA, USA, 2013.
27. INCOSE. *Systems Engineering Leading Indicators Guide*; Version 2.0; Roedler, G., Rhodes, G.H., Schimmoller, H., Jones, C., Eds.; Massachusetts Institute of Technology, INCOSE, and PSM: Cambridge, MA, USA, 2010.
28. STANAG 4671; Unmanned Aerial Vehicles Systems Airworthiness Requirements (USAR), 1st ed. NATO Standardization Agency: Brussels, Belgium, 2009.
29. Glinz, M.; van Loenhoud, H.; Staal, S.; Bühne, S. *IREB Handbook for the CPRE Foundation Level According to the IREB Standard*; Version 1.0.0; International Requirements Engineering Board (IREB) e.V.: Karlsruhe, Germany, 2020.
30. Schwaber, K.; Sutherland, J. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*; ©Ken Schwaber and Jeff Sutherland: Francisco, CA, USA, 2020.
31. European Space Agency (ESA). *Technology Readiness Levels Handbook for Space Applications*; Issue 1, Revision 6; TEC-SHS: Noordwijk, The Netherlands, 2008.
32. Borodacz, K.; Szczepanski, C. GNSS-denied navigation system for the manoeuvring flying objects. *Aircr. Eng. Aerosp. Technol.* **2023**, *96*, 63–72. [CrossRef]
33. Pogorzelski, T.; Zielińska, T. Vision Based Navigation Securing the UAV Mission Reliability. In *Automation 2022: New Solutions and Technologies for Automation, Robotics and Measurement Techniques*; Szewczyk, R., Zieliński, C., Kaliczyńska, M., Eds.; Springer: Cham, Switzerland, 2022; Volume 1427. [CrossRef]

34. ISO/IEC 25010:2011; Systems and Software Quality Requirements and Evaluation (SQuaRE)—Software Quality Models. ISO: Geneva, Switzerland, 2011. Available online: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010> (accessed on 10 February 2023).
35. Explanation of “Issue—Gitlab”. Available online: <https://docs.gitlab.com/ee/user/project/issues/> (accessed on 15 October 2024).
36. Project Plans. Available online: <https://www.projectmanager.com/guides/project-planning> (accessed on 15 October 2024).
37. Rierison, L. *Developing Safety-Critical Software A Practical Guide for Aviation Software and DO-178C Compliance*; Version Date: 20121016; CRC Press Taylor & Francis Group: Boca Raton, FL, USA, 2013; ISBN 978-1-4398-1368-3.
38. Cohn, M. Estimating with Use Case Points. Available online: <https://www.mountaingoatsoftware.com/articles/estimating-with-use-case-points> (accessed on 15 October 2024).
39. Wheatcraft, L.S. *Thinking Ahead to System Verification and System Validation*; Requirements Experts: Austin, TX, USA, 2016.
40. Wrycza, S.; Marcinkowski, B.; Wyrzykowski, K. *Język UML 2.0 w Modelowaniu Systemów Informatycznych*; Helion SA: Gliwice, Poland, 2005; ISBN 83-7361-892-9.
41. Delligatti, L. *SysML Distilled A Brief Guide to the Systems Modelling Language*; Pearson Education Inc.: Upper Saddle River, NJ, USA, 2014; ISBN-13: 978-0-321-92786-6.
42. Cockburn, A. *Writing Effective Use Cases*; Addison-Wesley: Boston, MA, USA, 2001; ISBN 0-201-70225-8.
43. Black, R.; Claesson, A.; Coleman, G.; Cornanguer, B.; Forgacs, I.; Linetzki, A.; Linz, T.; van der Aalst, L.; Walsh, M.; Weber, S. *Certified Tester. Foundation Level Extension Syllabus. Agile Tester*; Version 2014; International Software Testing Qualifications Board (ISTQB): Brussels, BL, USA, 2014.
44. Cerquozzi, R.; Decoutere, W.; Dussa-Zieger, K.; Riverin, J.-F.; Hryszko, A.; Klonk, M.; Pilaeten, M.; Posthuma, M.; Reid, S.; Riou du Cosquer, E. *Certified Tester. Foundation Level Syllabus*; v4.0; International Software Testing Qualifications Board (ISTQB): Brussels, BL, USA, 2023.
45. International Institute of Business Analysis™ (IIBA®). *BABOK® A Guide to the Business Analysis Body of Knowledge*; v3; International Institute of business Analysis: Toronto, ON, Canada, 2015; ISBN 13978-1-927584-02-6.
46. *MIL-HDBK-61A*; Configuration Management Guidance. USA DOD: Washington, DC, USA, 2001.
47. Git Branching—Branches in a Nutshell. Available online: <https://git-scm.com/book/en/v2/Git-Branching-Branched-in-a-Nutshell> (accessed on 15 October 2024).
48. Beck, K. *TDD. Sztuka Tworzenia Dobrego Kodu*; Helion SA: Gliwice, Poland, 2020; ISBN 978-83-283-6572-8.
49. Shoan, W.; Jun, L. *C++ Coding Standards and Style Guide*; NASA Goddard Space Center-Mission Application Branch—Code 583: Greenbelt, MD, USA, 2005.
50. Wheatcraft, L.S. *Triple Your Chances of Project Success: Risk and Requirements*; Requirements Experts: Huston, TX, USA, 2011.
51. US Department of Defense—Systems Management College. *Systems Engineering Fundamentals*; Defense Acquisition University Press: Fort Belvoir, VA, USA, 2001.
52. Hooks, I. *Writing Good Requirements*; Requirements Experts: Huston, TX, USA, 1993.
53. Chrabski, B.; Zmitrowicz, K. *Inżynieria Wymagań w Praktyce*, 1st ed.; Wydawnictwo Naukowe PWN SA: Warsaw, Poland, 2015; ISBN 978-83-01-18018-8.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.