Project plan+study diary

JangliGame

version 1.4

TUT	Pervasive Computing	TIE-21106 Software Engineering Methodology
Author: Vitsikkäät vekkulit		Printed: 29.04.2018 18:34
Distribution:		
Document status: Final		Modified: 29.04.2018 18:34

VERSION HISTORY

Version	Date	Authors	Explanation (modifications)
1.0	23.1.2018	Daniel N	End of Sprint 0 Version
1.1	16.2.2018	Daniel N	End of Sprint 1 Version
1.2	8.3.2018	Daniel N	End of Sprint 2 Version
1.3.	8.4.2018	Atte L, Leevi K	End of Sprint 3 Version
1.4.	29.4.2018	Atte L, Leevi K	End of Sprint 4 Version

TABLE OF CONTENTS

1. PROJECT RESOURCES

This chapter holds the project resources.

1.1 Personnel

SCRUM MASTER – Atte Lamminsalo
GIT MASTER – Daniel Natri
PRODUCT OWNER and COURT COMPOSER – Leevi Kulju
UNITY BLASTER – Henri Laakso

Contact information:
atte.lamminsalo@student.tut.fi
daniel.natri@student.tut.fi
leevi.kulju@student.tut.fi
henri.m.laakso@student.tut.fi

1.2 Process description

Our project goals are to make a simple game of Jungle Hunt. Our sprints end on Sunday. Our main virtual communication channel is Telegram.

Below is our rough schedule for the project requirements per sprint. We also plan to meet every Thursday at 18:00. In these meetings we will agree on who will do and what during next week. Every time we start implementing a new user story, we will divide it to tasks, if not already done. Whenever possible, group members will code together together. Project will be split into four sprints. The first sprint starts on 8.1. and the fourth sprint ends on 29.4. Pre-determined user stories will do for milestones.

Our team assigns themselves tasks from Agilefant based on what can be done at the moment. If a user story is clearly a package one person can handle by themselves, the team member can assign the whole story to themselves as well. The SCRUM master checks to see if the tasks are evenly divided and possibly forces tasks on lazy team members.

Unfinished user stories are moved to the backlog and usually into the next sprint.

1.2.1 Definition of done

Definition of done can be approached in various ways. One approach would be, when can the developer mark “Done” for a feature in Agilefant. Another could be, when the customer is totally happy with the feature, and is not planning on changing it in the near future. We decided to define the former, as cases where the customer is sure he is happy with something are rare (in the real world). We feel a more concrete approach with the developer finishing a feature is more appropriate. We choose to define a feature done, when it is successfully merged into the master branch, thus being ready for deployment. If changes are going to be made to the same feature, it will be a new story, and it will be defined done separately.

1.2.2 Effort estimation

Effort estimation is a tricky subject for any software project. A certain task can take a longer time to complete for one than the other. We decided to cut down each task into small enough pieces to determine as closely as possible how long each bigger story will take to complete. We estimate these small components in a best case scenario. What we noticed was that these best case scenarios often might not be accurate for the single story, but they balance out in the long run.

1.3 Tools and technologies

1.3.1 Agilefant

URL: https://app.agilefant.com/TTY-TIE/product/376289/tree

User stories have already been saved to Agilefant, and they have been divided to tasks. Each story and task have been labeled with a number and a name to make it easier to the group to see what is going on in git. When a group member picks up a task, he will do an effort estimation as section 1.2.2. describes, and mark it down in Agilefant.

1.3.2 Version control

We are following a master-feature version control schema on our project. The master branch is supposed to always be a working copy of the program. Developers create feature branches for new features, for example *feature/add_score_system* that is merged onto the master branch once fully implemented. This is done in Gitlab via merge request. The merge request is inspected by a team member, and only merged once it has been accepted. This ensures that development is never halted due to a broken copy of the program. If a broken version is merged into the master branch, a quick bugfix branch, e.g. *bugfix/fix_score_system* should be implemented as soon as possible. This is visualized in picture 1.

If new versions of tools we are using come up, we only update the master branch once it is clearly evident that the updates do not break the project.

Commit messages should be in a passive tense and clearly determine what has been done in English.

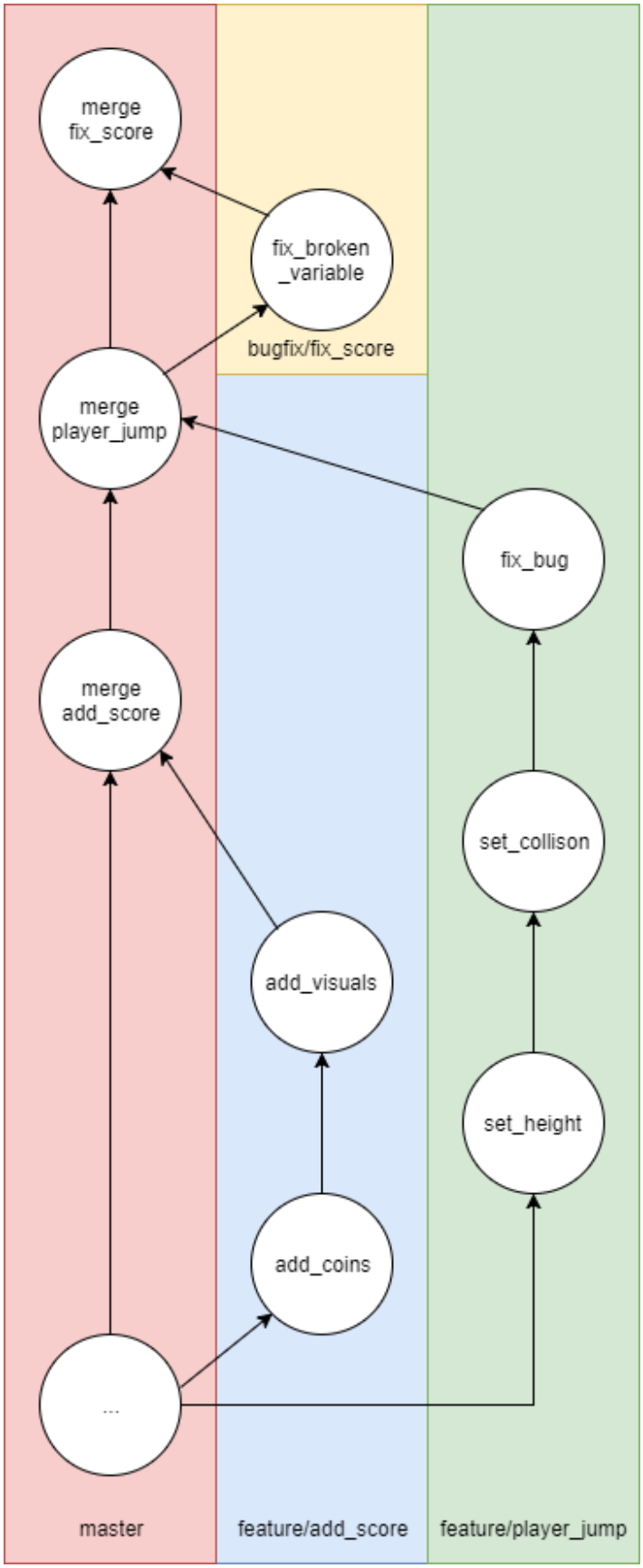
Good examples:

- “Add enemy hitboxes.”
- “Remove broken platform.”
- “Bugfix for swimming component.”

Bad examples:

- “lisätty vettä”
- “commit”
- “bugfix….”
- “merge damn it”

We opted not to include story numbers in commits, but instead in the feature branches. It should be clear from the branch names, what the feature in question is.



Picture 1. Master-feature schema

Table 1.1: Tools used in the project.

Purpose	Tool	Contact person
Documentation	MS Word (word processing) office.microsoft.com	D.N.
	UMLet	L.K.
Communication	Telegram	A. L.
Version manage-ment	Git	A.L.
Game engine	Unity3D	H.L.

1.4 Testing

Testing is a key component in every software project. If we would've had more time we would have also made fully functional unit and integration tests. These tests would be implemented in Gitlab as scripts that are run every time you commit upstream. This would allow the developers to notice bugs as soon as they are pushed, instead of a long way down the line, where they may be forgotten.

We added a very bare-boned gitlab_ci.yml tester in our repository. The tests require a runner, that we didn't have time to setup. The basic idea of the runner is that you can set up a testing machine that runs the tests for you. The machine does not have to be a physical machine, it can also be a virtual machine running for example in a Docker container. Docker also allows for multiple containers running runners, which allows for different setups and/or parallel testing of different commits.

1.4.1 Unit tests

One of the tests would be a unit test. The idea of unit testing is to isolate functions and test them on their own. This forces the developers not to make monolithic functions and it keeps the source code readable and functional.

1.4.2 Integration tests

Integration testing happens right after unit testing. This combines different modules and ensures that the code works as a whole. Different testing methods test up-to-down and down-to-up, ensuring that the code is logical.

1.4.3 Code inspections

On top of automatic testing, it is also wise to do manual inspections to the code. This way you can not only find bugs in the code, but also

situations where the code is running perfectly, but the character might get stuck etc. Human testing is a vital component for a successful software, and automatic testing is only there to weed out the most obvious technical bugs. Humans are there to get rid of the unplayable aspects of the game.

1.5 Sprint Backlog

Sprint backlogs can be found in groups Agilefant (1.3.1). For each sprint, the tasks that have not be marked with the label "Done" represent the sprints backlog.

2. STUDY DIARY

This chapter holds your journal of lessons learned during the course. That is, **more detailed analysis of previous Sprint's contents**.

2.1 Sprint 1 (every sprint as a section)

2.1.1 What went well

We got the skeleton for the Unity project and learned to use the game engine. Team members were introduced to the toolchain. We were also able to create a basic game with a controllable player. Our group meetings were productive. We had fun. :)

2.1.2 What difficulties you had

We had understood the scrum stories a bit wrong. We underestimated the required time for learning to use the tools. The risks we found were not accurate enough.

2.1.3 What were the main learnings

We learned that it is quite difficult to estimate the required time to complete a task. Weekly group meetings helped in continuous progression in project development and allowed us to share knowledge and solve problems as a group.

2.1.4 What did you decide to change for the next sprint

We will improve our user stories and do more planning for our project. We will also do more risk management and continue to evaluate current risks. We also updated our risk management to be more accurate. Version control and process description got updated. We decided to continue user stories that we didn't complete in sprint 1.

2.2 Sprint 2

2.2.1 What went well

We managed to complete almost all the tasks on the list. Some of the stories were harder than the others. However, we did not have time to merge all the stories to the master branch.

2.2.2 What difficulties you had

We spent a lot of time in testing the components of the program. Some persons from our group had time off because of sickness and stress. There might have been some overestimation in work quality and resources. Merging same unity scenes was difficult and it took more time than we expected and caused a lot of "*mergeles!*".

2.2.3 What were the main learnings

We learned how to use unity and project tools better. Also implementing stories became clearer. Seems like every sprint we can estimate the required time better, but we should always overestimate the required time for a task. We talked about implementing a test script using Gitlab CI, but we opted not to due to time constraints.

2.2.4 What did you decide to change for the next sprint

We will pick tasks more wisely so we can ensure the completion of the tasks as well as merge the changes in time. We also have to work more throughout the sprint instead of just half of the sprint to maximize code quality.

2.3 Sprint 3

2.3.1 What went well

We were able to clean up all the branches into a single master branch for the first time in this project (since the start). It feels good to be able to test all the different features at once. We also were able to catch up with the documentation that was lacking from previous sprints.

2.3.2 What difficulties you had

Team communication was sub-par. We were not able to complete most of the tasks given. The programming was understandably left at a smaller role for this sprint, because we had a good head start from the beginning of the course.

During sprint 3, we encountered difficulties similar the risks introduced in chapter 3. Group members have not been committed enough due to heavy workloads from other projects and courses. This could be said to be both long- and short-term absence.

2.3.3 What were the main learnings

The value of good communication is unmeasurable. A lot of our team members had a lot on their plates in Sprint 3. Not only was the team

unable to complete most of the stories, also any attempt from the SCRUM master to communicate in Telegram often went unnoticed.

After the end of sprint 2, we managed to discuss more about how agilefant should be used in our process, and more suitable naming practices concerning git. We aggreed, that each story will be given a number, as well as each task. This way it is easier to keep track of the git workflow. ("3.6: Add collisions that use function to remove lives" VS. "Add bg materials") These tags can be seen in both git and agilefant.

2.3.4 What did you decide to change for the next sprint

Prioritize work on our game and weekly meetings, and communication. This is a group work, not an individual project. The group needs to pull together, as it is unfair for the team members who actually do stuff.

2.4 Sprint 4

2.4.1 What went well

Sprint 4 was an especially good sprint, because we could focus mainly on polishing our project. We were able to finish all our tasks. We were able to make a fun game, that has an epic soundtrack! We could also focus on testing and inspecting the game, now that it was mostly fully completed.

2.4.2 What difficulties you had

Time management is yet again an issue for our team. Luckily, we still managed to make the required features. The issues we had with time management are probably issues that are dealt with in work life, and it has truly been an eye opener for self-improvement.

2.4.3 What were the main learnings

We learned that testing is key to a successful merge. Unfortunately, this was learned the hard way in many cases. Peer reviewing code, even just by running it on another computer often brings out bugs. If it works on one computer, it might not work on every computer.

2.4.4 What did you decide to change for the next sprint

Because there is no next sprint, we are focusing on the end gala. We are going to record a play-through of our game that we can show to the other students and teachers.

3. RISK MANAGEMENT PLAN

CHANGES FOR SPRINT 2: The risks were re-evaluated and adjusted to be more realistic. No changes to risk management plan

The most probable and impacting risk that can happen during the process is insufficient planning which leads to increased workload. Other possible risks and their consequences are described below.

Table 4.1: Project risks.

Risk ID	Description	Probability	Impact
1	Short term absence	3	1
2	Insufficient planning	3	2
3	Long term absence	1	2
4	Changed customer requirements	3	2
5	Communication problems	1	1
6	Overconfidence in own skills	3	2

3.1.1 Short term absence of one person

It is very likely that someone will get sick during the course. Impact of this is directly dependent on the length and severity of absence. In some cases, short term absence can be negated by online meetings if possible. We can minimize the damage these absences cause by not scheduling our work to last days of sprints and doing all the work in time.

3.1.2 Insufficient planning

It is very likely to have problems caused by insufficient planning because we are learning new skills and our evaluation skills regarding the tasks may not be adequate. Severity of these problems can be anything from total catastrophe to small inconvenience. Most of these problems can be avoided by allocating enough time for planning and sticking to said plans.
Insufficient planning covers all possible problems that might occur regarding required time for certain task. This covers inability to learn and use new technologies according to plan or general extra time spent over estimated.

3.1.3 Long term absence

This is almost the same as short term absence, but this also includes someone of the team quitting. It is very likely that someone will get sick during the course, but not so likely that someone will quit the course. Impact of this is increased workload on the other remaining colleagues. In some cases long term absence can be negated by online meetings if possible. We can minimize the damage these absences cause by not scheduling our work to last days of sprints and doing all the work in time. Also workload should be somewhat equal so no one gets burned out.

3.1.4 Changed customer requirements

Customer might change his requirements which leads to more work and even possibly failure to complete the project. This can be avoided by constant communication with customer and ensuring our product is developed in the right direction.

3.1.5 Communication problems

We might be unable to reach one person of the group for some time which causes some inconveniences for the whole group but these inconveniences should be minimal. Probability for this to happen is very low.

3.1.6 Overconfidence in own skills

A developer can easily overestimate his or her skills technology-wise. This can cause speed bumps in development and serious time loss. The amount of overconfidence can vary in size which causes problems ranging from negligible to problems in delivering user stories.