

# Agile development of computer games

Version 1.8. ---- 28.3.24 – tidy up of the document

Version 1.7 ---- 7. 2.24 – Template for requirements

Version 1.6 ---- 26. 1.24 – GameObject dictionary

Version 1.5 ---- 22. 1.24 – requirements: Tip parameters specified

Version 1.4 ---- 14. 1.24 – adding planning poker

Version 1.3 ---- 1. 1.24

## Reference Link to Module Handbook: Advanced Practical (BSc,MSc)

[https://www.informatik.uni-](https://www.informatik.uni-heidelberg.de/images/pruef_modul/MHB_DataCompScie_MSc_WS202122_neuePO.pdf)

[heidelberg.de/images/pruef\\_modul/MHB\\_DataCompScie\\_MSc\\_WS202122\\_neuePO.pdf](https://www.informatik.uni-heidelberg.de/images/pruef_modul/MHB_DataCompScie_MSc_WS202122_neuePO.pdf)

IMP: Advanced Practical (with teams of 3 students)

8 ECTS

240 h – 25h in presence (online meetings), 10 h for preparation of the presentation

Assessment of the project results (software, documentation), the project report (10+ pages per team member), and the presentation (approx. 30 minutes plus discussion)

Passing the module examination

Since you are working in a team of 3 and since we are organizing the contact meetings online over heiconf:

<https://heiconf.uni-heidelberg.de/hes-32w-u4v> the rules are modified:

12h for preparation of joint presentation

36h for preparation of joint report (one report for the team)

- The length of the report is typically 10-20 pages: details later in this document

### Timeline:

Phase 1: use-case, UML-diagrams and requirements document: 24h

Phase 2: SCRUM/agile development: 168h

= 10 bi-weekly meetings

Phase 3: report and presentation: 48h (36h+12h)

## Reference Link to Module Handbook: Anfängerpraktikum (BSc)

[https://www.informatik.uni-](https://www.informatik.uni-heidelberg.de/c/image/f/default/pdfs/mhb2023/MHB_Informatik_BSc_100_2023-24_WiSe.pdf)

[heidelberg.de/c/image/f/default/pdfs/mhb2023/MHB\\_Informatik\\_BSc\\_100\\_2023-24\\_WiSe.pdf](https://www.informatik.uni-heidelberg.de/c/image/f/default/pdfs/mhb2023/MHB_Informatik_BSc_100_2023-24_WiSe.pdf)

IAP: Anfängerpraktikum (with teams of 3 students)

6 ECTS

180 h – 15h in presence (online meetings), 10 h for preparation of the presentation

Das Modul wird mit einer benoteten Prüfung abgeschlossen. Diese Prüfung umfasst die Bewertung der dokumentierten Software, des Projektberichts (ca. 5 Seiten pro Teammitglied) und des Vortrags (ca. 30 Minuten zzgl. Diskussion). Zur Vergabe der LP muss diese Prüfung bestanden werden. Die Modulendnote wird durch die Note der Prüfung festgelegt.

### Timeline:

Phase 1: use-case, UML-diagrams and requirements document: 24h

Phase 2: SCRUM/agile development: 108h

=6 bi-weekly meetings

Phase 3: report and presentation: 48h (36h+12h)

## Use of the document:

Consider the document as a guide. **Please read it carefully** and adhere to the concepts described below, in particular the processes and the structure of the report. In case of questions: ask!

The document provides many helpful tips and links.

## Bachelor/Master

This internship can be used for the bachelor or master program, for the bachelor program it can count also for “Anfängerpraktikum” – in the latter case there are separate regulations if some of the team want to take this option. There is an extra document for this case.

## Timeline

The internship is planned to stretch at least over 10 weeks since 200 h equals  $200/8 \approx 25$  days @8 hrs. If you plan 1 day per week (~8 h) to work on this internship and when we meet every two weeks, this would then stretch over about 5-6 months. The shortest time would be a weekly meeting and nearly 3 days per week you invest for this internship.

It is not recommended to stretch it over more than 7-8 months since the teams often cannot stay together so long time.

## ToC

1. Objective of the internship:.....	7
2. To the organization .....	8
2.1. Management: .....	9
2.2. Trello:.....	10
2.3. Git: (e.g. GitHub).....	12
2.4. discord: .....	13
2.5. Comments .....	14
2.5.1. Trello.....	14
2.5.2. Final Meetings .....	14
3. Important criteria for evaluation .....	15
4. Formal results .....	16
5. The initial steps: 24h .....	17
5.1. Creating a storyboard/use case: 8h.....	17
5.1.1. storyboard: 4h .....	17
5.1.2. UML diagrams: 4h .....	18
5.1.3. Create requirements analysis 8h.....	22
5.2. Develop and document architecture/class diagram: 8h .....	25
5.3. SCRUM:.....	27
5.3.1. Development process.....	27
5.3.2. Gradual realization of this process:.....	28
5.3.3. Contact meetings every two weeks key aspects .....	30
5.3.4. Contact meetings agenda points .....	31
5.3.5. Structure of YOUR planning meeting: .....	32
5.3.6. Table for Sprint Meetings .....	33
5.3.7. Process change table .....	33
5.4. Time estimate .....	36
5.5. Test-Driven Design - a must!!!! .....	37
5.5.1. Recommendations.....	38
5.5.2. Code review .....	41
6. Report: 10-20 pages ~36h .....	42
6.1. Scope .....	42
6.2. Contents .....	43

6.2.1.	Chapter 1 - Introduction & Basics: .....	43
6.2.2.	Chapter 2- Use case, requirements analysis and architecture (each sub-chapter) .....	43
6.2.3.	Chapter 3- Implementation .....	43
6.2.4.	Chapter 4- Development Process (should cover about half of the report).....	43
6.2.5.	Main focus: These two components MUST be used to successfully accomplish the internship .....	44
6.2.6.	Attachment/Appendix DO NOT FORGET!! .....	45
7.	Presentation ~12h .....	46
8.	Tips and experiences .....	47

## 1. Objective of the internship:

A:

Try out game-specific **software development processes** in a "real laboratory" with a team of ideally **3 members**.

Comment:

If you cannot find a team of 3, then the absolute minimum is 2. More than 3 team members is not considered.

Important point: You should further develop a given start-up process that you implement exactly so that you no longer have problems with the development in the course of the project, that is neither

- communication problems
- what is tested and documented as requirements
- no programming errors - 0 errors programming due to
  - good processes
  - extensive testing
  - code review processes
- clean and repeatable programming
  - therefore do not use scripts but the offered programming languages like C++ or C#
    - Reason: scripts offer only a limited repertoire of data structures and programming features such as anonymous functions, complex inheritance hierarchies with abstract interfaces, generic programming
    - Scripts get unstable once the task is too complex or there is a lot of code
    - Scripts are used whenever flexibility is at premium

In other words, we interpret each problem as a problem of the software development process and hence adjust the process individually.

B:

Learn how to use and extend a commercial game engine (Unity, Unreal Engine, Godot or others). There is an excellent documentation of these engines on the respective web-pages!

C:

Learning how to define an appropriate architecture for a given game that has the characteristics like flexibility, modularity etc.

D:

efficiently solve a large software project in a team.

At the end of the internship you should have mastered a development process so that you have implemented the zero-error strategy and have also implemented your computer game with the most important elements. You can later use this experience for other projects.

## 2. To the organization

We meet at the beginning (physically or virtually (e.g. via heiconf: <https://heiconf.uni-heidelberg.de/hes-32w-u4v> )) for the first meeting - you have already received this information by email. Here I discuss the important steps described below.

Then create a 1) Trello<sup>1</sup> (which is a Kanban board that is free of charge for personal use – as it is here) and a 2) GitLab/GitHub<sup>2</sup> (which is again free of charge for personal use) account. You may also have a 3) discord<sup>3</sup> account.

---

<sup>1</sup> [www.trello.com](http://www.trello.com)

<sup>2</sup> [www.github.com](http://www.github.com)

<sup>3</sup> [www.discord.com](http://www.discord.com)



### 2.1. Management:

We meet at the beginning (kick-off) and at critical points, such as storyboard, requirements document, architecture and if necessary, if there are any questions. I always have a look at Trello to see if there are any problems - possibly a separate meeting. At the end before the report we meet again.

## 2.2. Trello:

Use the **Kanban** structure, where at least four columns are used – from left to right:

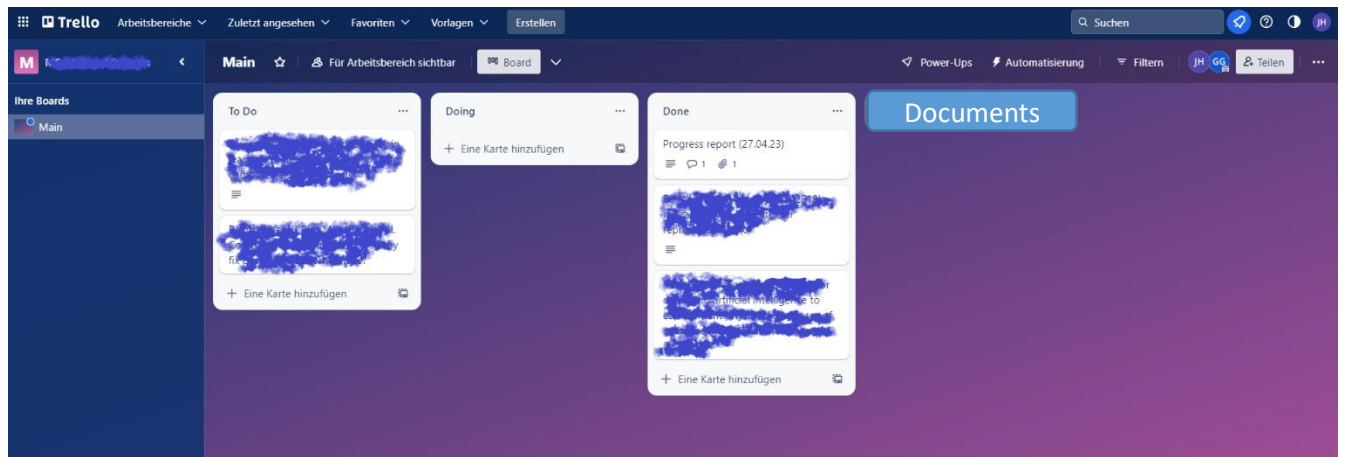
- backlog
  - contains all tasks that are still to be processed
    - each task is a short part that should be programmed
    - it is fulfilling a part of the requirements (a good strategy is to mention which functional requirement is solved by the task)
    - it is embedded in the class diagram, i.e. the part of code that is developed might be a class or only one method of the class
- (sprint optionally)
  - contains the tasks to be processed in the sprint, in the order in which they were processed
- active
  - on programming days, the top card is taken from Sprint and moved to Active
  - There can be more information here, such as a link to the git or the requirements
  - if the card has not been completely processed, they end up back on the sprint - right at the top
- done
  - If a card in Active has been successfully processed, it lands on Done and is therefore complete
  - if it later turns out that an error has occurred that has to do with a processed card, it ends up back on Sprint and right at the top

I typically suggest to use an extra column for

- use-case and diagrams to the use case such as interaction diagram, use-case diagram, state diagram
- requirements analysis document (not only a link to wherever you write the details, but also a PDF document where I can quickly see what are the requirements and if they are clear, complete, verifiable and quantitative)
- Architecture – in this case a class diagram
  - With abstract classes whenever this makes sense
  - Programming patterns such as manager classes (singletons), factory, strategy or façade pattern....
  - Components to let the number of methods and attributes in a class to be manageable

You can name the columns differently or add more columns. Purpose of Trello: Coordination of the project, assignment of tasks and schedule (possibly with a checklist).

With Trello, you can reference the Git directly. But you can also use different other options such as attaching files (e.g. your documentation, use-case, requirements analysis etc.), but also to insert checkboxes etc.



### 2.3. Git: (e.g. GitHub)

This is a) the version control b) the place where the documentation is (versioned) c) the place where all the development tasks are as tickets (e.g. prototype, test code, code).

You can use the wiki there to store general information that is directly related to the code and not to project management.

Git uses a wiki where you can organize the programming issues that are distinct to the organization. For example, you might track the ongoing, collect detected errors with ticketing, you install a CI/CD pipeline and automatic unit testing. You can also version documents.

Git is not considered for managing larger chunks of data – there are specific solutions for that issue.

- Use version control for the code
  - initially build a minimal version that serves as a deployment reference - can be quite minimal that it just shows an empty window without any function.
  - Everyone who checks out first builds their own branch and when it runs, it is transferred back to the main branch
    - this is tested by running the unit tests all successfully
    - This is then documented in such a way that everyone is informed
  - each unit test is run again with each transfer
  - if it turns out that there are still bugs, the unit tests will need to be updated to include that test
    - a ticket is created in GitLab/GitHub and processed
  - Ideally, refactoring is not necessary with a structured approach:
    - if it turns out that something has been forgotten, this must first be documented and then the refactoring carried out
    - The same goes for the requirement: if it turns out that requirements are missing or imprecise, they must be specified, then it must be checked whether the architecture still needs to be updated and then move on
    - Ditto for the architecture - which, however, is defined as early as possible with regard to interfaces so that no refactorings are necessary
  - You can use the wiki in GitLab/GitHub
  - You can automatically run the unit tests before check-in so that it doesn't forget.
  - Trello has a GitHub plugin that you can use to manage GitHub from Trello.

Unity also offers a version control system - that would be the alternative but you can also use the standard git – to use less memory, you can use the gitignore option.

Take your time to better understand the possibilities of the given version control systems.

Visual Studio offers a connection to Unity:

<https://devblogs.microsoft.com/cppblog/whats-new-for-cpp-developers-in-visual-studio-2022-17-9/>

#### 2.4. discord:

Communication tool - alternatively Slack or MS Teams.

## 2.5. Comments

### 2.5.1. Trello

In Trello I will then comment on the individual cards directly in the course of the internship.

Use the opportunity there to create checklists and the like. to use. Use the opportunity to learn how to optimally use Trello as a tool for management and organization.

Write the tasks and problems for it or links to the git.

We meet every 2 weeks via heiconf.

And in particular for

- use case discussion
- Requirements Document Discussion
- Architecture Discussion

Store this information in the trello – such as a dashboard so that we all can easily access all relevant information at one spot.

After that we meet every 2 weeks for a SCRUM meeting

- Discussion: previous plan: what was achieved, what not: current status
- Discussion: what went well, what didn't go so well: improvement of one's own process - see also the procedure below
- Discussion: next task list

### 2.5.2. Final Meetings

At the end there is a meeting for the report and before the presentation

- I emphasize again the essential things of the report or the presentation
- I look at the versions beforehand before they are submitted or presented

Once that's done and I agree with the latest version of the report, everyone gives a presentation of about 30 minutes, where they present

- goal of the project
- use case
- Requirements: what and why
- Architecture: how and why
- implementation
- SCRUM agile principle: as realized
- short presentation of the game with its features

When the lecture is finished, please send me the report and lecture and the internship is finished.

### 3. Important criteria for evaluation

1. the individual steps are well documented such as use-case/game storyboard, requirements analysis, architecture, documentation (including that of the tests), correct use of tools such as Git (version control, automatic testing with unit tests), Trello and correct administration with Kanban board are taken into account .
  - a. it is expected that the above points will be processed in accordance with the previous courses you had in your study of computer science
2. then it is checked to what extent the goal of agile development of a game has been implemented, i.e.:
  - a. Did you implement the process correctly?
  - b. you have significantly improved the process
3. Documentation of the internship: Report according to the template below: clear, understandable structure, no experience report, but an objective and factual description of the procedure

#### Quality of the report

- a. Presentation (legible, understandable, correct, appropriateness of language)
  - b. Content: describes the main points as mentioned above
4. Presentation according to the contents of the documentation:
  - a. Clear and understandable presentation of the content (no spelling mistakes, etc.)
  - b. Slides are informative and well structured
  - c. Content: the essential points of the report are described in the presentation
  - d. Demonstration of the game (online)
5. Quality of the individual steps
  - a. Clean Code
  - b. Tests (quality and completeness)
  - c. Documentation (quality and completeness)
  - d. Class Diagram – good abstraction, use of programming patterns where useful, modularity, simplicity
  - e. Requirements Document – quality, completeness, verifiability, quantitative statements

PS To be on the safe side, I look at the report and slides beforehand, but please don't get out of hand, i.e. the number of iterations should be closer to 1. Hence the structure that I specify, so that nothing could actually be done on my part when it was carefully implemented.

## 4. Formal results

1. Report of approx. 10-20 pages see below + all documents (use-case including interaction diagram, requirements beginning and end, class diagram), code, documentation, test-document with test code
2. presentation of results
  - a. Objectives of the internship
  - b. development steps
  - c. result of the game
  - d. Agile development: what is the concept, how is it realized



## 5. The initial steps: 24h

### 5.1. Creating a storyboard/use case: 8h

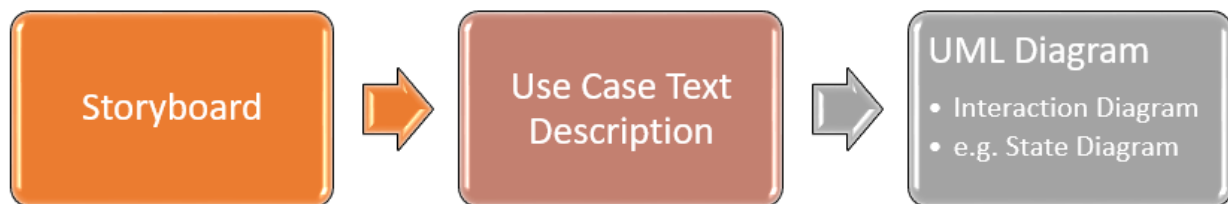
For the use case/storyboard, the following items should be included:

1. storyboard
2. UML diagrams (see UML-diagrams)
  - in any case in use-case diagram
  - an activity or interaction diagram - or both
  - possibly a state machine diagram
  - further diagrams are possible if they still characterize the game well

And do not forget to describe the use-case as a text (e.g. ½ p) to explain the game and its mechanics, objectives etc. These details are necessary to understand what you plan and what should be considered in the game. You can get this information by first formulating it in a storyboard and then distill from this text UML diagrams that help structuring the requirements that have to be formulated later.

Hence the typical sequence is:

- 1) formulate your game by sketches on a storyboard
- 2) from this storyboard write a description of the game that is generally understandable and allows others to understand what you want to realized
- 3) from this text, you structure the information via UML diagrams, which help to see whether you have a complete picture of the game mechanics that have to be implemented: a good strategy is the interaction diagram, but sometimes state diagrams might be helpful as well.



#### 5.1.1. storyboard: 4h

A storyboard is used to characterize the core graphic elements of a game, e.g. the appearance of game characters, the environment, etc. It can also be used to represent the content of a game very well.

good tool: <https://www.stormboard.com/>

<https://www.picmention.de/blog/storyboard-erstellen-zeichnen-162/>

<https://www.referencefilm.de/filmwissen/storyboard/storyboard-erstellen/>

<https://www.dummies.com/programming/programming-games/designing-video-games/>

<https://gamescrye.com/blog/how-to-storyboard-your-game/>

<https://gamestorming.com/storyboard/>

<https://itstillworks.com/12196599/how-to-design-a-game-storyboard>

<https://www.cs.cornell.edu/courses/cs3152/2013sp/labs/design1/>

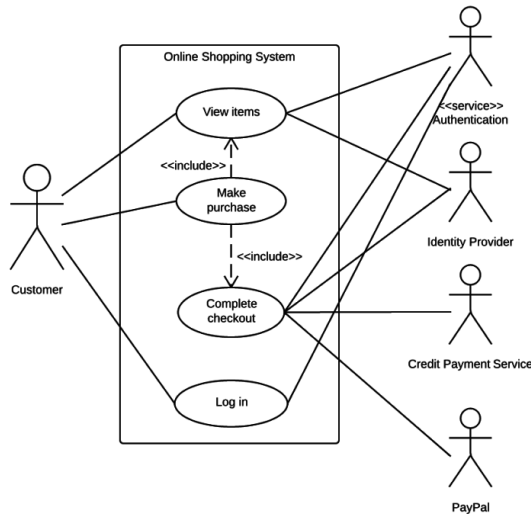
### 5.1.2. UML diagrams: 4h

It has turned out that different UML views are well suited to formulating the use-cases, including:

(see also <https://creately.com/blog/diagrams/uml-diagram-types-examples/>

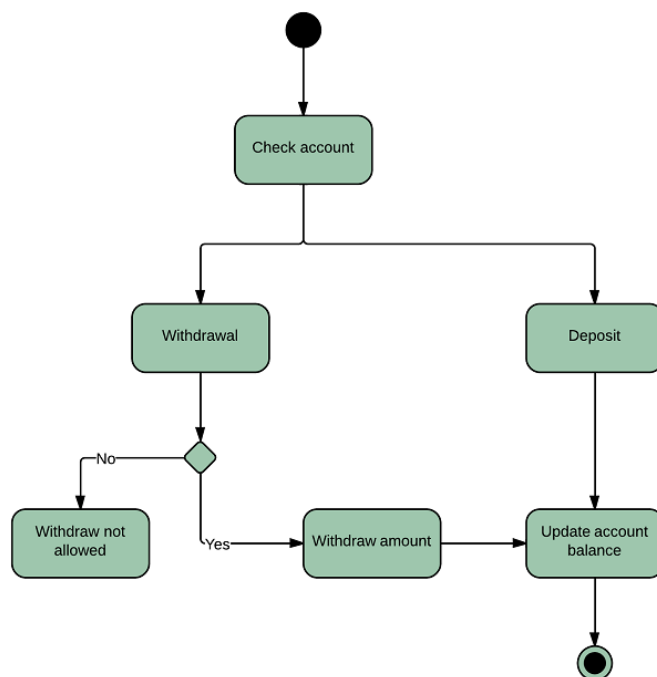
A good example is also: <https://www.visual-paradigm.com/> )

- use-case diagram: roughly describes the interaction between players and the components of the game



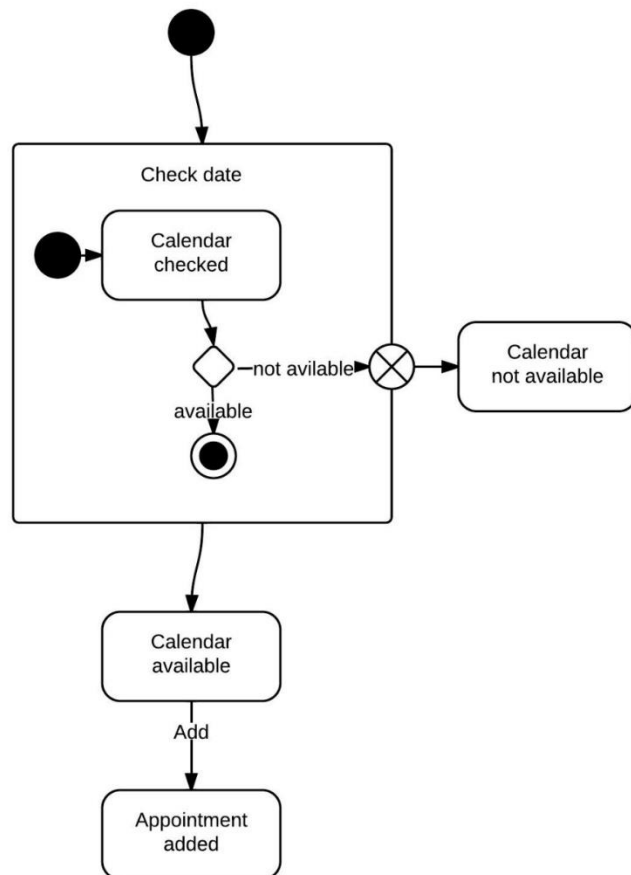
<https://www.lucidchart.com/pages/uml-use-case-diagram>

- activity diagram: describes well a game mechanic from the point of view of the processes

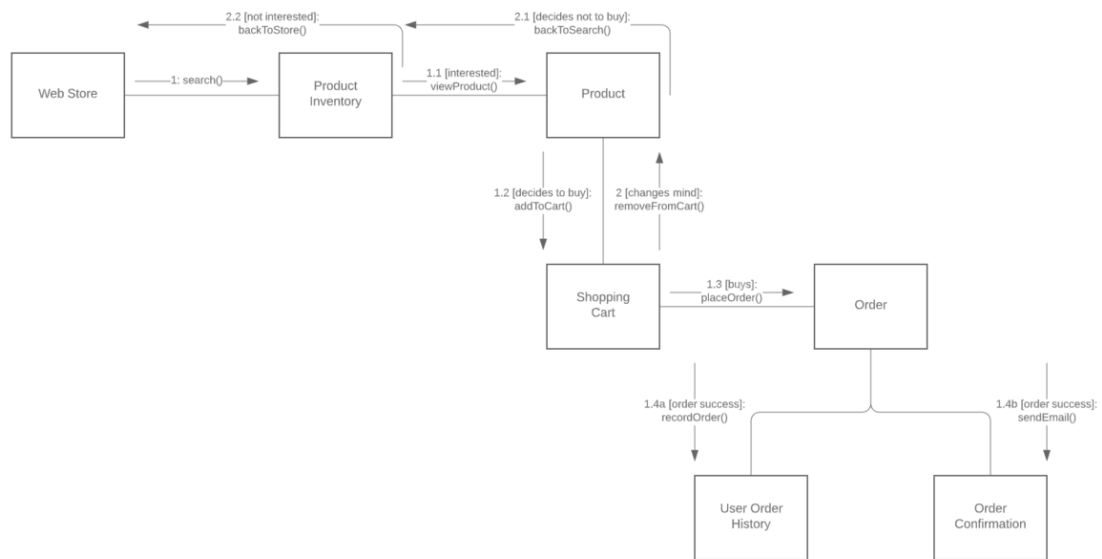


<https://www.lucidchart.com/pages/uml-activity-diagram>

- state machine diagram:  
defines state machines that can be easily integrated into the game and formally describe the course of the game so that nothing is forgotten - many errors are based on something being created but no longer destroyed, something being opened but no longer closed; this can be seen quickly in a state machine

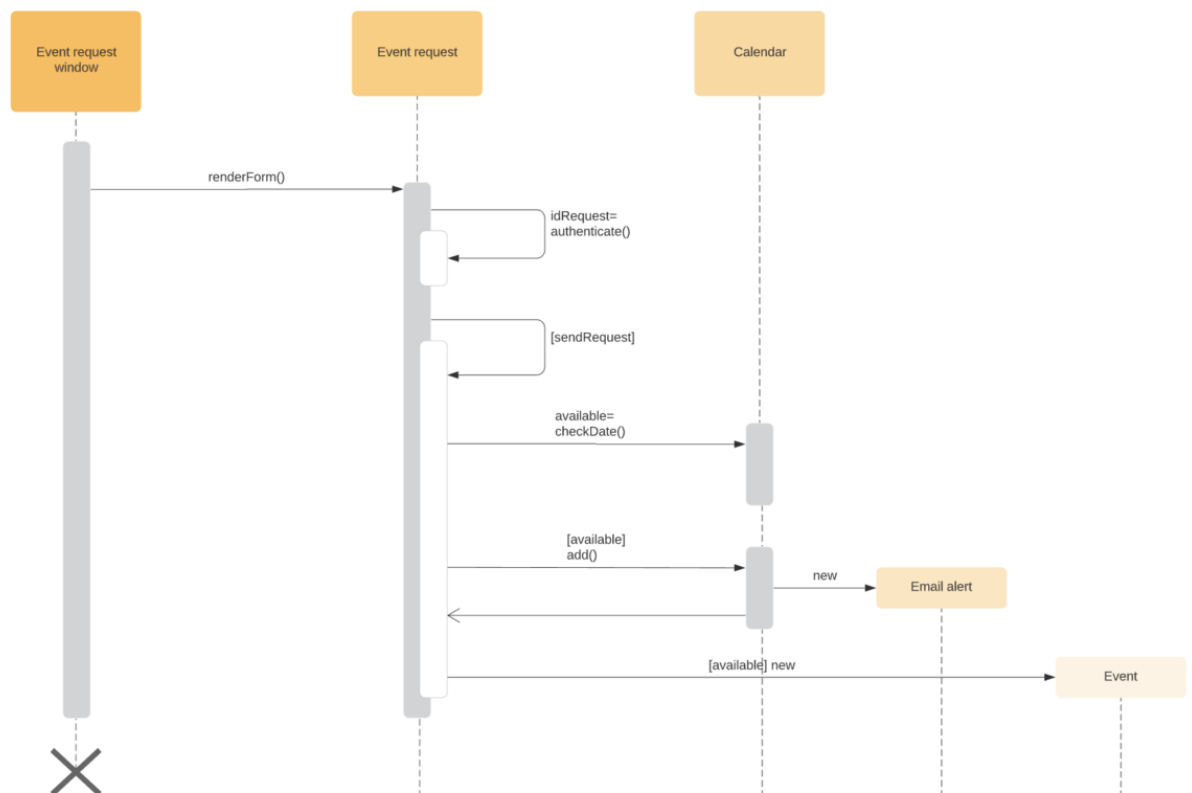


<https://www.lucidchart.com/pages/uml-state-machine-diagram>

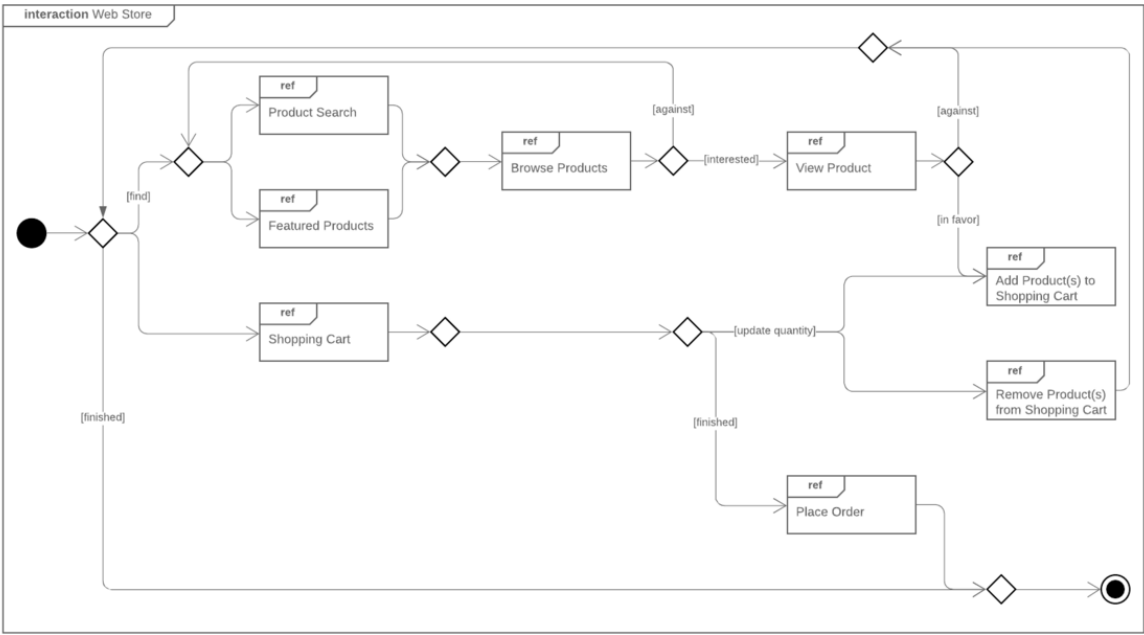


- interaction overview Diagram: gives an overview of the game mechanics from the point of view of interactions

#### communication diagram



#### sequence diagram



Interaction diagram

### 5.1.3. Create requirements analysis 8h

- as in lecture

Or see typical material such as:

<https://www.nuclino.com/articles/functional-requirements>

<https://www.jamasoftware.com/requirements-management-guide/writing-requirements/functional-requirements-examples-and-templates>

<https://www.visual-paradigm.com/guide/use-case/what-is-use-case-specification/>

The requirements analysis should be done at the beginning, but it is a dynamic document: You should formulate the requirements in such a way that it is clear how this is tested or how functions, for example, ensure that the requirements are met. Initially, you have no clear picture of the game and you might have not sufficient information how to realize all parts; hence you will have a first draft that reflects the state of knowledge.

The text document should be versioned in Git (e.g. GitLab or GitHub).

The final version of the requirements should then be available at the end.

Below there is a standard template for writing a requirements analysis document. Stick to it whenever it is appropriate. If some points are not appropriate, leave them out.

Sub-items are always numbered!

Structure from the lecture or as an example - ideally as a text file versioned in Git!

The first version goes in the report, the last in the appendix of the report.

#### Tips for requirements analysis.

If you define a variable like health points, then make clear that

- 1) If health points is a constant then use the typical notation of all caps HEALTH\_POINTS that makes from reading clear what is meant. Use it then as const type.
- 2) If health points is a variable, define its range and type: e.g. 0...MAX\_HEALTH\_POINTS; the reason for the range is that by this way you can quickly identify whether an error occurred and not unrealistic values appear. This is conservative programming and simplifies identifying potential memory leaks.
- 3) A good idea is to separate code and data, i.e. to read-in the health points from a configuration file during the constructor call – this allows to modify the parameters without going to the code which reduces potential later problems with erroneously adding some errors to the code.

### Structure of the requirements analysis

- 1. Introduction
  - 1.1. Purpose of the system
  - 1.2. Scope of the system
  - 1.3. Objectives and success criteria of the project
  - 1.4. Definitions, acronyms, and abbreviations
    - % here include all game object definitions: names and role in the game – so that you can use % them later in the text – this clarifies often the functional requirements
  - 1.5. References
  - 1.6. Overview
- 2. Proposed system
  - 2.1. Overview
  - 2.2. Functional requirements

% **Please number all sub-items**

% The big mistake at the beginning is that the requirements are specified far too vaguely, so the requirements **should be specified** as much as is possible at the current stage and they should be **verifiable**

% in short: requirements should be

% - complete – there should be no missing case

% - understandable

% - verifiable/testable: you code verifies the realization of the requirement

% - quantitative (instead of big and small: how big in mm, how small in mm)

%

% If a request is found to be unreachable, it will be marked as unreachable

% BEFORE PLANNING the TASKS in SCRUM MEETINGS, the requirement is specified in

% detail.

Template for a clearer presentation of the items:

ID	Title:
Description	
Acceptance Criterion	
Notes	

Tip: do not make too long items – better to subdivide them to small chunks so that you can process them within a day.

- 2.3. Nonfunctional requirements

% Please include the coding style here, as well as the performance as the frame rate that is to be achieved

% the requirements are written in such a way that they can also be checked - later this should also be checked explicitly in the results

- 2.3.1. User interface and human factors
- 2.3.2. Documentation
- 2.3.3. Hardware considerations
- 2.3.4. Performance characteristics
- 2.3.5. Error handling and extreme conditions

- 2.3.6. Quality issues
- 2.3.7. System modifications
- 2.3.8. Physical environment
- 2.3.9. Security issues
- 2.3.10. Resource issues
- 2.4. Pseudo requirements
- 2.5. System models
  - 2.5.1. Scenarios
  - 2.5.2. Use case model
  - 2.5.3. Object model
    - 2.5.3.1. Data dictionary
    - 2.5.3.2. Class diagrams
  - 2.5.4. Dynamic models
  - 2.5.5. User-interface -- navigational paths and screen mock-ups
- 3. Glossary

Examples:

Good practice examples for use case, functional requirements and development process:

[https://docs.google.com/document/d/1CayV6jYnNfRcY\\_39pDNZKR3tZzQeFna2ZCIA71M8ySc/edit?usp=sharing](https://docs.google.com/document/d/1CayV6jYnNfRcY_39pDNZKR3tZzQeFna2ZCIA71M8ySc/edit?usp=sharing)

Slides:

<https://docs.google.com/presentation/d/1YdQA-h8Klhxamligdc7y1Dpm9j17HAF5RPyYx4YbltM/edit?usp=sharing>

You might use tools such as

<https://use-case-maker.sourceforge.net/features.html>

that help to collect the information and display it in a structured way.

UML Designer is also a nice tool to generate UML diagrams or use instead draw.io



## 5.2. Develop and document architecture/class diagram: 8h

- think carefully about which architecture is chosen, check out beforehand what works in game engine (unity), try different designs and select one that is maximally flexible and modular
- Functional and non-functional requirements
- number everything so that you can refer to it later in the tasks
- write everything in such a way that you can later objectively verify that the tasks have the desired result (test cases)

When it comes to the architecture, it should also be considered to make it as modular and expandable as possible. You start with a minimal system that leads to a first playable system. This should be created as early as possible. After that, further components are specified in the requirements and then the architecture is expanded.

Often, instead of an architecture, a **class diagram** is provided – which is also ok. It defines the main interfaces so that the development later would be easier.

Tip: Start with a concrete, simple example and go in two directions:

- higher abstraction
- further components / classes (e.g. through inheritance from the parent class)
  - component-based architectures are more flexible
- test beforehand what works in game engine (unity)
- Architecture must be described: see lecture
- UML diagrams: different views
  - classes - structure
  - activities
  - Behavior: State Diagrams
  - interactions

Represent architecture – or class diagram:

tip:

[https://www.st.cs.uni-saarland.de/edu/se/2015/files/slides/08\\_Software%20Architecture.pdf](https://www.st.cs.uni-saarland.de/edu/se/2015/files/slides/08_Software%20Architecture.pdf)

The architecture should be such that it can be developed incrementally! KEEP IT HIGHLY MODULAR! You never know whether you might have to modify it (in most cases this happens!)

The architecture is versioned and stored in Git.

**You develop the game on the basis of a compiled language**, not with the scripts, since these are not that powerful and you often lose the overview with larger projects.

Comment:

Simple games can also use a UML class diagram instead of an architecture, more complex games benefit enormously from the arrangement of the classes in modules, which then represent the respective architectural elements between which communication takes place.

If you create a class diagram:

Always consider whether you can use an **abstract class** as an interface. Use **components** and **inheritance hierarchies**, use **generic programming**<sup>4</sup> (C++: templates) if it makes sense. And always think about sensible and suitable data structures that are simple/elegant on the one hand and efficient on the other.

Take also into consideration to use **design patterns**<sup>5</sup> such as

- manager classes (singletons)
- controllers
- factory patterns
- decorator classes etc.

### **GameObject Dictionary**

For larger games, it is often difficult to get the complete information of how the different entities interact over the functional requirements.

So typically, a dictionary that presents the different game objects, acronyms/names and their relationship is very helpful since these terms can later be used to describe what is going on.

There are two locations where to do this:

First you use the **definitions 1.4.** to define all names of objects and their relationships (named relationships) such as shopOwner as game object name and vending as relationship between shopOwner and goodToBeSold

Then you define the properties of each game object in a **data dictionary in 2.5.3.1.**

---

<sup>4</sup> C# <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics> C++: templates

<sup>5</sup> <https://refactoring.guru/design-patterns>

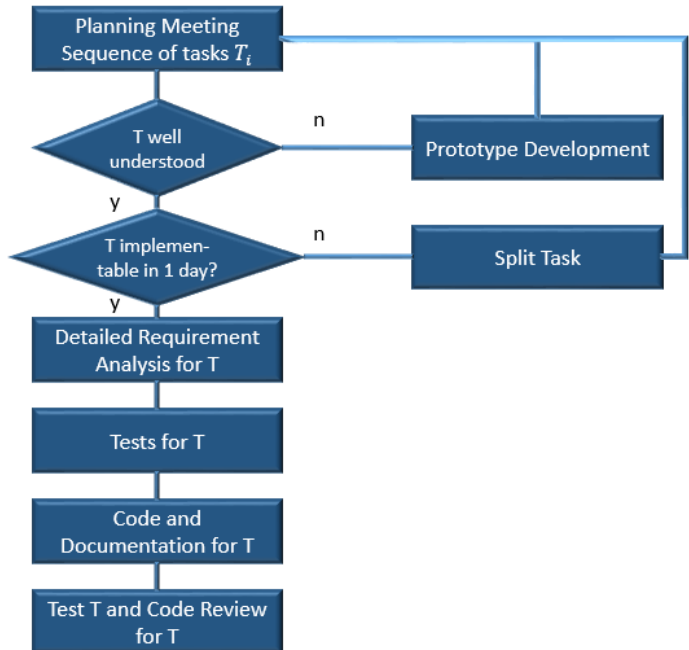
### 5.3. SCRUM:

#### 5.3.1. Development process

At the beginning of this chapter the given development process. You **must follow this process exactly** at the beginning, and then expand and improve (not shorten!) it accordingly.

Figure: Defined process (obligatory):

1. Sequence of tasks according to necessity (abstract before concrete class) or relevance
2. Task T ok means that you completely understand how to implement it, how to correctly use the game engine interfaces and classes necessary and how to test it. If any of these conditions is not met: develop a prototype for each open question (and define this prototype development as a new task)
3. If the task takes more than a day to realize, split it into sub-tasks: Generally: The implementation strategy for a class is the following
  - a. Define first the interfaces, i.e.
    - i. the class header and parent class(es) including the generic model (e.g. template)
    - ii. the class attributes
    - iii. the interface of the class methods (public, private, protected), i.e. just the method head without any content; and if this is a return-type method, return 0.
  - b. This interface should be written before starting writing the test since this defines how the test accesses the methods in the class.
4. Writing tests: Use tests whenever possible (it is an essential element of the CI/CD pipeline). The game engines often provide strategies for Test Driven Development<sup>6</sup>.
  - a. Derive tests from the requirements, which means
    - i. trivial methods/functions: review process
      1. e.g. only a few lines of code
      2. no loops or branches
      3. no sub-functions/complex method calls
    - ii. test more complex functions using **unit tests!** (you can always test your own part by a unit test using standard unit test frameworks)
      1. assertions for the correctness of the ranges of the input variables are useful
      2. develop unit tests for each complex method/function
      3. you might have to program such that the code is testable, be aware of that!



<sup>6</sup> <https://blog.unity.com/technology/testing-test-driven-development-with-the-unity-test-runner>

5. Writing the code requires that you adhere to clean code development<sup>7</sup>.
  - a. Document while you write: Doxygen or another documentation tool such as Sphinx
  - b. Typically, the code for a method should not be more than 10 lines (which easily fits on a screen): if it is longer, group the code into private methods that are called.
  - c. When methods from the game engine have a lot of arguments and only a few are relevant, there is often a good idea to have a decorator or interface method with only the few parameters that are necessary and hide the remaining complexity.
  - d. It is often also a good idea to refer a structure element to a named variable with is easier to understand – the code gets shorter and easier to understand.
  - e. Define lint or coding style (for all languages there is a lint that can be used to check the coding style: e.g. cpplint or pylint).
  - f. Use static code analysis tools to check the quality of the code. A good tool is SonarQube (<https://www.sonarsource.com/>). It allows for
    - i. Checking the code quality
    - ii. Identifies weak code that is prone to errors
    - iii. Identifies code complexityIf the code does not pass the static code analysis, define a new task for the to-do-list to correct the code accordingly and use the issues from the static code analysis as element of your process defects list.
6. Coding should be directly followed by documentation using a standard documentation tool such as Doxygen<sup>8</sup> or Sphinx<sup>9</sup>.
7. Then the real tests are executed: If they do not pass, make a new task with correcting the code. Try not to use debugging if possible.

### 5.3.2. Gradual realization of this process:

Experience showed that directly starting with the full-fledged agile process is a hurdle. Hence, the following strategy is planned for this internship/practical:

1. Week 1+2: only implement prototypes (i.e. no productive code)
  - a. This allows to get acquainted with the game engine for the most simple tasks.
  - b. It allows to learn how to realize unit tests for the prototype code (which is for many not really clear how this is realized)
  - c. It allows to correctly use trello (e.g. that the backlog stack should be completely filled with tasks as far as they are already known) and to use an extra stack for all documents (such as use-case text, UML diagrams, requirements)
  - d. It also allows to correctly use the Git version control system in the first two weeks – hence the amount of programming is limited and the quality of the prototypes is secondary.
  - e. In the 2-weekly meeting it is checked by how far you are able to correctly use trello.

---

<sup>7</sup> <https://www.codingdojo.com/blog/clean-code-techniques>

<sup>8</sup> <https://www.doxygen.nl/>

<sup>9</sup> <https://www.sphinx-doc.org/en/master/>

2. Week 3+4: develop first productive code
  - a. Use the test-driven approach – and learn how to write code so that it is testable by unit tests
  - b. Estimate used time
  - c. Learn to use static code analysis such as SonarQube
  - d. In the 2-weekly meeting the code quality is checked (clean code, code complexity) and what results you got using static code analysis
3. Week 5+6: Process understanding
  - a. In this period you learn how to correctly list defects in the process
  - b. You also learn how to efficiently perform tests
  - c. In the 2-weekly meeting the defect list is discussed and the means how to overcome them, also the tests are checked to see by how far your code is covered by tests.
4. Week 7+8: Full SCRUM cycle executed
  - a. In this period you assemble all parts for your complete process and the first updates of the process
  - b. In the 2-weekly meeting the process updates are checked.
5. Week 9-10: Continuous improvement
  - a. In this week you observe how your process improves and report this in the 2-weekly meeting
6. Week 11-12: Depending on how much time is left, there are occasional checks or suggestions, also accounts for potential delays.

If you perform an advanced practical, you will have the following elements as well:

7. Week 13-20: Depending on how much time is left, there are occasional checks or suggestions.

### 5.3.3. Contact meetings every two weeks key aspects

We meet every second week for about 30 minutes.

In the sprint meetings, I will gradually explore the following points

1. Setup and structure of the Trello board; **Time estimates** – book keeping list
2. Verify requirements analysis: **Requirements refinement** – suggestions how to improve it: and from this time on you SHOULD do it accordingly
  - a. complete (all elements and their behavior defined)
  - b. does not contain any unclear parts, e.g. fast -> how fast, state number
  - c. it is formulated in such a way that it is clear how the tests are to be written
2. Tests (details later): are the tests, given the requirements, complete, only very simple tests were left out, which were then dealt with in the code reviews (please document).  
**Tests** – please prepare an example case where you can demonstrate that you test a given requirement for a given function/method (take care that the functions/methods have arguments when called since otherwise this is hard or impossible to test) and take care that in case of private methods you call the tests from a helper function within the class! – it is expected that you write tests and a sufficient detailed test plan to convince that most relevant cases are tested
3. Code: is the code (clean code): look for details that characterizes clean code  
**Code** – please prepare a typical example of a complex function/method and class you implemented: demonstrate by this that you follow the clean code concept and that the code is as well structured as possible and that you found out how to write it with minimal complexity – from this time on, you SHOULD stick to these suggestions
  - a. well documented and structured in such a way that it is easy to read and understand,
  - b. Typical problems such as catching possible errors (division by 0, uninitialized variables, etc.) have been avoided - if not, there is a comment explaining why an error cannot occur,
  - c. it is guaranteed by means of assertions that the preconditions (i.e. e.g. variables are in the correct range - are the valid ranges specified in the documentation and
  - d. was it programmed conservatively, e.g. was unsigned xxx used for positive variables?
4. **Process improvement**: (details later) you see next the given process: on this level you update your process, you do not make a list or description what you do but how this process is modified independent of the programming task you do – you should update this process over time and should at any time to show and demonstrate what are the updates. Due to my experience it is good to have a list of entries:
  - a. Problem occurred
  - b. Reason in the process
  - c. Update of the process
  - d. Evaluation if this modified process still generates the problem or notThis is important to keep updated since this is an essential element in the later report.
5. **Code review** process: how was this approached, in particular (details later)
  - a. what questions were asked, such as
    - i. good structure of the code (see point before)
    - ii. you can follow the process so well that you can clearly see whether the requirement you are looking for is met
    - iii. any code conventions you give yourself are met
    - iv. Process changes: have you made any process changes?

- b. Make a check list for the code review and look what are good code review practice guidelines

#### 5.3.4. Contact meetings agenda points

Preparation: about 2 hrs

Duration: 30 minutes

For each SCRUM meeting I have a focus topic such as (assuming 1 full day per week where you work on the internship)

1. Time planning/Trello
2. Code (clean code principles, typical flaws, complexity)
3. Requirements analysis – three criteria: verifiability, completeness and quantitiveness
4. Tests (completeness, assertions, testability)
5. Process improvement
6. Architecture revisited

----- + another 4 for advanced practical typically the SCRUM meetings finish

7. Code review
8. SCRUM planning concept
9. Prototyping revisited
10. Lessons learned

At any time you might ask me for advice for any item. My concept is that in the two-weekly contact meetings, I give you a feedback about how to improve a certain part of the process and its implementation, and from that time on you update previous parts accordingly and stick to that in all future steps. If there is still a need for feedback, you might simply ask me in the sessions for further feedback.

### 5.3.5. Structure of YOUR planning meeting:

Preparation: about 1/2 day

Duration: <1 hour

- discussion of current status of work
  - what has been reached, what has been shifted
  - validation of process changes
- Which problems occurred?
  - Make a written list of problems you detected, describe them in short sentences
  - Why did they occur?
    - Note the reason in short since you need it in the report and presentation
  - Which process step was not optimal?
  - How to improve the process?
    - Document that since you need that for your report and presentation
- Which is the next task T to be processed?
  - According to functionality and relevance
  - Consider the logical order from dependencies i.e. abstract class before concrete class
  - If the task is too large for being resolved within a day, you perform a so-called backlog refinement, i.e. split the task in smaller tasks and put the smaller tasks on your backlog pile, each obtaining an own card; here are tips for doing this:  
<https://www.atlassian.com/agile/scrum/backlog-refinement#:~:text=Backlog%20refinement%20is%20the%20process,wants%20and%20the%20business%20needs>.  
Best is to use the functional requirement item number and add a .1, .2, .3 as suffix to it so that it is clear that this handles the corresponding functional requirement – which is then also refined accordingly in the requirements analysis document.
  - If you feel that you cannot process the task directly, you form questions and for each question you define a Prototype. Make clear what is a prototype e.g. by adding a suffix .p1, .p2 etc. so that everybody sees that you are working with prototypes  
Here is a tip for generating prototype code: (in software engineering, this is typically known as operational prototype)  
<https://ieeexplore.ieee.org/document/156899>

It is IMPORTANT not to forget the discussion of your planning process WHENEVER you experienced problems or errors in the code. Each of these problems is considered as a process problem and you find out what might have been the reason for this problem and reason about strategies to fix it. Always check whether the process modification was successful.



### 5.3.6. Table for Sprint Meetings

Date, Sprint Meeting Number	What was done: Task Number	Problems <ul style="list-style-type: none"><li>• which</li><li>• why</li><li>• suggestion for solution</li></ul>	Next Tasks <ul style="list-style-type: none"><li>• sorted List according to relevance</li></ul>

Please formally write down your process at the beginning and then change it each time, commenting why you are making a change and what caused it.

A graphical representation is very useful.

As a general rule, you should always have improvements in your process within the first 4-6 weeks. Any problem like "I don't know what to write as a requirement" (then the task needs to be scaled down because obviously the task was too complex - or it's not so clear how EXACTLY to implement the code), "it's unclear how the test is written" (then the requirement is either incomplete or not clearly written, because it must be written in a way that makes it clear how to test), "I don't know how to implement this" (then have to you try out how you do it with a prototype beforehand), "I had bugs in the code" (then the cause was a bug in your process, e.g. you could have done a code review to avoid typos or unnecessarily complex code or it is a consequential error, then you have not tested extensively and there is a lack of control as to whether everything is complete) is therefore specifically related to the process and any problem that occurs should be accompanied by a process change on how this can be remedied.

### 5.3.7. Process change table

Ideally, you write a log list in your Trello:

Task Number	Error type	Reason for Error	Process Change	Validation of changed process if error still occurs

Since many find it difficult to understand how a process should be changed to make it better, here are a few examples:

#### Example 1:

Errors: Similar errors always occur during programming, e.g. uninitialized variables.

Process change:

Specifically, the code review process you include should include further investigation and see if this type of error was avoided here.

The code review process is expanded to include a check list containing all items to be checked.

#### Example 2:

Error: The process was changed, but the errors reappeared.

Process change:

In the case of process changes, this process is described in a table (see above) and specifically examined at each planning meeting to determine whether the change was successful.

#### Example 3:

Bug: The game engine was misused due to not fully understanding how to use some elements.

Process change:

Prototyping is done using examples to understand how the game engine is used. But this does not cover the case that e.g. in the given context the game engine behaves differently than in the example.

Therefore, a branch is created and then a prototype is written, how the implementation is imagined. If this implementation works, the requirements are fully defined, the tests are written and then the code including code review.

If it doesn't work, the prototype is further investigated and experimentally modified until it works.

#### Example 4:

Error: The prototype was too big and took more than a day:

Process Change: Make a list of how long it will take you to implement a small prototype. To do this, estimate the complexity, e.g. based on the required elements of the game engine or the number of functional elements such as transfer variables, etc. and write the time you needed to implement the prototype.

They can also introduce elements like integration tests, tables to document all test cases, establish coding guidelines, introduce structures for their planning meetings, etc.

It is very helpful if you also use Trello for all collected information and tables.

#### Example 5:

Error: The role of game objects was not understood until a later stage of the implementation.

Process Change: Realize a spiral-model-process, i.e. define first a small subset of each requirement and implement it. Evaluate whether it fulfills the criteria and uses the game engine elements as expected. If

not, consider the existing code as a prototype for a new round. Use refactoring strategies to turn the existing code to a better code model.

---

Counter Example 1:

Defining elements of the process that are already defined are not a process improvement. Example: you include prototyping for clarifying the requirements or to split up large tasks into smaller tasks.

Counter Example 2:

You use steps that are not part of a process. Example: you discuss what has been programmed by presenting the code to others.

Counter Example 3:

You describe details of existing process elements such as: you discuss how to split up a task into smaller tasks: this is already a part of the defined process flow.

Counter Example 4:

You collect a list of defects in the implementation and for each new code, you go through this list and check whether all aspects have been considered. This is not useful since it does not scale.

## 5.4. Time estimate

You should learn to estimate how much time you need at the beginning.

It is best to keep a diary in which you enter the following data for each task:

Task - develop code:

Supplementing the list of requirements (at the time you know exactly which functionalities of the Unity Framework, for example, you want to use and you already have the code in mind - now you should also be able to write down exactly what the method does)

Formulate and implement the tests that should ultimately test all requirements. If you have trouble here, you have not written the requirements completely and clearly.

Writing the code, always considering whether you are actually only implementing the requirements that you have written.

Testing: when a test goes wrong, your best bet is to note what goes wrong and why - and finally investigate what point in your process led to it: that's the point in your SCRUM meeting.

So each journal entry has the following line with minutes taken for each part:

Time estimation table

Date and Task Number	Complexity of Tasks 1-10 (estimated)	Formulate requirements	Tests writing	Code writing	Test execution and error reporting	Comment what went wrong

Over time you will learn how much time what costs and how this has to do with the complexity of the task. If you want, you can estimate that using linear regression.

An recent strategy of estimating the effort for a given task is planning poker:

<https://t2informatik.de/en/smartpedia/planning-poker/?noredirect=en-US>

This is a strategy that typically works for experienced groups but if you feel that your estimates are ok, you can try this out.

## 5.5. Test-Driven Design - a must!!!!

This part is written since often there are difficulties to test the own code.

A good overview how you can manage that with Git: <https://docs.gitlab.com/>

### 5.5.1. Recommendations

For information on how to develop tests:

<https://www.informatik-aktuell.de/entwicklung/instrumente/was-ist-ein-guter-unit-test-und-wie-develops-man-ihn.html>

See also:

<https://www.browserstack.com/guide/learn-software-application-testing>

<https://techbeacon.com/app-dev-testing/5-key-software-testing-steps-every-engineer-should-perform>

<https://www.guru99.com/software-testing.html>

<https://www.testim.io/blog/software-testing-basics/>

Q1: When to write code and when to write a prototype:

- If it is **clear** what exactly is being developed and how something is needed, then **test-driven** design is used.
- If it is **not entirely clear** what is to be done, how something is to be realized or whether something is going to work the way it is imagined, a **prototype** is developed first. If the prototype works, it can be used as a further test, if it works without errors - if not, then it only serves as a construction site from which you can use for the final development (only concepts, no copy-paste of the code, because that otherwise leads to careless mistakes and ugly code).

Q2: Test or not to test:

For each point that is implemented, the **requirement defines what is to be tested or verified**

- then unit tests are written that show that the requirement is met
  - Interface tests: correct ranges, correct values in output given inputs: Interfaces are defined in architecture and both sides must be tested
  - Assertions: Intercepting errors with incorrect inputs or incorrect outputs and bounds checks / is switched off again in productive operation
  - Try-catch for file handling problems or over/underflow
- In case of simple methods verification is a better choice
  - If person A writes the code, person B verifies its correctness by comparing the requirements with the implementation
    - If a requirement is very specific, e.g. provides a formula, this formula should be seen in the code as well
    - If a method is called, this can also be verified by just checking that the method is called correctly i.e. with the correct parameter values (check this!!)

### Q3: Unit tests

- comment: it is a bad idea to write methods without arguments – they are very difficult to test and the methods depend on class-relative global variables (with all the corresponding side-effects)
  - comment: if you use private methods, find strategies to be able to test these methods e.g. by using factory patterns
  - comment: not to have too many tests, keep the methods short (in terms of lines of code)
    - use sub-methods instead to simplify the code
    - use templates (in C++) resp. generic programming to keep the method short
    - spare if-conditions and loops - especially if they are nested, this makes testing expensive
  - never allow direct access to attributes: you have get() and set() methods for that purpose which allow to include checking the access
- Use test tools to find memory leaks (depends on the programming language)
  - Use assertions to catch cases on the fly, no printouts, and only debug when there's no other option.
  - write tests first, then code
  - a task should last one day
  - if the task turns out to be too complex to implement, identify reasons and then return to the task list: reconsider
    - Often there are too many dependencies
    - Too much is often put into the task
    - When programming, you often notice that some things were not taken into account: only then do you understand how it works (prototype) and back to the task list
  - Set up code review boards, i.e. when code is written, have another review meeting in pairs
  - Early deployment - develop to always have a running game, i.e. incremental functionality!

There was always the question of how to test. So here's a quick note:

The requirements in the requirements document must be formulated in such a way that it is clear how they can be verified - if this is not possible, the requirements are formulated incorrectly and cannot be verified and are therefore not a requirement in the classic sense.

The verification of the requirements can be done in two ways:

- Is the functionality available?
- Does the requirement meet your criteria: this can be tested automatically by writing a test routine.

A requirement exists if it is correctly implemented: In the case of methods in particular, one formulates:

A test is carried out to determine whether the input variables are set correctly.

For a set of pairs of input variables - correct output takes place, so that statistically the typical application scenarios are covered: i.e. typical inputs, wrong inputs (should be caught), inputs on the edge (e.g. positive integer expected, test for -1, 0, +1)

These are the methods of black-box testing, which are the bare minimum for a reasonable test.

<https://www.guru99.com/test-driven-development.html>

The wiki also offers a good overview: [https://de.wikipedia.org/wiki/Testtriebe\\_Entwicklung](https://de.wikipedia.org/wiki/Testtriebe_Entwicklung)

or this lecture: <https://www.cs.colorado.edu/~kena/classes/5828/s09/lectures/21-tdd.pdf>

Each test is documented, i.e. it is described exactly how what is tested. If test stimuli are used (input-output pairs), then justification is given for how these are chosen. The documentation of a test is part of the program documentation.

[https://de.wikipedia.org/wiki/Software\\_Test\\_Documentation](https://de.wikipedia.org/wiki/Software_Test_Documentation)

Use static code analysis whenever possible. It is recommended to perform a static code analysis

Here is a link to tools:

[https://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)



### 5.5.2. Code review

It makes sense to carry out a code review, this allows major errors to be eliminated before the test. You can then document this either in Trello or, even better, in Git, then you always know the current status of the project.

Good approach for code review:

- Is the code easy to read and well documented: if not: comment
- Have the requirements been formulated under which the code should be correct: in which range are the input variables, is this also checked?
- Are the variables always maximally constrained that are passed to let the compiler do the static test
- If the names of all variables, etc. are good, the code is well formatted, brackets are set sensibly, coding guidelines may have been correctly implemented
- Is only implemented what is also tested or what is trivial to check.
- Can you walk through the code and clearly demonstrate that it's doing the right thing? If not, then tests must be written

Here is a nice link that gives you valuable help in how to perform efficient code review:

<https://www.atlassian.com/blog/add-ons/code-review-best-practices>

Using the Kanban Board for Agile Development: Columns (can be named slightly differently, but should contain this information)

1. backlog
2. SPRINT: Tasks for current sprint: each task should include one day:
3. please plan, write down how long it really lasted
4. for the subsequent planning, the complexity of the tasks is reduced, tasks may be split
5. ACTIVE: In progress: for each max. 1-2 tasks
6. DONE: Completed
7. (possibly) improvements

Swimming lane in Kanban board: each has its own row

A new sprint every 5 working days

1. consider what went well and what didn't go so well - retrospective view
2. new task list for the next sprint: Adjust task complexity based on experience
3. Test-driven design: were the tests sufficient, if not, define how the tests are made better
  - a. possibly schedule code reviews if task is complex: extra day for two team members
4. Cross check with requirements
  - a. missing requirements
  - b. if the requirements are not precise: then do not develop any code, but first adapt the requirements
  - c. tasks are too complex because there is a lack of experience: first develop the prototype, then the task

Tip: An essential point at the beginning will be that you learn to correctly estimate the time you need for each task.

## 6. Report: 10-20 pages ~36h

### 6.1. Scope

Summarize all documents such as storyboard, specification (at the beginning and end), architecture and documented result in one report; short presentation at the end.

Issues:

- must write clearly in writing
- what agile development should look like
- how test driven design should be
  - o how to do tests
- how game description or use case should be
- how the requirements analysis should be and how precise that the tests that describe the fulfillment of the requirements are derived from it
- how architectures are described and which essential architectures exist
- how to estimate and then improve the working time per task
- how to do a retrospective analysis every 5 days
- The report
  - o must be structured more clearly - specify structure
  - o the discussion should be like this
  - o what was specified
  - o How did we implement this as close to the specifications as possible
  - o why did we deviate
  - o did it make sense to deviate?
  - o what went well, what didn't
  - o if not, why not
  - o how can this be done better

[https://softwareresearch.net/fileadmin/src/docs/teaching/WS05/SE1/04Folien\\_Kapitel6\\_Teil2.pdf](https://softwareresearch.net/fileadmin/src/docs/teaching/WS05/SE1/04Folien_Kapitel6_Teil2.pdf)

## 6.2. Contents

### 6.2.1. Chapter 1 - Introduction & Basics:

Goal of the internship:

“The goal of the internship is to learn agile development processes similar to SCRUM and to further optimize the own process in order to achieve a 0-error level.”

Most important frameworks and classes and terms of Unreal Engine or Unity (these are the most often used game engines).

### 6.2.2. Chapter 2- Use case, requirements analysis and architecture (each sub-chapter)

Brief description of what the end product should look like. So basically a short version of the GDD (Game Development Document), starting with the storyboard, the requirements derived from it and the architecture with which the requirements are to be realized. Include please all the UML diagrams – which should have the correct format!

The respective documents (at the beginning and at the end should follow as an attachment).

### 6.2.3. Chapter 3- Implementation

How were the requirements technically implemented?

Brief description of the components.

Then essential results/scenes as a screenshot.

### 6.2.4. Chapter 4- Development Process (should cover about half of the report)

Describe and analyze the development process and **present it graphically as a process**, namely how it was at the beginning and how it developed - and what was the cause and the change in the process).

What components did the agile development process have and what is it based on.

The best thing to do is to first write the given process that you implemented first and then your changes. Use a graphical representation of the process as you should implement it (containing Test Driven Development (TDD) and Prototyping). And then you depict on this graphics how you modified the process.

Keep in mind that with the changes there was always a reason why the process was improved and you made a change on the basis and examined whether it also achieved the goal.

- What is the role of each component?
- How were the components implemented in each case?
- for each component: what worked, what didn't and if not, why not
- If you had to play the game again, how would you do it now after that experience?

- What was good/bad? How good was our effort estimation / time planning in hindsight? What have we learned (Lessons Learned)? Perspective on what can be done better.

#### 6.2.5. Main focus: These two components MUST be used to successfully accomplish the internship

##### Test-Driven Development

- Agile development strategy
- How good were the use cases for characterizing the requirements
- How good were the requirements for the later implementation
- How appropriate was the architecture, what went well and what didn't

##### Prototyping

- Use prototypes whenever there are open questions of
  - How to realize the requirements – make a prototype that tells you what can be realized and what components are necessary as requirement
  - How to realize the code: unknown functions from the game engine, unclear API and specification, unclear how to test
- Each open question is an own prototype (keep it short) and register it as a task
- Each prototype should be as simple as possible just as big as necessary to answer a task
  - You should be able to implement the prototype within a few hours
- To understand interfaces/API, start with an example provided by the game engine documentation and modify that slightly until you end up with the code that solves your problem
- You can use the scripting abilities for prototyping since this guarantees a shorter cycle time

##### What was the development process like

- Test-driven development: What went well, where were problems
- Agile development with estimation of effort, management, improvement of the process

##### Development Environment:

It makes sense to use the same development environment for each party of the team.

Use code documentation tools (Doxygen, Sphinx) - available for all languages, saves a lot of work, should always be used and for every task (last step if task is otherwise completed): assign meaningful names, document not what, but why (clean code principles).

##### Use good programming style

- Maximum restriction for the variables: don't use an `int` but use a range if the values should be in a range - errors will then be recognized automatically
- Always use try - catch when loading or saving things - because there's always a chance that something isn't there.

- Division by zero, overflow, underflow: this should always be tested at least in the development stage - see also `try - catch`
- Quality before quantity:
- rather less features than poorly implemented or documented/planned features
- always make it clear what you are currently working on in the specification/requirement with the task
- Document what didn't go so well in the process and how it could be done better

It is also instructive how you changed the development process in the course of the project: what was the reason, what was changed and how did that affect it?

Outline the process flow with a sketch and make it clear where and why you have made changes compared to the original version.

#### 6.2.6. Attachment/Appendix **DO NOT FORGET!!**

1. First and final requirements analysis - latest Git release – this is important to see whether all the requirements have been refined as required
2. First and final class diagram
3. Code you developed (source of the final version) and the test (including test plan) – this is important to evaluate the clean code quality and the completeness of the test.
4. Doxygen document synthesized from the documented code – this is important to evaluate the class diagram and the documentation
5. Possibly further documents, if useful

## 7. Presentation ~12h

The presentation is based on the structure of the report. It should last about 30 minutes (about 15-20 slides), then the game should be presented and at the end there are questions about the lecture and the game.

Orientation of the structure according to the report

We do a trial presentation about 1 week before the presentation date

Please send me the following information for each participant by this time at the latest:

Surname, first name, matriculation number

## 8. Tips and experiences

Start with a minimum viable product

Carry out usability tests with friends and revise the design based on the feedback

Agree on a programming style

Use same versions, use CI in GitLab/GitHub

Consider tests sensibly:

And consider which tests are necessary, which are minimal, automate

Use a good tool for communication

- WhatsApp
- discord
- note

Take care of clean code – use static code analysis for learning how to make code better. There are also tools that help in formatting, code writing (e.g. code completion with Copilot from Microsoft) etc. You might also try out large language models for getting inspiration how to write a specific code – but should take this as inspiration for your code and not only for just programming with these tools since they often do not produce accurate, correct and well structured code.

See for example:

<https://docs.github.com/en/copilot/using-github-copilot/getting-started-with-github-copilot?tool=vimneovim>

[https://www.reddit.com/r/Unity3D/comments/12c4ppb/has\\_anyone\\_tried\\_github\\_copilot/](https://www.reddit.com/r/Unity3D/comments/12c4ppb/has_anyone_tried_github_copilot/)

<https://kchodorow.com/2022/01/21/a-review-of-github-copilot/>

<https://www.jonathanyu.xyz/2023/11/13/ai-powered-unity-game-development-with-github-copilot-6-month-review/>