Bachelor's thesis

Degree Programme in Information and Communications Technology

2021

Timo Haavisto

# SCRUM FRAMEWORK EXECUTION IN THE COURSE PROJECT GAME DEVELOPMENT

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information and Communications Technology

2021 | 65 pages

Supervisor: Principal Lecturer Mika Luimula, Adj. Prof.

Timo Haavisto

# SCRUM FRAMEWORK EXECUTION IN THE COURSE PROJECT GAME DEVELOPMENT

This thesis describes the Scrum framework and agile approach, then observes two game development course projects from initiation to delivery from Scrum framework execution perspective.

During the first project, the focus was on identifying challenges arising from Scrum framework execution in time-constrained course projects.

The Scrum framework is built on continuous communication and a fixed process cycle through sprints. The sprints consist of predefined Scrum rituals, which repeat in the same fashion for each sprint. The framework assumes a daily work schedule, which cannot be arranged in a part-time course project setting, due to participants having other time commitments throughout the course.

The development sprints are generally 1-3 weeks in length. However, such guideline assumes daily work and meetings. Therefore one week of a course project is not equal to a one week sprint in terms of progress. Longer sprint, on the other hand, hinders inspections regarding accomplished results. In addition, the stakeholders and Product owner are generally only available during the course allocated time slots.

Based on the first project's findings, a hypothesis of framework enhancement was created, containing adjustments to Scrum framework execution under these constrained conditions. The created framework was tested in the second project with the same constraints, and the project was then observed in the same fashion. Enhancements and effects of alterations were monitored and compared to the first project's results.

The thesis commissioner was Futuristic Interactive Technologies Research Group of Turku University of Applied Sciences. The practical result of this thesis was a recommendation and framework for organization of course project teams and schedule, presentations for each major Scrum ritual, and a toolkit of found best practices for students. By following the recommendations, observed pitfalls of game development course project could be avoided or mitigated.

KEYWORDS:

software development, game development, Unity, project management, agile approach, SCRUM, Extreme Programming, case study

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2021 | 65 sivua

Ohjaaja: yliopettaja Mika Luimula, dos.

Timo Haavisto

# SCRUM-VIITEKEHYKSEN TOTEUTUS KURSSIMUOTOISISSA PELIKEHITYSPROJEKTEISSA

Tässä opinnäytetyössä tarkastellaan kahden kurssimuotoisen pelikehitysprojektin kulkua Scrum-viitekehyksen toteutuksen näkökulmasta, innovaatiovaiheesta tuote-esittelyyn.

Ensimmäisen projektin aikana keskityttiin kurssimuotoisen pelikehitysprojektin Scrum-toteutukseen liittyvien keskeisten haasteiden tunnistamiseen.

Scrum-viitekehys rakentuu jatkuvan yhteydenpidon ja kehityskierrosten ympärille. Kehitys-kierrokset koostuvat ajallisesti sidotuista tapahtumista, rituaaleista, jotka toistuvat kehityskierroksesta toiseen. Scrum-viitekehyksessä rituaalien määrittely rakentuu kokopäiväisen projektin työstön ympärille. Se ei esitä mukautuksia sellaisten aikarajoitteiden varalle, joita lukujärjestykseen sidotuissa kurssimuotoisissa projekteissa esiintyy.

Koska kurssimuotoinen projekti etenee pääasiassa lukujärjestykseen sidottuina päivinä, viikon mittainen pyrähdys sisältää viikon työn asemesta vain muutaman päivän työn. Pidempi pyrähdyksen kesto vuorostaan vaikeuttaa projektin etenemisen tarkastelua. Myös projektin toimeksiantaja ja tuoteomistaja ovat tavoitettavissa pääsääntöisesti vain kurssin aikataulun mukaan: ohjaus ja palaute viivästyvät ja kokoukset venyvät.

Tarkastelun ja haasteiden tunnistuksen jälkeen rakennettiin viitekehyksen mukautus tehtyjen havaintojen perusteella. Mukautettua viitekehystä testattiin toisessa projektissa: saavutettuja tuloksia sekä hyötyjä vertailtiin ensimmäisen projektin tuloksiin ja havaintoihin.

Opinnäytetyön toimeksiantaja oli Tulevaisuuden interaktiiviset teknologiat -tutkimusryhmä Turun Ammattikorkeakoulusta. Opinnäytetyön lopputuloksena esitettiin suositus viitekehyksen mukautuksesta aikataulun, Scrum-tiimin ja projektin organisoinnin osalta. Rakennettua viitekehystä seuraamalla havaitut kurssimuotoisen pelikehitysprojektin toteutuksen haasteet pystytään kiertämään. Opiskelijoille rakennettiin tietopaketti tarkastelussa löydetyistä hyvistä käytännöistä. Viitekehyksen toteutuksesta rakennettiin esityspaketti kurssitoteutuksen tueksi.

ASIASANAT:

ohjelmistokehitys, pelikehitys, Unity, projektinhallinta, ketterät menetelmät, SCRUM, Extreme Programming

# TABLE OF CONTENTS

Click here to enter text.

# 1 INTRODUCTION

Game development is a rapidly growing sector in the software development industry. As the complexity of the games also increases, with technical capabilities and funding, the amount of personnel involved worldwide in the development of game products increases accordingly.

With a larger number of people working on more and more complex projects, project management complexity also increases. In addition, game design and development is driven by innovation: the full scope of a game is often not known beforehand – rather it appears throughout the design and development phases, and may change at any point during the project. In traditional software development, the product's scope remains the same throughout the project.

With these peculiarities, agile software methodologies would be a natural choice for game development: agile frameworks offer requirement flexibility as well as testing, inspecting and feedback cycle. Scrum is presently the most used agile framework in the software development industry, followed by Extreme Programming.

The goal of this thesis was to find possible improvements and framework guidelines for projects executed during game development project courses of Turku University of Applied Sciences. In these projects, the game development teams are heavily constrained time-wise by the course schedule.

This thesis observes the execution of Scrum in two game development course projects. The observed development teams would only be able to work on the project for roughly 16 hours per week per person. This constraint directly affects the Scrum framework, which has requirements for a specific set of meetings, inspections and timeslots, all allocated and executed in a static manner. The experimental manner of these projects meant undefined scope of the end product: even the technology stack could change during the project.

The course participants had two major paths: programming and artwork, which were mutually exclusive choices for specialization. Also, the game industry uses freelancers for both programming and art. The heavy time constraints observed by the student teams would also appear in freelance work and in game studios where personnel cannot be dedicated to work on a specific project but have instead several projects in parallel.

The first observed project was a VR game project with over 20 members in the development team, further split into the product's different features. The second project was a mobile game project with 7 members in the development team, with Scrum execution adjustments based on first project's findings. Again, the team members specialized in programming or artwork.

The thesis is structured as follows:

Chapter 1 introduces the context and the objectives of this thesis.

Chapter 2 summarizes software engineering characteristics and goes through the evolution of software development from traditional methods to agile principles.

Chapter 3 describes the most common agile software development frameworks, namely Scrum and Extreme Programming.

Chapter 4 describes the characteristics, used tools and techniques of the game development project course and then compares the strengths and weaknesses of the two agile frameworks.

Chapters 5 and 6 present the characteristics and observations from Project 1 and the initial framework for improvement. The execution of the framework is tested in Project 2. The framework is then refined further by reflecting on the practical findings with the expectations already described in Chapter 4.

Chapter 7 presents the final results for the agile framework execution with the modified framework in a time-constrained Unity game development project. It also discusses proposals for future iterations.

# 2 FUNDAMENTALS OF AGILE IN SOFTWARE ENGINEERING

This chapter introduces agile core values, discussed the emergence of agile frameworks, and then describes the characteristics of the game development course project, reflecting on agile values.

## 2.1 Traditional software development

Software Engineering is a process of developing a product which consists of software. Various definitions of Software Engineering as a term exist, and the scope of those terms varies from development-focused concrete engineering to ones that encompass the whole product life cycle, also considering the economic efficiency of the product. (Sommerville 2004, 7.)

From the 1970s to early 1990, traditional software development frameworks were widely used, and the best practise for software development was through careful planning and design, formalized quality assurance, use of analysis and documenting every possible detail during all phases. The processes were rigid and controlled. The largest portion of the project resources was dedicated to testing. (Royce 1987, 5-7; Sommerville 2004, 396.)

These frameworks were favored to meet the demand of the market at the time: computer systems were often centralized and ran critical operations, they were interfaced with a large number of other software, and once the systems' development project was completed, the systems were taken into use and then operated until the end of their product life, untouched. (Sommerville 2004, 396.)

The phases of traditional software development are generally described as a waterfall, without feedback loops. The phases, simplified from the paper by Royce (1987, 2), are shown in Figure 1.

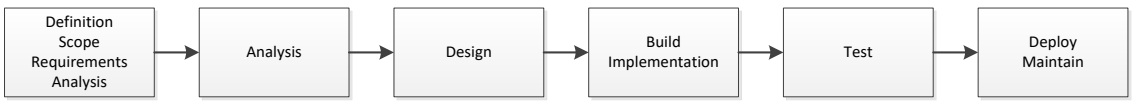


Figure 1. Traditional steps of software development.

The waterfall model specifies a team for each of these phases, which is specialized to work on that specific phase (Sommerville 2004, 67). The amount of documentation is rigorous and necessary, because personnel and individual knowledge do not carry over the phases: without documentation, any mistake would likely be reviewed by the same person who made those mistakes originally (Royce 1987, 5).

The development of a product using a traditional framework, is generally described as a linear process, following the traditional non-software engineering cycle. To build a bridge, each phase must be executed and completed in order which cannot change. Similarly, in the waterfall model, software projects have specific phases: each step is evaluated against the predetermined set of requirements sequentially, one after another. If the evaluation has a positive outcome, the phase is signed off. Once a phase is completed, it is not be revisited as the requirements have been signed as fulfilled. (Sommerville 2004, 67.)

If requirement acquisition was poorly executed at any of the phases, all subsequent phases would also fail: the features and system characteristics would be implemented and tested against those requirements. If any of those requirements changed, all phases would need to be reworked.

While the waterfall model also had mechanics for revisiting previously completed phases, in practice the development companies treated the development process as completely sequential, similarly to the traditional construction engineering projects. (Pressman 2014, 42.)

When the project scope was rigid, bound on predetermined specifications, the development time and development resources were planned for the exact scope and could not suffer any changes. With any change of the project scope, the development time and/or resources were therefore generally exceeded. (Matković and Tumbas 2010, 3.)

According to the Chaos Report (Johnson et al. 1994, 2), software development projects had as high as 83.8% rate of impairment or hindrance in the 90s: more specifically, 52.7% went over budget or overtime, and had fewer features than designed. 31.1% of the projects were canceled at varying stages. This situation was called software development lag, software development crisis or even software development chaos (Johnson et al. 1994, 1). Software engineering was considered the same as any other engineering, followed the same rigid waterfall development phases, which did not allow

accommodation of changes: the same framework was used for bridge building and software engineering, except in case of project failures (Johnson et al. 1994, 1).

Even though the Standish Report's figures have been disputed later by several other authors (Glass 2005; Eveleens & Verhoef 2010), the report mentioned several wider values, which later appeared in the Agile Manifesto (Beck et al. 2001a).

In addition, the report (Johnson et al. 1994, 3) recognized that smaller companies were more successful with their projects than large companies. User involvement was a key factor in both successful and challenged projects (Johnson et al. 1994, 4). Incomplete requirements were found to be a key factor in the failure of a project (Johnson et al. 1994, 5). The report concluded with further notes, indicating that "delivery of software early and often will increase the success rate", describing iterative software development as a solution: "Growing software [instead of developing software] engages the user earlier, each component has an owner or small set of owners, and expectations as realistically set" (Johnson et al. 1994, 8). To conclude, all of these findings were incorporated into the agile principle later as part of the Agile Manifesto (Beck et al. 2001a).

## 2.2 Agile principles

Reviewing the waterfall model, many of the processes and roles do not seem to contribute to the actual implementation or decision-making. Defining the concept, finding the scope, and designing the system was a larger step in the project, when compared to the actual realization steps. Such resource distribution incurs high resource costs to gain even a small development effort. Also, testing with end users only occurred when the whole system was completed, therefore, any failure in previous steps was catastrophic: the product would not fulfill the user's needs. Finally, any blocker during the phases would delay every subsequent phase proportionally due to the sequential process. Therefore the deployment of the product was also delayed.

The agile approach emerged as a response to dissatisfaction with traditional software development frameworks. In the mid-90s, large number of software developers proposed a new, more agile approach – the Agile Manifesto.

The manifesto contains 4 core values, which define value: it values communication and human interaction higher than processes and tools; a working solution over

documentation; interaction and collaboration with the customer over contract negotiation and finally, responding to a change over following a plan. (Beck et al. 2001a)

For accomplishing the core values, the Agile Manifesto presents a collection of 12 agile software development principles, meant to direct values towards agile approach: for customer collaboration, it elevates the end user's satisfaction to the highest measurement of value, and welcomes changes of requirements and scope during the development. From the development point of view, the manifesto encourages simple, working, and sustainable solutions, which can be rapidly released to production. From the team management and personnel perspective, the Agile Manifesto encourages face-to-face communication, team self-reflection, and individual knowledge of one's own strengths and weaknesses. All this should be delivered in a cost-efficient way. (Beck et al. 2001b; Hohl et al. 2018.)

The Agile Manifesto was an overhaul of how software development was understood until then. Its contents opposed traditional software development framework, while it also lacked the exact steps and processes on how to achieve the values it described. Many, therefore, understood the definition of "agile" differently, and there are still many different definitions for agile, as well as differences in emphasis, as described in the conference paper by Laanti et al. (2013, 5-8). They suggest that instead of defining agile itself, it would be clearer to focus more on the advantages of specific agile practices and accept that agile is still developing (Laanti et al. 2013, 12-13).

Agile, as defined in the Agile Manifesto, is a mindset and philosophy (Highsmith, 2001), which once grasped, could be executed in a framework of software development.

However, the core values of the manifesto may not be in line with the goals of a particular project. If the project goal and needs of the customer required certain processes, tools, verbose documentation, and following a plan precisely – all of which are features that agile values less – such requirements would contradict many of the agile core principles. As agile values customer satisfaction over other aspects (Beck et al. 2001b), in such a project, agile frameworks could be severely constrained. It could be agile to use a non-agile traditional development method instead.

Because agile by itself is not a framework or a process, the choice of the agile framework needs to be made according to the project's characteristics.

The manifesto gives a set of relations between values: what it values more and what it values less. The implementation of different agile frameworks grant different foundational strengths and advantages.

Therefore a generally defined agile framework does not necessarily fit all development projects, and even agile principles may get discarded. With an incorrect selection of elements and the wrong framework, the framework's strengths could turn into weaknesses. The original authors of the Agile Manifesto have later evaluated many current frameworks in an interview (Hohl et al. 2018, 19-21), takeaways pointing out the current trend of fashionable agile, reminding that agile is not a solution to everything. Current frameworks are rather a good starting point for future than the end of the road.

# 3 AGILE FRAMEWORKS

This chapter describes the characteristics of two agile frameworks, Extreme Programming and Scrum, in preparation for discussion regarding their advantages and disadvantages in a game development course project in Chapter 4.

## 3.1 Extreme programming

### 3.1.1 Introduction

Extreme programming (XP) was invented and presented by Kent Beck in the 2000s, in his book “Extreme Programming explained”. It defines practices to fulfill the agile principle, focusing on customer involvement and satisfaction, minimalism, fast deployments, testing, and pair programming (Sommerville 2004, 400). In the foreword of his book, Beck states that XP as a whole can be seen as a plan for a change in software development (Beck and Andres 2004). A year later Beck was driving that change as a member of the group who jointly published the Agile Manifesto (Beck et al. 2001a).

In XP, scope change support is high. It can happen at any point of the development cycle. By contrast to the traditional software engineering, XP does not design or plan for change. Instead, it relies on **continuous integration**, where all system features are planned and implemented in the simplest way possible, without any consideration to requirements or user stories that have not yet been defined (Sommerville 2004, 401). XP method claims that planned changes often do not come to reality, which means the development effort considering those changes is often wasted – instead, unexpected changes tend to occur (Sommerville 2004, 401). This **incremental planning** also matches the experimental nature of game development projects and saves time, and gives the widest flexibility for scope changes.

In XP, all developers work on all components that are developed during a sprint (Martin 2002, 14). This is done to **share the knowledge and ownership** regarding all of the system’s features. Developers of different components work together in **pair programming**. Therefore both gain knowledge regarding the components developed by another. The pairs do not stay the same for long. Instead, they change to spread the knowledge within the development team. (Sommerville 2004, 401.)

This practice also avoids single points of failure, where a single developer becomes an expert of a specific component, while others do not know how it works. When all developers have worked on all components of the system at least briefly, the possible changes in personnel resource reallocation and prioritization of personnel become easier. This could be advantageous also in project course, where team members can be shuffled in the middle of the project, and participation can vary.

The development team of XP is expected to work in an **open workspace**, with the whole team present and available for communication at any time. Knowledge sharing is therefore always possible, even outside of the pair programming team, and the team can reallocate its developer resources on fly when necessary (Martin 2002, 15). For quick feedback on requirements and features, there is a dedicated **customer's representative in the development team**, working in the same workspace together with the developers (Sommerville 2004, 399-400). The customer's representative decides how important a feature is, while the development team gives feedback on how much resources it needs. Developers give a customer an estimated budget of available development resources, while the customer decides how those resources should be spent (Martin 2002, 15).

Testing has a high focus in XP. It expects developers to write extensive unit tests for all their development, and only after completing tests can the increment be accepted and integrated. XP calls for wide sets of tests for everything, and it may be challenging to figure out tests of all possible cases. In addition, these tests should include the customer's representative as part of the testing process to get immediate feedback on new development. (Sommerville 2004, 404.)

Comprehensive **test-first development** emphasizes testing, which in turn is generally not a priority in experimental proof of concept projects. In addition, creating wide sets of tests could consume too much time of time-constrained project course team.

With no planning for change in advance, together with extensive testing, XP would seem to cause more and more complexity and smelly code, driving the development away from any proper architecture. XP attempts to resolve this via **continuous refactoring** of already existing code. All developers in XP are expected to maintain and refactor their code continuously, not only when working on newly risen changes and requirements (Beck and Andres 2004; Sommerville 2004, 404).

While refactoring, the developers also write the accompanying unit tests. This means that in XP, source code would constantly be in change, as the development team would visit already implemented components to refactor them for those requirements. This emphasis on code refactoring could become an issue if developers' knowledge level varies like they do in course projects. Also, source control must be well organized and known by all development team members.

When adapted to the traditional software development phases, the development cycle follows a circle of continuous integration and continuous development (Sommerville 2004, 399), as illustrated in Figure 2.

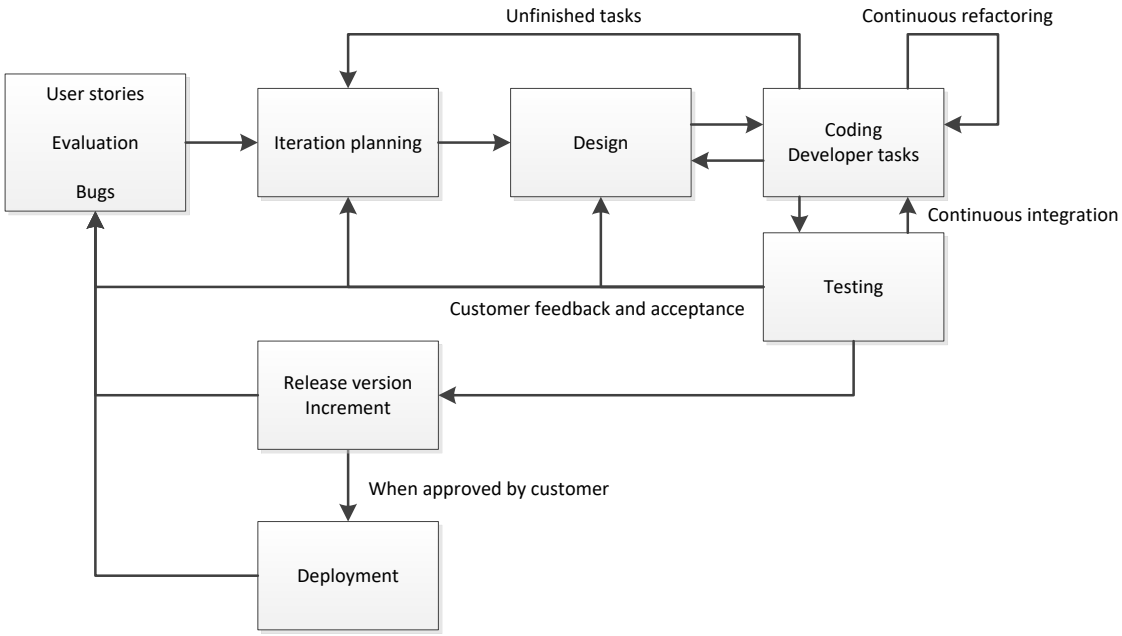


Figure 2. Extreme Programming phases and flows.

The release cycle is very quick, with new **small releases** created several times per day, and deployments to customers are provided roughly every two weeks (Sommerville 2004, 401). With project course constraints in schedule, limiting development work hours and days, this could be both advantage and a disadvantage. On the one hand, even small development pieces have measurable progress as a new release contains new features. But on the other, too large development chunks would be left unfinished until the next project time slot, hindering progress on any depending development.

To manage source control with these rapid deployments, XP proposes a simple rule: everyone needs to merge their work with whoever was first to check in (Martin 2002, 14).

The source control is, therefore non-blocking, nobody can reserve a module to work on it – everyone must always be prepared to merge their work with whatever concurrently done work. This is called continuous integration, and it is the first part of a commonly known agile process: continuous integration and continuous development (CICD in short). It is supported by many major software development tools and development support systems.

The workload of the development team depends on their expertise and knowledge. As in many agile frameworks, the goal is to start slow and ramp-up speed as developers get more accustomed to the technologies and customer's needs. While development is split into sprints, the software development project as a whole is a long process, where **sustainable development speed** (also called momentum or velocity in other agile frameworks) is valuable. It grants the ability to estimate development speed and set feasible deployment goals. XP goes even further by stating (according to Martin 2002, 15) that working overtime is outright forbidden. There is one exception, though, and that is the last week before release, assuming the team can be reasonably expected to reach its goal.

### 3.1.2 Conclusion

Extreme Programming is very flexible for scope changes. It heavily focuses on the interaction between developers by emphasizing pair programming and customer connection through the representative who works as a member of the development team.

The customer's representative could be considered a vision keeper of the final product. However, in XP, the role appears to be more practical: definitions and user stories for features, previously planned or not, unlike in other frameworks with more restrictions regarding change of the project scope.

With knowledge sharing, developers are aware of all components of the developed system. Thus XP allows quick reallocation of personnel on the spot, minimizing obstacles and avoids development getting stuck.

## 3.2 Scrum

### 3.2.1 Introduction

Scrum was originally presented in 1995 during the annual ACM research conference, OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) in Austin, Texas, jointly by Jeff Sutherland and Ken Schwaber. Both were 5 years later part of the group that presented the agile manifesto. The published paper's abstract claimed that planning, estimation, and successful completion of software development does not occur in practice (Schwaber 1997, 1). This claim is similar to the Chaos report by Johnson et al. (1994, 1), based on their findings from practical participation in large projects at the time (Hohl et al. 2018, 13).

To address the crisis in software development, Schwaber proposes Scrum, an enhancement of the iterative-incremental development cycle, where the software development process is assumed to be an unpredictable and complicated process. A framework of activities that combine tools and techniques to build systems in a "controlled black box". (Schwaber 1997, 2).

Black box means acknowledgment that as professionals and specialists of their field, developers know best how to resolve specified requirements most efficiently, and details of implementation are not important. Hence, management and outsiders do not interfere with how the system is being implemented. They cannot affect its development details except by setting specific requirements for the product goal. Unless explicitly specified, all tools, programming languages, and platforms are therefore freely chosen by the Scrum team to create as much value to the product as possible. Progress of that value creation is inspected at regular intervals.

Scrum defines three roles, who together work as a Scrum team to create a product: **Scrum Master**, who observes the execution of the Scrum process, isolates the development team from non-development related concerns and finds possible impediments; **Product owner**, who creates and refines the vision and goals of the product, through maintaining and taking sole undivided responsibility of Product backlog; and finally the Development team which contains all personnel working on the project by creating increments (Schwaber and Sutherland 2020, 5-6).

The **Development team size is 3-10 people**, with less than 7 recommended (Schwaber 1997, 16; Schwaber and Sutherland 2020, 5). If more persons work in the development team, the team should be split and reorganized into **multiple teams, each working on the same Product backlog** and with the same Product owner (Schwaber and Sutherland 2020, 5). The teams can be specialized, assigned to certain product functionality, or assigned to a specific system component (Schwaber 1997, 16).

Even though the responsibility of Product owner cannot be lifted, the tasks of the Product owner may be delegated (Schwaber and Sutherland 2020, 6). Delegation of a system component to a **Product owner proxy** is one alternative, however that is not advised: with proxies, decision-making and responsibility could become vague, slowing the feedback cycle (Rubin 2012, 183).

It is, however, better than having no access to the Product owner at all. The risk of vision conflicts between the proxies and actual Product owner is higher, which means development team tasks could be organized in a conflicting manner. Product owner proxy also emphasizes reliance on backlog items for requirements without enough discussion. (Watts 2013, 67).

In the context of a game development project course, this split by component or specialization seems natural, especially when the development team contains course project members from both programming and artwork focus paths. The split of programming and artwork also drives flexible design, interfacing by decoupling artwork from program code. Artwork could be originally set as placeholders with the same scale, dimension, and characteristics, which then could be substituted as artwork completes, without affecting code development. Hardcoded direct artwork manipulation is discouraged.

Planning and architecture design is executed in Scrum during a preplanning phase, which is done before any development or sprints. Unlike Extreme Programming, Scrum follows traditional steps of software engineering during the **planning phase**: the initial requirement acquisition, stakeholder discussions, overall schedule, budgeting, analysis, planning, and design are set here. Once they are finalized, they are not changed during the future Scrum steps. Based on the outcome of these defined processes, an initial Product backlog is created. After that, the planning continues to design the architecture which would support the implementation of the backlog items. The outcome of this phase is an overarching system design on the high level, which then remains throughout the project. (Schwaber 1997, 9-13).

Therefore, Scrum is not as flexible as Extreme Programming, when experimental proof of concept projects are concerned. If the initial knowledge of the product's features are vague, broad design may be faulty and the underlaying architecture created for it could be incorrect for implementation. Especially for emerging technologies, unexpected architecture incompatibility may appear.

After the planning phase is completed, the development phase starts.

Scrum seeks to **perform the most valuable work first**, including the planning phase (Rubin 2012, 18). Even with only the planning phase completed, the project can be continued later with actual development cycles. The development would start from the most important items in the Product backlog, again ensuring that items creating the most value are done first.

All development is done during **iterative-incremental development cycles** called sprints, each starting with **sprint planning meeting** to set a goal for the sprint, ending with **sprint review**, where work is inspected, and finally **sprint retrospective** where sprint events are recapped (Schwaber 1997, 14).

During the sprint, a short 15-minute stand-up called **daily meeting** is done every day. In the daily meeting, every development team member answers three questions: what they have done, what they will do next, and have they found any impediments or obstacles hindering their progress. The Scrum master facilitates these meetings and drives the team to resolve any impediments. (Rubin 2012, 264; Schwaber and Sutherland 2020, 9.)

Sprints are usually 1 to 4 weeks in length (Schwaber 1997, 13-14). The sprint length is decided according to the project complexity, level of uncertainty, and development team experience: more risk factors require more inspections and monitoring; therefore, sprint length is shorter if more risks are present. Sprint length is generally kept the same throughout the project (Rubin 2012, 20; Schwaber 1997, 13-14).

For experimental course projects, short sprints would be the obvious choice for frequent inspections and adjustment. However, they would be proportionally too short, considering the work hour restriction per week. In essence, the length of 1-week sprint in the course project environment would only contain 8 to 16 hours of development time instead of a full workweek of 40 hours.

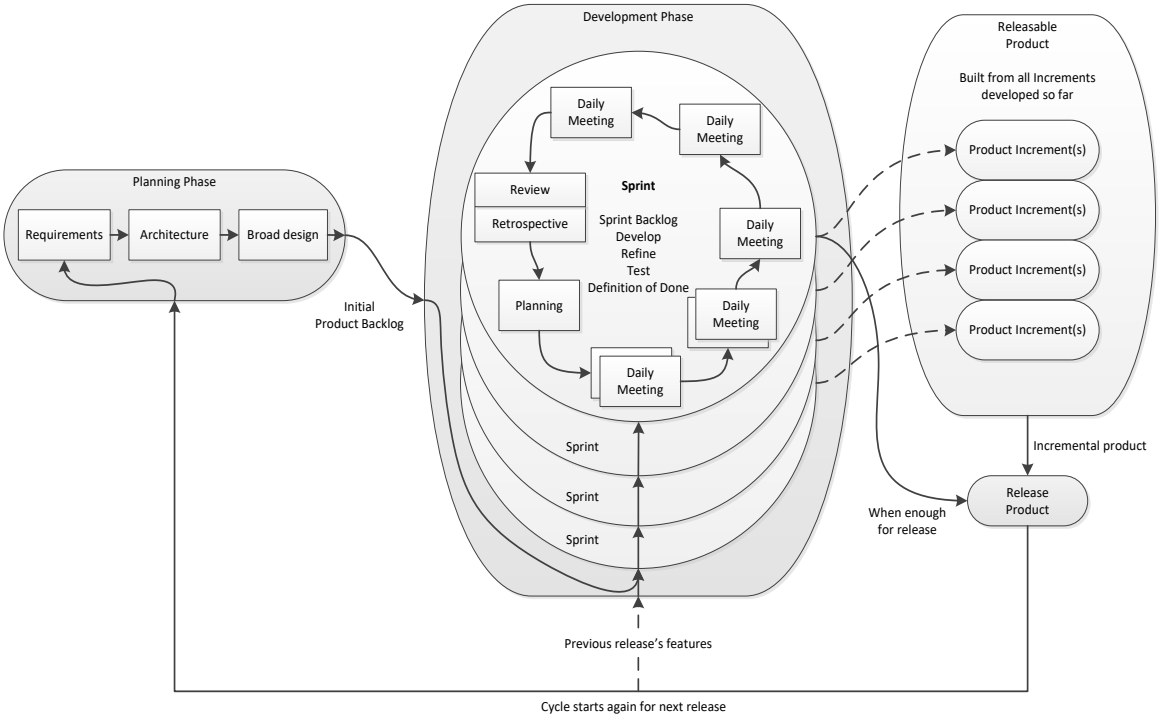


Figure 3. Scrum execution phases and flows.

Sprints continue until stakeholders determine the product to be ready for a release. At this point, the product release is closed, not to be visited again, signaling switch to traditional sequential process flow again (Schwaber 1997, 14).

Unlike in traditional software development the outcome of a sprint cycle is an increment to the product. Therefore it is possible to release a product even if resources (generally time and budget) are consumed prematurely or business events cause project closure. This is advantageous in both volatile game development and project course development: if the project completed even one development sprint successfully, it has already created value for the product by achieving a sprint goal that reflected Product backlog ordering. It is potentially shippable, standalone product fulfilling some of the needs.

Scrum describes three artifacts, which drive the development process within and across sprints. These artifacts are Product backlog, Sprint backlog, and Product increment (Schwaber and Sutherland 2020, 10).

The **Product backlog** contains all currently visioned features and development parts of the product. It is an ordered list, maintained and constantly refined by the Product owner. The Product backlog is a collection of all parts that could be in the final product. The ordering is based on the Product owner's decision on a particular item's value addition to the product, reflecting stakeholders' wishes. The backlog items are in the form of written stories, which define the wanted functionality, requirement or knowledge acquisition. Not all of the items in the backlog necessarily end up in the product, and the Scrum team internally prioritizes items based on their feasibility, complexity, risk, cost, knowledge, and other values. (Schwaber 1997, 13; Rubin 2012, 18-20; Šochová 2016, 127; Schwaber and Sutherland 2020, 10-11). If all Product backlog items are implemented, the product is as perfect as it could be envisioned.

The **Sprint backlog** is a subset of Product backlog items, which are planned to be developed during the particular sprint, fulfilling a sprint goal. Sprint backlogs are generally locked down when the sprint starts, and only the changes that do not affect the achievability of the sprint goal can be done (Schwaber and Sutherland 2020, 7). At the end of the sprint, the sprint's success is determined by the ability to fulfill the sprint goal. A sprint can be canceled altogether by Product owner in case of significant unforeseen changes. In that case, the Sprint backlog items are returned to the Product backlog. (Schwaber and Sutherland 2020, 11.)

The **Product increment** is an improvement that adds value to the product on top of all previous increments, and many increments could be created within one sprint (Schwaber and Sutherland 2020, 11-12). It is an addition to the product that could be released as it has passed all tests and requirements. Product increment occurs only when work fulfills the definition of done and is usable. Definition of done is a set of requirements for the increment, which define a minimum standard for the item to be considered complete. (Schwaber and Sutherland 2020, 12.)

### 3.2.2 Conclusion

Scrum is proven to handle complex projects with good results, and it is currently the most commonly executed agile framework in software development. This achievement is partly due to the diploma certification system, which according to Bob Martin (one of the agile manifesto group members) was a breakthrough idea of Ken Schwaber, creator of Scrum (Hohl et al. 2018, 18). Some of the original authors of the manifesto state that the true meaning of agile has been lost, and the definition has eroded over the years (Hohl et al. 2018, 23).

However, Scrum does not fit every project, and one of the noticeable differences is its inflexibility during the planning phase, which follows the traditional software development method.

# 4 GAME DEVELOPMENT PROJECT COURSE

This chapter describes the characteristics of the observed game development project courses and the tools and techniques used for future evaluation.

## 4.1 Characteristics

To review agile frameworks for a game development project course, the wide range of project types even within that scope should be recalled. In addition to varying types of projects and goals, uncertainty regarding requirements changes is also present.

Regardless of game project type, uncertain scope, and features, the clients and end-users of games often have very high expectations regarding the product. Also, resources change during the project, depending on impressions from prototypes.

Experimental proof of concept type of projects have, by definition, many uncertainties. Only the topic and presented problem are known beforehand. Scope, context, content, and means of development are unknown, depending on results from experiments. Both of the projects observed in this thesis are in this category.

The time constraints in course projects are bound by the weekly work hours as well as available working days. The equipment and facilities required for development may not be available outside of specific time slots. The allocated course schedule constrains the project lifetime. The expertise of the developers varies, as does the knowledge of technologies used. Because the course project has beginning and end, only executed once per developer group, there is no deep knowledge regarding the technologies. Therefore, each course and project is executed with a completely new set of developers with varying experience regarding the project technologies. External resources may become necessary to inspect and outsource components to maintain the schedule.

## 4.2 Tools used in the course

Part of this thesis's goal was to review the tools used during the game development project course, evaluate different tools for future course templates, and find best practices.

### 4.2.1 Project management tool Trello

Trello is a project management tool created by Atlassian. Students were accustomed to using Trello for project and task management during the previous project courses. The tool allows usage of templates for agile software development frameworks and tasking of development activities. (Atlassian 2021a)

The tool is a card-based task system, where each task can be assigned to any participant of the board. The tasks can be commented on and moved between task lists. (Atlassian 2021a)

Trello does not readily support relationships or hierarchies between tasks and cards, nor work effort calculations. Therefore, its effectiveness for following workload is limited: each card can describe a whole feature set or simply describe an individual task. Due to a lack of hierarchies, small tasks cannot be clearly bound to large work units. (Atlassian 2021a)

Trello was used in project 1 by all teams from their own initiative.

### 4.2.2 Project management tool Jira

Jira is a heavy-weight project management tool created by Atlassian, with a much larger feature set than Trello. The two tools can be integrated together, however, Jira supports a much wider set of integrations, including direct integration with the version control platform.Jira also includes native reporting for workload and project work progression and sprints, user stories, and hierarchies and relations between them. (Atlassian 2021b)

This system is widely used in the industry. Its complexity requires a special setup, and the participants of the project course are generally not familiar with it. It does not offer a free license like Trello. (Atlassian 2021b)

Jira was used in project 2 to compare its effectiveness to Trello used in Project 1.

### 4.2.3 Source code and version control platform

During the development, all source code was stored in version control platform, Git. It is a tool developed for following changes in files. As Git follows every change of files included in its designated directory (including subdirectories), a full chain of changes can be followed from the first creation of the file to its current content. Therefore, with the chain, it is possible to revert changes of any file to its any past state. (Git 2021)

With each change committed to a file, information about user who made the change is stored. Thus it is also possible to see who has done which change in the past.

Any type of file can be added to version control. It is useful for software development source code files: program source code is human-readable text, and changes of the files can be reviewed per-row basis (Git 2021). For example, by reverting to previous changes, finding the exact change caused a program error to appear.

For files stored in binary format, version control can be used to see the version chain of files themselves (Git 2021). It is possible to follow the progression of artwork and graphics, from drafts to final versions, as long as this is done without changing the name of the files.

The version control also supports merging different change chains of even the same files together (Git 2021). This is especially convenient for continuous integration of new work when many developers are working on the same component of the developed system. However, editing of the same file causes conflict in version control, where the system is unsure of which changes to keep, which to discard and which to merge. These conflicts are resolved per-row basis. (Git 2021)

### 4.2.4 Development tool Unity

The primary game development tool in both observed projects was Unity, a cross-platform software development program supporting mobile application development, desktop application development as well as virtual reality and augmented reality development with integrated 2D and 3D engines. (Unity 2020a)

Unity adds constraints regarding version control. The scenarios which bind different user interface views together are stored in YAML, a data serialization language that is not easily human-readable (Unity 2020b, Unity 2020c). This means that even though scenarios often bind together the development work of several developers, the version control change chain merging cannot be easily used. Essentially, the editing of scenarios should be restricted to one individual at a time, which is a major constraint regarding the workflow.

Unity also has support for packaging of scenario elements into smaller independent packages, called prefabricated objects or prefabs in short (Unity 2020c, Unity 2020d). Even though they are stored in YAML as well, this component splitting ability mitigates the constraint of scenarios because each prefabrication within a scenario can be independently edited (Unity 2020d). Still, from a version control perspective, these edits should be done by single individual, because merging prefabrication chain of changes is difficult due to its serialization.

These observed constraints emphasize the need for communication within the development team to ensure that individuals only ever edit each scenario and each prefabrication at a time. Otherwise, a conflict in version control appears, which would cause work to be done over again.

## 4.3 Reflecting with Extreme Programming as a framework

This section discusses the advantages and disadvantages of the Extreme Programming framework in the context of a course project and further as a game development project.

### 4.3.1 Course project point of view

**Advantages**

Inspection and sharing information: peer-teaching fills the knowledge gaps. It also mitigates absences, as a key member of the team can be covered by others. Two highly knowledgeable individuals benefit from pair programming: programming errors done by one are spotted by the other’s continuous simultaneous inspection.

Common space: pair programming could also mitigate issues arising from lack of development equipment. With the whole course being in the same space during their allocated timeslot, the whole project team is naturally available on the premises as XP recommends. The usage of remote communication tools and remote meeting scheduling becomes crucial, and the challenges for arranging effective meetings are similar as during remote work caused by the global COVID-19 pandemic 2020-2021.

If the course project is executed to create an experimental proof of concept, XP's ability to handle scope changes is advantageous: quick and unexpected changes of project scope or even underlying technologies can be handled following its processes.

As development and integration of work are done with short intervals, several times per day in a full-time work schedule, it naturally fits also time-constrained project work. Fast cycle allows smaller pieces of increments to be merged and followed.

**Disadvantages**

Requirement for continuous integration requires high knowledge of source control systems from all development team members. Such knowledge may not be present in a course project team. XP's frequent integration and work merging combined with the lack of source control knowledge increases the risk of merge conflicts. Resolving any mistake or conflict in source control could waste a developer's time. There are development tool stacks that automate CICD. However, their license costs could exceed the course project budget. On the other hand, knowledge of source control is essential for any software developer, and everyone working in development should be able to follow basic git processes.

Customer involvement is intense in XP: customer's representative should be available during every development session, full-time. The representative needs to be able to make decisions regarding the product and individual features without a committee while also having a clear vision of the priorities of all features.

Planning of XP sprint needs estimation of resources and required work effort. In a course project, estimates could be inaccurate, causing overpromising or underpromising. This makes the planning of development sprints difficult, and also development pace is difficult to sustain.

Refactoring could take a considerable amount of more knowledgeable individual's work time if everyone is working on all components in true XP fashion

### 4.3.2 Unity game development course project point of view

In addition to previously mentioned remarks, there are also conclusions specific to Unity game development.

**Advantages**

Artists can be integrated into the development team directly. With continuous integration, they could merge artwork into the product source control continuously, replacing the initial placeholders. With the whole team working together in the same space and using artist-programmer pair, new revisions of arts and visuals could be integrated quickly after finished, as well as tested if the end result was as expected. Customer presence in the team allows instant feedback on visuals and drafts.

Experimental game development is very prone to unexpected scope changes. XP welcomes the changes as part of its process.

Quick prototype and deployment cycle allows rapid testing with the customer as well as inspection of accomplished work.

**Disadvantages**

Specifically, with Unity, source control of scenarios and prefabrications is difficult due to the format of the files containing those objects. Conflict resolution and merging of scenarios and prefabrications is complex and undermines XP's strength of continuous integration: each scenario and prefabrication would need to be reserved for work per individual to avoid source control conflict.

## 4.4 Reflecting with Scrum as a framework

This section discusses the advantages and disadvantages of the Scrum framework in the context of a course project and further as a game development project.

### 4.4.1 Course project point of view

**Advantages**

The Scrum product increments and backlog items have specific measures when they are considered completed: when they fulfill the definition of done. After this point, the feature can be incrementally improved, however, it is not refined, and its code is not refactored like in Extreme Programming. This gives a clear overview of what has been accomplished.

The Scrum daily meeting questions, what has been done, what will be done next, and what impediments were found are beneficial for driving the development and learning from mistakes. It also gives a clear overview of who has accomplished what.

Scrum is presently the most commonly executed agile framework in software development. Learning its fundamentals is, therefore, beneficial to the course participants.

Product increments are integrated when they fulfill the definition of done. This also means that source control is not as hectic as in Extreme Programming, where work is merged even several times per day. In Scrum, the work can be merged when the definition of done is completed and/or for sprint review. The members of the Scrum team do not need to be as proficient with source control systems and a specialist could execute the work merges all at the same time before review. The amount of time wasted in resolving merging conflicts could therefore be reduced.

Characteristics of a specialization-based split of teams is built into Scrum, which encourages modular work practices. Unlike in Extreme Programming, the teams can focus on their specialized work and merge with other teams only for sprint review.

For experimental projects, it is possible to assign time and backlog items for knowledge acquisition. Even though completion of these backlog items does not directly increase

the value of the end product from the development point of view, they may still be requirements from stakeholders. It is possible to use a fail-fast strategy to abandon a feature or technology after research, before any development time has been committed to it, and thus saving resources. (Rubin 2012, 93-95)

**Weaknesses**

The project course runs only on certain days of the week, with a constrained amount of hours. This undermines one ritual of Scrum, the frequent, defined interaction and inspection point: daily meeting. If the course is only running once per week, the daily Scrum essentially becomes a weekly Scrum – while at the same time, sprint length would be a week due to risks associated with experimental projects.

Similarly, sprint review and retrospective meetings cannot be arranged in a Scrum fashion because the amount of work hours is constrained. Complete review, retrospective, and next sprint's planning could even take the full course project timeslot for the week.

Scrum does not cope as well as XP with the experimental project, where technology needs to be opened as development progresses: the initial planning phase in Scrum is expected to provide a defined outcome considering available resources, product architecture and broad view product design, after which they remain unchanged until the product release is completed.

Proper requirements acquisition, analysis, and planning would also consume time from the course schedule, while the development teams need tasks from the beginning. Therefore the initial planning phase, containing initial Product backlog creation, possible feasible technology options, and product vision as well as product goals, should be done beforehand. The initial backlog items could also contain tasks for investigating specific development technologies – however, if those technologies are not used, they would not create value for the product, and therefore, they would be wasted effort from the Scrum process point of view.

### 4.4.2 Unity game development course project point of view

**Advantages**

Unity architecture can be designed in the initial planning phase, which directs the development and object-oriented patterns. Unlike in Extreme Programming, where architecture would be designed as the development progresses, the design foundation created in the Scrum planning phase simplifies the development during sprints and reduces the amount of refactoring and rewriting of code.

With less frequent merging of work, the source control is easier than in Extreme Programming. Scenario and prefabrication source file's version control remains an issue that is specific to Unity, however, it can be mitigated slightly by splitting the teams into components based on product features: preferably, each scenario and prefabrication should only be worked by one team.

Unlike in Extreme Programming, the architectural design in the planning phase could take merging work of several teams into consideration, by minimizing the amount of commonly edited scenarios and prefabrications. This could be achieved by splitting them into smaller and smaller prefabrications until they are team-specific.

# 5 PROJECT 1 CASE STUDY

The case study was conducted by participating in all project management, planning, review, and presentation meetings and events. Observations were noted from each meeting. Each team was individually interviewed, and the Trello boards were reviewed. In addition, the extra source material was gathered from the instant messaging platform used by the course participants.

## 5.1 Introduction of project

The project was executed with a student team of 42 individuals, separated to teams based on expertise and skill. The team leaders were selected based on expertise. In addition, a project coordinator was selected to handle rolling matters, timetables and communication.

The student team had 16 hours of work allocated per week, and the project was scheduled to last 11 weeks. 8 hours of the weekly workload was scheduled for Monday, and the remaining 8 hours were left for students to organize themselves. The hours and work was monitored with individual work log.

The development team had a 2-week (32 hours allocated) Scrum practice prior to initiation of the project.

The project goal was to develop a prototype of a large educational VR game system, which could then be expanded and finalized in a continuation project. The development would be done using the game development tool Unity and programming language C#.

### 5.1.1 Scrum team

Product owner (external) - representing the customer and ensuring the product vision is followed. Available on Monday afternoons.

Developer (external) - ensuring that external technical requirements were followed and provided assistance in case of technical difficulties. Available on Monday afternoons.

Development team (internal) – There was a total of 7 teams, split by product component (scenarios). Each development team member had a specialization in artwork or programming. In addition, engineers were available for guidance during office hours.

Development team (external) – In addition, some parts of the graphics and integration were provided by the external development team, which did not work under project course constraints.

Table 1. Scrum artifacts and rituals in project 1.

| | |
|---|---|
| Daily meetings | No, due to scheduling and hour constraints |
| Sprint review | On Monday morning |
| Sprint planning | On Monday, following the review of previous sprint |
| Sprint retrospective | No, due to scheduling and hour constraints |
| Definition of Done | Was specified |
| Product planning | The initial Product backlog was not known |
| Product backlog | Individual teams |
| Sprint backlog | Individual teams |
| Backlog items | Individual teams |
| Sprint goal | Was specified |
| Product increments | Work was merged and released for milestones |

## 5.2 Observations

### 5.2.1 General

Sprints were concluded in a weekly basis, with longer-term goals (“milestones”) defined in advance. Combining several teams’ work each week was not possible with the constrained hour allocation, and sprint completion was approved in a per-component basis during demonstrations held every Monday morning.

Product backlog was written and maintained by individual student teams instead of Product owners, each with its own structure. Even though the individual backlog items were not reviewed, the sprints were reviewed based on the set sprint goal.

From the project management point of view, the responsibility of the feature set and prioritization was left to the development teams and pressured the development team leads.

Initial Product backlog would contain customer's strict requirements on technology, found during the planning phase. After these initial constraints are set, the Product owner should not be involved in practical implementation and technology used by the development team as long as it completes backlog items, user stories and produces expected product increment.

During the project, it was found that a specific set of deprecated versions needed to be used for integration with an external company's system. These library constraints are an example of requirements that would be found during the planning phase. Other similar constraints are licensing and outsourcing demands. In case there are such technical or external requirements, they should be documented as static constraints to the development process, which should be considered throughout the project. Implementing a feature that later would turn out to be incompatible with external requirements would essentially cause rework on the whole feature.

The Product owner role would be essential to monitor and focus the development efforts correctly and to maintain the Product backlog and backlog items, as well as sprint goals. In a large project consisting of several development teams with component split, the importance of Product backlog maintenance increases. The development team leads would not be able to each maintain a product vision separately while focusing on their component. The Product owner's role acts as a common point of contact for all components, maintaining an understanding of component interactions. These should be presented in the Product backlog as user stories, available for all component teams to review at all times.

Even though the definition of done was made without reviewable component specification, it is difficult to determine if the feature is done or in progress. This confusion increased the amount of features where implementation has started but has not finished.

### 5.2.2 Meetings and Scrum master

During reviews, all members of the project were generally present. Even though each team showed their work in turns, other teams and their leads were reserved, and did not proceed with development.

Each team could be individually slotted for inspection meetings, and other teams could be completely released to work on development while one group is in a meeting. These inspections and demonstrations could be arranged on different days, while main recommendation would be to split the Monday 8 hour day into several sessions over the week. The duration of the meeting should be kept to a minimum, recalling that the Daily Scrum meeting is not a problem-solving or a discussion. Those events can be arranged immediately afterward, releasing extra members to proceed with development.

Due to the lack of a Scrum Master, the development team leader was not isolated from the project management tasks. This affected the development team leader's ability to focus on the development, implementation, and testing of components. Overall the team lead was constrained to responsibility within their team's component, and it would be a waste of resources for each team lead to also design the interaction between components. This responsibility should be either exported to the Product owner or delegated to a dedicated development team member who is not the team lead. In a larger project, this role would be called Component designer, and it acts as a proxy for Product owner, while in the smaller project, it would be Product designer. As focus paths of course participants were programming and artwork, not project management, this role may be too out of the specialization path for a student team member.

Another notable downside caused by the lack of a Scrum Master was the delayed impediment detection and delayed corrective actions. With a Scrum master and daily meetings, impediment resolution could be faster and fewer resources would be wasted. With the project workhour constraints, additional meeting times should also be arranged during the week, outside of the assigned time slots. The project had an external resource of laboratory engineers at disposal for consultation, such resource should be eagerly used to preserve momentum in development since impediments raising in student team could be quickly resolved with the expertise of laboratory engineers.

With Sprint review on Monday morning, the majority of development tasks were done over the weekend or on Monday. Over the weekend, the members of the development team could not meet at the facilities nor use laboratory devices, resources or consultancy. In combination with Monday deadline pressure, this stressed the development team.

Due to infrequent meetings, the sprint length of one week was essentially a sprint length of 2-3 work days in a full-time employment environment. This is a very short sprint length in work hours, and for a large project with several component teams, it means a considerable effort of merging work together and testing. The focus of development should move from the creation of demonstrations towards the creation of functionality – the product increments, in a sustainable and extensible manner, following agile principles.

Without Product backlog and Sprint backlog items, it is, however, difficult to evaluate the progress of the Scrum team. As Scrum is currently the most used agile framework, it would be beneficial for the students to learn due diligence regarding development tasks and backlog item progress monitoring.

### 5.2.3 Tools

The development teams used Trello as a primary tool for organizing their work. Each development team had a different way of organizing their boards, however, which made inspections more complex. In addition, each student maintained a separate sheet of work hours and descriptions of individual tasks for the lecturer. The framework could incorporate a structured way of the Product backlog, project management, and task management while also encouraging usage of the tool through the project as a documentation tool for consumed work hours. If possible, these tools could be integrated under one platform to have a combined status of the whole project in one package. The platform could be Teams, as the project course is organized in Teams, and Teams has many options for integration of project management tools.

Unity scenario handling issues were apparent in the project. In many cases, work had to be redone, which wasted resources, especially time. As only 1 developer may work on any Unity scenario at any given time, the solution would be “reserving” of a scenario for editing per each developer when they work on the scenario, and “releasing” it after the work has been done.

Splitting scenarios into prefabrications should be considered when project architecture is designed.

## 5.3 Conclusions – Framework improvement

To resolve constraints and incorporate the improvement suggestions, the team member roles could be adjusted. The component task areas should be defined from the beginning and adjusted in case of impediments.

In student teams, there were two developer paths: programmers and artists. This naturally limits the duties that can be assigned regarding implementation work, however, any person from the team could be given the role of Scrum Master, Product owner proxy or Team lead.

These roles are key of the test framework that should drive the development team towards Scrum framework execution. In Scrum fashion, the development team could choose the persons fulfilling these roles, by itself – ensuring motivation and commitment, as well as following agile principles.

In this context, project size is defined as follows:

- if the project has several development teams, it is a large project.

- if the project has one development team, it is a small project.

Here, Scrum’s definition of the development team is used. Therefore the development team also contains the graphical artists. External resources and outsourcing are not considered separate development teams. Instead, their contribution is considered as a readily available component or asset.

### 5.3.1 Large project

Based on the review of Project 1, the large project framework can be refined with following adjustments.

The project's development teams would each be responsible for their component delivery and the product increments. This means that no single team member would be responsible for the achievements or failure of the team.

One of these responsibilities is integrating their component to the product's other components, which drives the team towards interface-based design and development, using software engineering design patterns.

In a large project, a specialized team for building an architecture and backend wireframe should be created. This team would then be aware of the integration process and manage the merging of work into a product increment. The team size can be a considerably smaller set of programmers with good knowledge of source control. It could also include artwork design and graphical guidance so that each component team's artwork is compatible with the style and requirements of the product.

At a team level, team members would be responsible for completing the tasks required to complete the user story of the component. It should be emphasized that these tasks should not be over-promised or under-promised, rather honest estimation of workload until acceptable result. The individual should quickly bring up both lack and excess of tasks before they become impediments of the component.

Usage of task board and minimizing the amount of work "in progress" would help the team members to balance their workload and also drive towards completing the tasks they start, not simply starting new tasks which linger.

The team should have a dedicated role of Component designer, who can maintain the vision of the component – acting as a Product owner's proxy for the component, based on the Product owner's vision and Product backlog. He would then construct a component backlog and driving the development team's focus on the component. The prioritization of the component backlog would be done using the expertise of the component development team as whole. This Product owner proxy role may be too distant from the specialization path and have too much individual responsibility, in which case it could be a member of staff, engineer, higher education student, or other senior.

The Scrum master of a large project should be one individual, without any implementation tasks. In a project course environment, this could be a member of the teaching staff to ensure focus, availability, and knowledge of the Scrum framework. Particular emphasis should be put on open and honest communication. There could still be a team member in a development team who focuses on finding impediments and organizing meetings.

Components form the product. As the students had a focus path of programming or graphical design, there are two options of separation: each component would also include the graphics of that component, or the graphics are considered as a component by itself.

In project 1, graphics per each main feature were considered as separate components, which created heavy constraints between teams. This has benefits and drawbacks: on the one hand, it encouraged interfacing in programming, as the developed feature was decoupled from artwork and should handle any changes of art in the future and same graphical design throughout the project's components. On the other hand, any impediments faced by the graphics team caused programmers to spend extra time with mockups and wireframes.

Another option would be the incorporation of graphics artists into the programmer team of each component. This would require a common graphical guidance to monitor the style and color scheme of the user interface and artwork but would naturally incorporate graphics as part of the component development cycle, encouraging decoupling of graphical and user interface features from program logic.

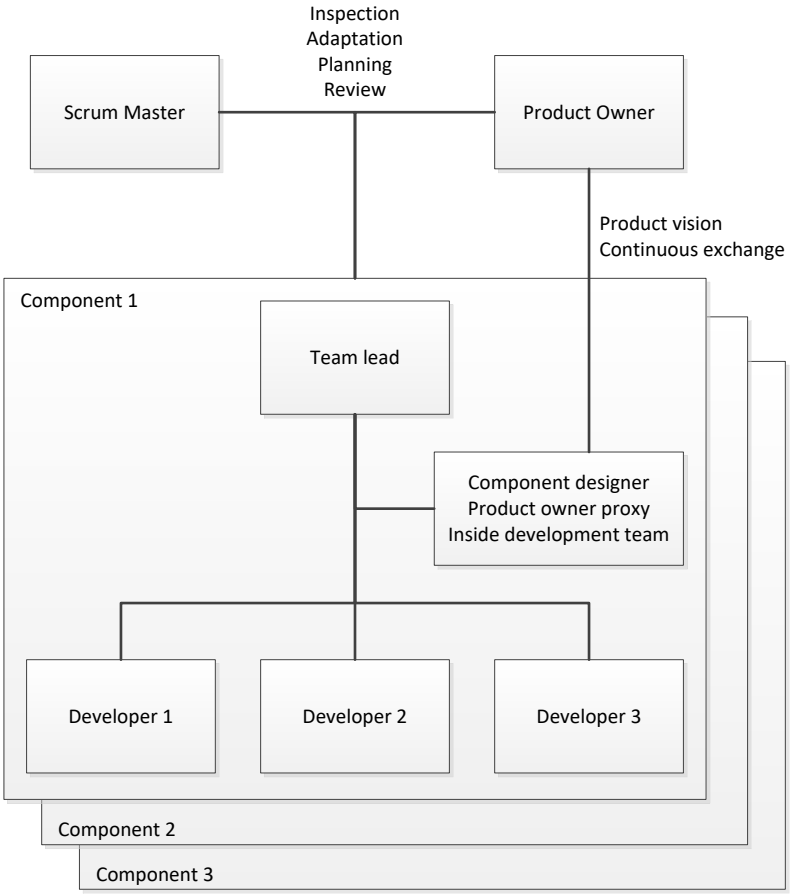


Figure 4. Communication paths in a large project with many component teams.

### 5.3.2 Small project

In a small project, each development team member would be considered responsible for their component and product increments associated with it. The responsible person is free to do prioritization within the component based on research and expertise he gains throughout the project cycle.

The roles and prioritization can be adjusted by the development team when resource reallocation is needed. The dedicated role a Product designer would maintain the vision of the product together with the Product owner. He would be one to write user stories and maintaining the Product backlog in case the Product owner does not have such resources or commitment. Each of the user stories written by the Product designer could be approved by the Product owner in a series. At the same time, their impact on value creation could also be discussed and documented.

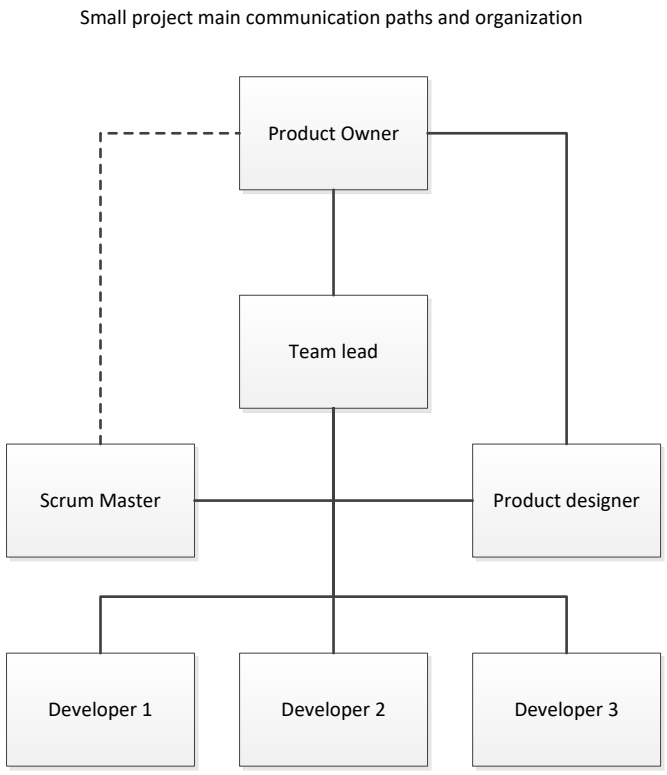


Figure 5. Communication paths in a small project.

In a small project, graphics artists are integrated as part of the development team. Graphics should be part of the version control cycle in such a way, that updating graphics would not cause changes in the program logic.

Using Unity, decoupling would be encouraged by leaving scenario design and construction to the graphical artist, splitting each functionality to separate prefabrication. Programmers would expose underlying logic and functionality for scenario and prefabrication elements, utilizing agile software architecture and design.

# 6 PROJECT 2 – TESTING AND IMPLEMENTATION

The case study was conducted by being a participant in the project in the role of a Product designer and a development team lead. This allowed close monitoring of all development team tasks and information flow between the Product owner and the development team. Also, Jira project management tool was maintained and updated throughout the project, allowing clear statistics on project progression and sprint velocity.

## 6.1 Introduction of the project

The project was executed with a student team of 7 individuals, out of which a Product owner proxy and Scrum master was selected.

The student team had 16 hours of work allocated per week, and the project was scheduled to last 7 weeks. 8 hours of the weekly workload was scheduled for Monday, and the remaining 8 hours were left for students to organize themselves. The hours and work were monitored with individual work logs.

The project goal was to develop a prototype of an educational mobile game with elements from machine learning and augmented reality. Also, a server backend would be used to collect research data about the players and their progress in the game. The project would be a kick-start proof of concept for a project which received funding and external resources. In addition, the project would be presented in an exhibition.

The development would be done using the game development tool Unity and programming language C#.

### 6.1.1 Scrum team

Product owner - ensuring the product vision is followed. Available for contact during office hours and participating in sprint meetings on Monday.

Development team – There was a total of 7 members. Each development team member had a specialization in artwork or programming. In addition, engineers were available for guidance during office hours.

Table 2. Scrum artifacts and rituals in project 2.

| | |
|---|---|
| Daily meetings | On Monday and Friday |
| Sprint review | On Monday after daily |
| Sprint retrospective | On Monday after review |
| Sprint planning | On Monday after retrospective |
| Definition of Done | Was specified |
| Product planning | The initial Product backlog was built |
| Product backlog | Maintained by Product owner proxy |
| Sprint backlog | Maintained by Product owner proxy |
| Backlog items | Team members, maintained by Product owner proxy |
| Sprint goal | Was specified |
| Product increments | Work was merged and released every 2nd week |

## 6.2 Scheduling of Scrum elements

As the time slot constraints in Project 2 would be the same as in Project 1, the test framework would utilize the same time slots. Test framework planning goal would be to define feasible Scrum rituals and practices with these constraints.

All these rituals would be facilitated and prepared by Scrum Master. For any changes to the agreements from the Product owner side, the Scrum Master would encourage direct communication between the development team and Product owner. Any tool, device, plugin, or software requirements emerging from the development team, would be put forward and handled by the Scrum Master. He would also be the contact point for any stakeholders and external personnel who wish to affect the work of the development team.

The goal of this workflow is to separate the development team from duties that are external to the implementation of the project or task assignment.

### 6.2.1 Daily meetings

Daily meetings would not be arranged daily due to the same time constraints as in Project 1.

Scheduling of Daily meetings was set up as follows, previously summarized in Table 2.

Monday 8:45am until 9am – Development team daily Scrum meeting, going through the three daily Scrum topics: what was done since the last meeting, what will be done today, what impediments or obstacles were present which hindered the progress.

Monday 9am until 10am – Meeting with the Product owner to verify game vision conforms with the development and prioritization planned with the development team. The meeting would be held in the same place as the development team daily meeting to minimize transition delays. The meeting area would be reserved for project duration in advance. Collective demonstrations were given to the Product owner only if there was a merged build. These builds would be prepared in two or three-week cycles. Additional builds would not be created during the sprint.

Monday 10am until 10:15am – Additional short meeting to adjust day's work based on the discussions with the Product owner. This meeting would repeat key points and expectations. Any uncertainties would be discussed here, and additional questions to the Product owner would be collected and sent to them by the Team lead. After this meeting, the development workday begins.

The focus of the meeting would be to keep their length at a minimum by following these guidelines:

- If a topic could be resolved in a subgroup, the other participants would be released to continue on development.
- After the meeting topic is discussed, the meeting should conclude with an outcome. Similarly, each meeting should start with an explicit definition of the goal of the meeting.
- Participants would be strongly encouraged to call a meeting at any time they think having one is beneficial and not wait until the next meeting.
- The participants would be strongly encouraged to resolve their impediments with a fail-fast mindset and quickly seek guidance from laboratory engineers

Meeting time is away from already very constrained development time. A practical solution should be valued higher, and feature sets should be locked as quickly as possible to have concrete goals.

Friday 11am – optional meeting to discuss impediments, progress, and constraints. This slot was pre-reserved for the duration of the project. The goal would be to motivate the progress of development over the week and adjust development resources. As an optional meeting, participation in this meeting would also give insight into the development team's motivation and provide a scheduled place for any feedback or topics uncomfortable for big group discussion where the Product owner, also a lecturer, was present. The meeting agenda would be the same as on Monday, a Daily Scrum followed by task distribution.

### 6.2.2 Sprint planning

With the same time constraints, Sprint planning meetings were arranged weekly. The test framework's focus is on the creation of backlog items and the construction of product and Sprint backlog. Sprint planning topics were discussed during the Monday meeting with the Product owner. With component-based separation, in a small project, each developer gave a delivery promise directly to the Product owner. In a large project, this promise was given collectively. These promises were mutually agreed on and then written down by the Team lead and Product/Component designer into a project management tool. The progress monitoring point of progress was on Friday, and inspection would follow on Monday.

### 6.2.3 Sprint Review

Sprint review was held weekly per component and every second or third week for the product as a whole. This was in sync with the builds created by the development team. Additional builds were not created, and for any additional demonstrations requiring the full product, the latest build was used. The Product owner accepted the sprint in the review of the backlog items fulfilling the definition of done. After the meeting, the Team lead marked the sprint as completed in the project management tool.

### 6.2.4 Sprint Retrospective

The retrospective meeting was planned for the end of Monday, at 4pm. It was planned to be with the development team only and focus on impediments and lessons learned over the previous week's sprint. In addition, its purpose was to act as a transition meeting from allocated scheduled development time to the more free solo work over the week.

Project 2 team and organization was built based on the Small project framework as described in Chapter 5.3.2. The time constraint in Project 2 was 7 weeks from initiation to delivery, and the team size was 7 students.

## 6.3 Observations and findings

The Product owner gave a briefing of the project and its background in a project start meeting a week prior to the start of the design and development phase. The project would be a prototype for a larger project, which would follow right after the proof of concept was delivered.

The project start meeting was arranged with all stakeholders present. The project start meeting was optional for the student development team, as their course had not yet started.

Product backlog and Sprint backlog would be maintained by the game designer regarding the user stories and team lead regarding the development team tasks. Prioritizing these items would be done according to the Product owner's feedback in weekly meetings. Due to time constraints, the game designer would switch to a developer role after 3 weeks of the project, after which the product would be feature locked and the remaining Product backlog prioritized. This would give a clear overview of the functionality that is planned for deliverable product, as well as limit Product backlog size to items which can be implemented in time.

During the project, similar to Project 1, the reserved development classroom was not utilized. Initially, an additional daily meeting was organized at the end of a workday to review and recap all that was accomplished during the day. This meeting was arranged in the classroom to encourage usage of it for the project work and to encourage more communication and pair-programming.

### 6.3.1 Meetings

The meetings were organized in two days instead of one day, with a specific repeating schedule.

Monday

08:45 – 09:00 – Development team meeting in Scrum daily meeting, arranged as planned.

09:00 – 10:00 – Project team meeting with the development team, external resources and Product owner: demonstration of work done during the week to the Product owner, feedback and planning of next week. Also, product and Sprint backlogs were updated at this point according to the Product owner’s decisions and feedback.

10:00 – 10:30 – As needed, development team meeting for discussing the technical details regarding features and giving task assignments. During this meeting, the game designer and team lead changed the statuses of backlog items from Ideas to Open and removed ones which the Product owner did not wish to be part of the product.

16:00 – 16:15 – This meeting was later canceled when the feature set was locked, and responsibilities regarding components were clear for everyone.

Friday

10:00 – 10:30 – Development team meeting, optional. If any of the team members anonymously wished to have this meeting, then it would be reserved and team lead + Scrum master would be present. These meetings were arranged every week, and it was found to be very useful for checking the progress of tasks, proactive inspection called from the team members' side. Also, with full Friday and weekend still available at this point, resources could still be reallocated if a developer struggled to implement the planned feature for the Monday demonstration.

### 6.3.2 Tools

User stories and sprint goals should be documented in a tool capable of handling complex task relations. The tool prepared for the user stories, tasks, and backlogs would be Atlassian Jira, a professional tool for Scrum development. As the thesis goal was to evaluate also project management tools, Project 2 used Jira to compare its usefulness to the relatively lightweight Trello board used in Project 1.

Product backlog, Sprint backlog, and Backlog items were maintained in Atlassian Jira, utilizing the Scrum development template. The system did not have a specific template for game design. Hence default template was used with modification: in addition to the normal Scrum Product backlog item states; Open, In Progress, Done; an extra Idea state was added. These backlog items would be ones which the game designer could suggest to the Product owner as possible new features during the next weekly meeting. If approved, those Ideas would be changed into Open items, indicating they are approved by the Product owner and would be part of the development cycle.

Time constraints caused miscommunication of sprint goals and requirements. With only one meeting per week, non-documented details can easily get lost. Even though the Scrum framework minimizes documentation, flowcharts of basic product feature loops and component interconnections could provide an overview of the product and additional references for developers. It could also be used as a visual method of finding missing features and on the other hand, a way to present to the Product owner which parts of the product have been developed and to lock down certain sprint features over the week

### 6.3.3 Communication

The development team occasionally developed features that were not in user stories. They were generally approved by the Product owner in the next meeting. However, these items were not in planned sprints and caused a loss of resources from agreed features.

The Product owner was very active during demonstration sessions, with clear vision indicated by definite and quick responses to feature expansions and current status. Also, the direction on prioritizing was clear. The sprint planning and review sessions often went overtime.

Occasionally scheduled backlog items and tasks were forgotten, even though they were assigned in the Jira system. Also, the Jira system password was occasionally forgotten, while the system provides a tool to recover it. All in all, these occurrences show unwillingness to use Jira as part of the work routines. Jira system was unknown to the development team and seen as too heavy for a small agile project.

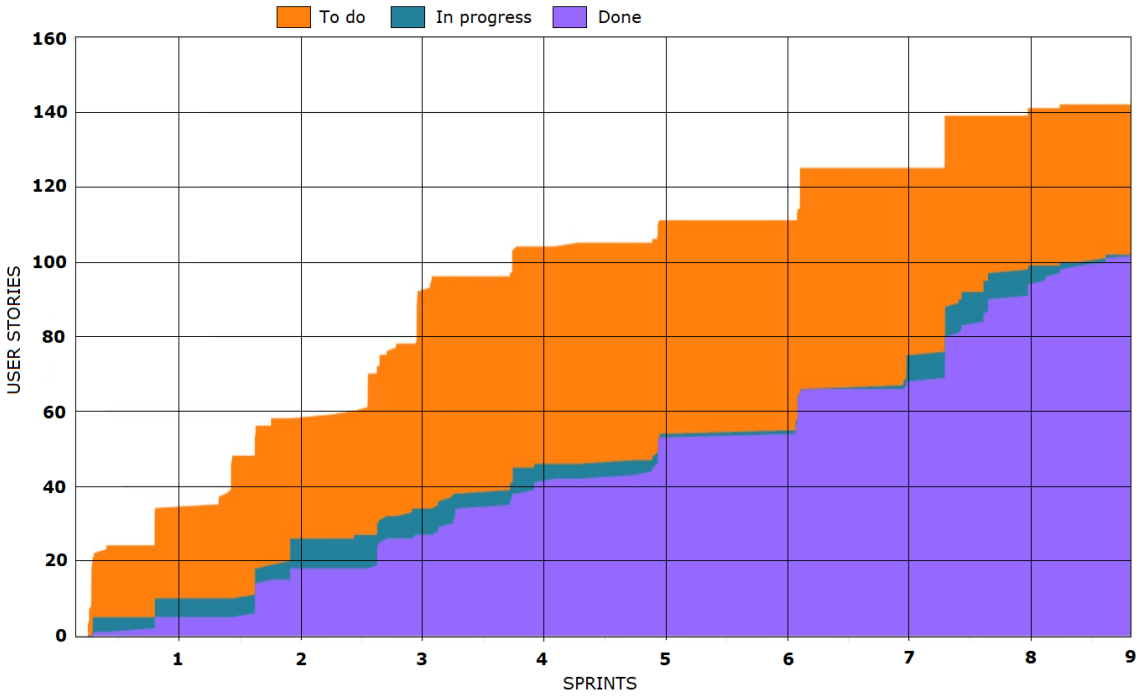


Figure 6. Project 2 progress through sprints.

The backlog maintenance was done on Monday, which can be observed as jumps in backlog items and their completion (Figure 6). The programming team systematically overpromised their sprint deliveries over the project's development duration.

Overreaching and overestimation of capabilities, as well as under-promising and uncertainty of abilities, occurred throughout the project duration (Figure 7). This was similar to the observation during Project 1. In the other hand, these impediments were caught much faster due to the addition of more Scrum meetings, and schedules and resources could be adjusted accordingly.

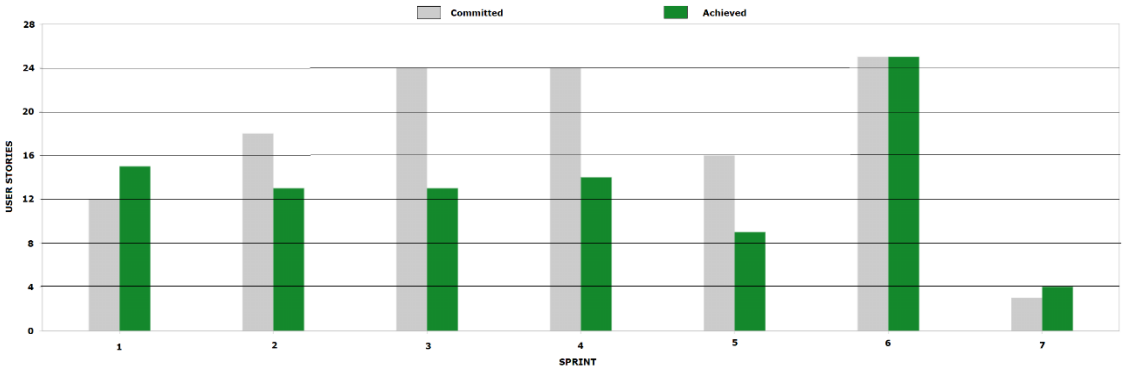


Figure 7. Project 2 programming team promises and achievements.

One of the development team members was working on another exhibition project in parallel as part of another course, and in addition, had two other projects within the laboratory. All of the projects were Unity development projects, and this noticeably hindered the individual's capabilities to focus and work on development tasks at hand.

Tendency to avoid individual communication was observed in Project 2 in a similar way it was observed in Project 1. Taking ownership of individual issues was not eagerly done, and the team lead was eagerly contacted to act as a relay. The Scrum Master should encourage individual direct communication and the habit of informing the team lead of the outcome instead of seeking approval or relay.

Pair programming in the laboratory and computer classroom was encouraged as much as possible. This way, developers of two components would be able to link their components on the fly, and also share their knowledge and understanding without spending time to write down explicit documentation. Extreme Programming techniques for sharing the keyboard could be used here, with the coder changing whenever the function being worked on changes.

## 6.4 Refinement

This chapter reflects observations and findings on how the ideas presented in project 1 carried over to the actual realization of project 2, what impact they had and what challenges were faced.

### 6.4.1 Scheduling

Issue: Monday afternoon computer classroom allocation was rarely used, and only a few or none were present at the end of the day. In general, allocation of half of the work hours to a single day at the beginning of the week seems to encourage working only on Monday and weekend. With a single day allocated, the meetings group together, and inspection of progress can only be done once per week. In the test framework, this allocation was found to affect project progress negatively. Even with an additional voluntary meeting on Friday, the scheduling and work hour allocation would encourage working on weekends or solo instead of working as a team.

Proposal: to encourage teamwork as a group and to encourage the progress of the project during the whole week, the work hour allocation should be split into two-morning slots. There two schedules would encourage the Scrum framework:

- The Monday and Friday timetable schedule, where each sprint could then start on Monday and end on Friday. This would add additional inspection points, add a full team daily meeting, and separate Sprint planning and Sprint review meetings. The workweek would start with planning and end with a review. After initial phases and depending on project uncertainties, the sprint length could be increased to two weeks. This way, every other week would be complete development week, with only daily Scrum meetings as the start of the day.

- The Tuesday and Thursday timetable schedule, where sprint length would be two weeks, with the planning and the review on Tuesday. This way, the students could arrange part of the non-scheduled part of the work hours to Monday and encourage teamwork instead of working solo over the weekend. Similarly, if the sprint goal was not reached on Thursday, there would still be Friday and Monday to work on it as a group before review on Tuesday.

Further testing would be needed to see the effectiveness of this split. The timetable's feasibility depends on the student group’s other courses’ allocations.

### 6.4.2 Task lists and timetables not maintained

Timetables contain a list of work hours and how they were used. The work descriptions contain tasks related to Product backlog items. This means redundancy is present, the same information is written into timetables and task lists.

Proposal: The students are required to log their work hours into a timetable sheet. Integrating the timetable sheet into the project management tool would avoid marking the same hours and tasks twice, encouraging proper documentation. This allows a collective review of progress and workload.

Atlassian Jira was not used by the student team. It was observed to be too heavy for a small project.

Proposal: both Trello and Jira can be integrated into the Microsoft Teams environment, which is widely used in the execution of information technology courses. Also, the students are familiar with the environment. As a common platform supporting direct integration, it could act like a commonplace to store all project boards, project management, and task management aspects of the course. This way also external Product owners would only need to have one invitation to join the project – and see all the project boards as well as demonstrations.

### 6.4.3 Scope and requirements acquisition unclear

Initial research and technology stack design uses a considerable amount of work hours at the beginning of experimental proof of concept -type projects.

Proposal: project initiation could be executed as an intensive week with team workshops. At the end of the week, the decision of go-for design idea could be made, and the project course could get a kickstart. As an alternative, the Product owner should be present especially in the early stages and deliver the product vision to the team. Research and experimentation of technologies should be done with a fail-fast mindset. A solution is better than no solution. Here, the project deadline and milestones should be used. The focus should always be in implementation and value creation.

### 6.4.4 Version control is confusing

Issue: Git version control system is not known to the students, especially to graphical artists. Accidents occurred where work was overwritten, or branching was executed in the wrong way, requiring resolving and extra work.

Proposal: Version control courses should be created or incorporated into programmer courses in the early stages. Git is widely used in the industry, and learning it from the start would also encourage the creation of a Git-based portfolio, a collection of student works, and a journey as a developer. In large projects, the version control flow should be kept simple and clean following the best practices of Git. Components could be founded in the same version control repository to have all material available for everyone and included in the version control, including the artwork. This would only be possible when students are familiar with the version control platform at sufficient proficiency.

### 6.4.5 Demonstrations and builds

Demonstration builds require a merge of all development team’s work. This in turn, needs knowledge of Git version control and a properly maintained repository to avoid conflicts.

Proposal 1: student team could have varying levels of Git expertise, which considerably affects the overhead caused by merging work together into a presentable build with features integrated together.

Based on observations over Project 1 and Project 2, the build interval of these merges should be minimum 2 weeks, preferably longer. Builds should be saved to provide a clear progress path that can be reviewed if necessary.

When modular architecture is developed, and in place, demonstration builds can be created faster, as there are fewer dependencies between components. Until the architecture is created, however, the demonstrations should be done without a build via Unity.

Proposal 2: On several occasions, the team's best developer was allocated for half of the day to merge the work of others in the version control system, followed by the creation of a build for presentation. In case a build is required, the most recent build should always be given. No additional builds would be done, and the next one would be available as the build interval dictates. Individual components could be demonstrated without build, as described previously.

### 6.4.6 Parallel projects overlap

Issue: student development team member may have many projects in parallel, all with similar deadlines and development stack

Proposal: it could be beneficial for the project and the individual if he could focus on a maximum of three similar projects. Where synergies are present, the tasks could be incorporated together. This would also encourage the creation of reusable assets and more focus on design patterns.

# 7 CONCLUSIONS

The thesis was commissioned in order to observe and evaluate the current execution of Scrum in game development course projects and to create a framework for future course projects as well as recommend changes to course roles, schedule, tools and execution.

As a result, two frameworks were created: one for a large project where several teams work towards the creation of a single product, and the other for a small project where one team works alone for a product.

The framework was tested for the small project and the result was the successful realization of the Product owner's vision as well as the documentation of the path from initiation to the final demonstration.

During the observations, various best practices were noted, and a best practices toolkit was collected as a knowledge base for future project groups.

## 7.1 Agile framework selection key factors

Some details of the observed projects were known in advance:

- The projects were **experimental and innovative**, opening new technology for future projects and probing for new **high-risk** prospects
- **Time** would be constrained, both by deadline as well as weekly working hours and days
- The **expertise** and amount of experience of the development team varied
- The project's technical **requirements** were not known in advance
- The project's **scope** and feature set was not known in advance
- The project **goal** was a proof of concept, working prototype
- Development **costs** would be low
- Customer **feedback** was available once per week

While Extreme Programming prioritizes fast implementation, it also requires a certain level of expertise. In an experimental project with proof of concept as a goal, Extreme Programming would be a natural choice, as it allows change of scope and design throughout the project. Its execution, however, requires the continuous presence of the customer as well as the cycling of developers through different components. Both are time-intensive requirements, not easily managed with course constraints.

Scrum, on the other hand, depends on product planning and design, which is set before development cycles begin. In addition, the requirement for meetings at regular intervals is present, which would not be achieved. It would not be possible to have daily meetings when the project is scheduled only for two days of the week.

## 7.2 Observations

The first major impediment to Scrum framework execution during both observed projects, was that the Product owner did not maintain a Product backlog, and did not write backlog items. In Project 2, this was partly mitigated by the introduction of a proxy for the Product owner. Comparing the two student-driven projects, the usage of proxy was observed to support the Scrum framework much better than not having backlog maintenance at all. With the Product owner's proxy, and maintained backlog with backlog items, the Scrum framework execution can be done. This role may be too distant from the specialization paths, which concentrate on development team roles: programming and artwork. Neither of the paths specializes in Product ownership or project management. Instead of using the time of an already specialized development team member for project management, an external resource could be used. Perhaps a member of software project management specialization path or a student of higher education.

The team member in this proxy role should be able to communicate actively with the Product owner throughout the project, and act as a vision keeper within the team, with last word on the order of backlog items and implementation prioritization. It is imperative that any uncertainties of the proxy are resolved with the Product owner to avoid rollback and rework of already implemented functionality.

Proxies of Product owners are used in the industry. However a proxy does not address the underlying issue, which is the time allocation of the Product owner who does not have time or resources to maintain the backlog or its items. A proxy causes separation of responsibility, and holds no final decision power. It, therefore, cannot fully replace a dedicated Product owner.

To conclude, the role of Product owner proxy is highly beneficial when the Product owner does not maintain backlog or the items or has low availability in general.

The second major impediment was the time constraint and schedule of the work. With 8 hours of allocated group work per week, especially when that work is scheduled for one day, the sprints become erratic and impediment resolution is delayed due to lack of meetings and inspection. Student attendance to the scheduled hours after lunch break was also arbitrary. To conclude, the 8 hour allocation should be split into at least two 4 hour allocations in the mornings. This will allow for two daily meetings per week, and also separation of sprint planning and sprint review meetings. Two group allocations may also help to guide the timing and usage of independent work hours. To kick-start a project, an intensive week could be a solution, but this would not be possible in the middle of a semester.

The third impediment was the student's knowledge of agile as a methodology and mindset. The students knew the Scrum framework roles and rituals but based on observations, did not understand the agile reasoning behind them. There was a conflict: heavy constraint in the work hours, which is not present in any theoretical implementation of Scrum, versus the daily rituals present in Scrum guidelines. This conflict turned execution into a mixture of XP, Scrum, and survival.

The teaching of agile software development patterns, especially interfaces, decoupling, and separation of program logic from interface and dependency reversal, should be encouraged. In experimental projects, individual parts of the project could change arbitrarily. Therefore good architecture supporting modular design and flexible changes are beneficial. The time constraints weigh heavily in architecture and design also.

The collection of found best practices could be founded as a deep knowledge library. Companies in the industry generally have a knowledge base of their project impediments and past crisis solutions, as well as guidelines if a similar crisis would appear again in another project.

The fourth impediment to Scrum execution was the lack of development task management. Individual development tasks were not written down, except in the individual work hour log. The filling of those timesheets was arbitrary and often delayed, sometimes even several weeks. The Jira system did not have the expected effect on the logging of work hours, mainly due to the observed complexity of the tool. This may be due to missing integration with other tools, mainly Microsoft Teams. To conclude, the tool containing the Product backlog items should also include a task list related to that backlog item. Each member of the development team could also mark down the work hours spent on the task, which would both reduce redundancy and help with workload estimation per development team member. Finally, this tool should be integrated into Microsoft Teams so that all parts of the project and its management can be in one place.

The fifth impediment was general communication. Agile depends on open and honest feedback and speaking out about any impediments. During both projects, team members had difficulty seeking help from laboratory experts in due time. This could be due to lack of experience and inability to recognize when to fail-fast and when to press on. More inspection points from two separate day's daily meetings could encourage the team members to speak up about constraints and worries faster. The key point could be recognition that other team member's work will become constrained by one team's delays. Peer pressure may be effective here.

Many of the above impediments could also appear due to remote work, in case the COVID-19 situation causes regulations that limit the amount of group work in school premises. The offered resolutions could increase inspection and control options for such scenarios.

## 7.3 Practices discovered during the projects

During the projects, technical processes were discovered that could be generalized for future student projects as recommendations:

The usage of a Kanban for Unity projects: each development team member reserves a scenario or prefabrication when they start working on it, and releases it after changes are done and in version control. This process eliminates situations where development work is rolled back due to conflicting scenarios and prefabrication files in version control. The practical implementation of this process would be a task board containing cards for each scenario and optionally prefabrication. The developers would assign themselves to the card when they started working on a scenario, and remove themselves after the would have been concluded. Any scenario without an assigned developer would then be free for another developer to work on.

- Git version control: usage of rebase instead of merge helps to keep Git clean and in addition, reduces commit overhead. As a downside, rebase is a potentially destructive operation as it rewrites commit history, and graphical Git interfaces are not always behaving in a similar intuitive manner manner. Git command prompt usage is encouraged by this, and knowledge of it would become useful when working in the industry. Reliance on a specific graphical interface is a constraint.
- Git version control: usage of build and release branches helps to keep track of project progress. With this branching model, after work has been merged into a release, all branches could be reset --hard and started anew from the merged commit. All branches would then have a common base again, and the repository is cleaner.
- Git version control: feature branches would encourage co-operation within the same branch, and this also prepares students for future work situations where feature branching is a predetermined system. Several developers may work in the same branch, and that should be practiced.
- Trello board: plug-ins “Scrum by Vince” and “Scaled” were tested and found to be useful for Scrum project organization. The downside is the constraint caused by the free Trello license, which only allows one plug-in per board.

- Microsoft Teams: Scrum framework could be incorporated into Teams by using Planner boards. They work in a similar fashion to Trello boards and could be incorporated into the school user groups. For further improvement research, automatic notifications could be added to certain actions, such as assignment to a card or completion of Sprint, which could then be sent to Discord or other instant messengers via API.
- Product backlog items could be linked to a version control repository commit, which contains the feature implementation. In this way, the development team would have common documentation on commit where the feature is implemented.

## 7.4 Final words

Due to the COVID-19 pandemic which started right after presentation of the small project, a large project could not be tested in the same setting. The usage of Teams as an integration platform for Trello boards was, however, incorporated into the remotely executed course.

The integration of different tools was found to be important to keep track of project progress. All of the tools have support for integration and together offer support for continuous development and integration. If all tools were integrated together, including version control, compilation, and building, it would allow seamless access for the Product owner as well.

The frameworks are created to mitigate two major impediments of course projects: time constraints from course schedule, and corresponding lack of Product owner availability and feedback.

A heavily limited group work schedule condensed to a single day, with the remaining work unallocated for the rest of the week, shares characteristics with remote work. Remote leadership and management techniques, as well as further integration of tools could help to create a virtual work group and encourage co-operation.

The experimental nature of the projects during the courses would benefit from the focused intense project week model, rather than single scheduled lecture timeslots in weekly schedule. This way also Extreme Programming could be tested as an agile framework for innovative experimental research projects. The planning and design phase of Scrum would not be possible in such projects due to their nature.

During the observations and research, it was noticed that the team leader of the course project team carried several responsibilities regarding both project management and product design that were not part of the specialization path. The Product owner proxy role assignment to a specialized team member remains to be tested.

The students of the observed projects seemed to struggle with the same issues. A deep, growing knowledge base for best practices and solutions of the specialization path could be founded, to store them for future groups. Writing an increment into a knowledge base as a team, could act as a final retrospective of the project. Such a document could be stored in a version control platform which naturally merges the work of different classes and teams.

The students of the Project Management path could be incorporated for project management tasks. Similarly Master students could be incorporated as a Product owner proxy role, architecture or design role or as development team leaders.

## 7.5 Recommendations for future research

The proxy Product owners seem to be considered as a negative feature in the industry, adding an extra layer decision-making, based on Scrum Alliance blog posts. Research on the topic may answer a demand.

Time-constrained execution of agile frameworks using teams with low work experience levels seems to be a new research topic that could be explored.

The integration of tools into an umbrella framework would allow usage of Continuous Integration and Continuous Development. This could remove issues regarding demonstration builds and version control conflicts, as well as integrate different teams at framework level. One such framework is DevOps by Atlassian, which merges several of their products together to support a full development pipeline from innovation and planning to development, deployment and live maintenance.

# REFERENCES


Atlassian (2021a). Trello Guide. Available at: https://trello.com/guide , cited 10.5.2021

Atlassian (2021b). Jira. Available at: https://www.atlassian.com/software/jira , cited 10.5.2021

Beck, K. & Andres, C. (2004). Extreme programming explained: embrace change, Addison-Wesley, Boston, MA.

Beck, K., Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas. D. (2001a). Manifesto for Agile Software Development. Available at http://agilemanifesto.org/ , Cited 10.05.2021

Beck, K., Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas. D. (2001b). Manifesto for Agile Software Development, Principles. Available at http://agilemanifesto.org/principles.html , Cited 10.05.2021

Eveleens, J. L. and Verhoef, C. (2010) “The rise and fall of the Chaos report figures,” *IEEE software*, 27(1), pp. 30–36.

Git (2021). Git Reference manual. Available at: https://git-scm.com/docs , cited 10.5.2021

Glass, R. L. (2005) “IT Failure Rates - 70% or 10-15%?,” *IEEE software*, 22(3), pp. 112, 110–111.

Highsmith, J. (2001). History: The Agile Manifesto. available at: http://agilemanifesto.org/history.html , Cited 10.5.2021

Hohl, P., Klünder, J., van Bennekum, A., Lockard, R., Gifford, J., Münch, J., Stupperich, M. and Schneider, K. (2018). Back to the future: origins and directions of the “Agile Manifesto” – views of the originators. Journal of Software Engineering Research and Development, 6(1).

Johnson, J., Mulder, H. and Group, S. (1994). THE STANDISH GROUP REPORT 1994.

Laanti, M., Similä, J. and Abrahamsson, P. (2013) “Definitions of agile software development and agility,” in *Communications in Computer and Information Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 247–258.

Martin, R. C. (2002) *Agile software development, principles, patterns, and practices*. Upper Saddle River, NJ: Pearson.

Matkovic, P. and Tumbas, P. (2010) A Comparative Overview of the Evolution of Software Development Models. *Journal of Industrial Engineering and Management,* **1**, pp. 163-172.

Pressman, R. S. (2014) *Software Engineering: A Practitioner’s Approach*. 8th ed. Maidenhead, England: McGraw Hill Higher Education.

Royce, W. W. (1987) “Managing the development of large software systems: concepts and techniques.”

Rubin, K. S. (2012) *Essential Scrum: A practical guide to the most popular agile process*. Boston, MA: Addison-Wesley Educational.

Schwaber, K. (1997) “SCRUM development process,” in *Business Object Design and Implementation*. London: Springer London, pp. 117–134.

Schwaber, K., Sutherland, J. (2020) The Scrum Guide™. Available at https://Scrumguides.org/docs/Scrumguide/v2020/2020-Scrum-Guide-US.pdf, Cited 20.3.2021

Sochova, Z. (2016) *The Great ScrumMaster*. Philadelphia, PA: Pearson Education.

Sommerville, I. (2004) *Software Engineering*. 7th ed. Boston, MA: Addison-Wesley Educational.

Unity Technologies (2020a), Unity Documentation. Available at: https://docs.unity3d.com/Manual/index.html , cited 10.5.2021

Unity Technologies (2020b), Unity Documentation. Available at: https://docs.unity3d.com/Manual/FormatDescription.html , cited 10.5.2021

Unity Technologies (2020c), Unity Documentation. Available at: https://docs.unity3d.com/Manual/CreatingScenes.html , cited 10.5.2021

Unity Technologies (2020d), Unity Documentation. Available at: https://docs.unity3d.com/Manual/Prefabs.html , cited 10.5.2021

Watts, G. (2013) *Scrum mastery: From good to great servant leadership*. Cheltenham, England: Inspect & Adapt.