AMO - Assembly, Maintenance, Observation

Requirements Engineering, Sprint and Backlog Documentation

Max Weise

March 9, 2023

Contents

1 Requirements and Architecture

1.1 Introduction

1.1.1 Structure of the Document

This chapter is a collection of the thoughts that went into the design and development of version 1 of the system. The document also functions as a documentation of the smallest set of functionalities and requirements that make up the developed application. Chapter 2 will contain a documentation of which requirements were implemented by which ticket and with which version. It will also server as a backlog and reference over the tickets that should be done in a given development cycle. More on the specific development workflow in said chapter.

The following section will give an overview over the context (subsection 1.1.2) and goals (subsection 1.1.3) of the project. Section 1.2 will list the requirements of the developed application and will give an overview over the

1.1.2 Context and Motivation

The fencing department of the local sports club provides equipment for kids and adults. This allows for newcomers to enjoy the sport without buying the expensive equipment without knowing how they will like the sport and whether or not they want to keep fencing. This equipment is not maintained by any special party and will sometimes go months without proper testing and maintenance. As members change frequently and the armourer is not a fixed position in the ranks of the club there is no comprehensive list or overview over the equipment and its working status. This is especially problematic when new fencers need to borrow equipment from the club to compete in their first tournaments.

To solve the lacking-overview-problem, a system will be developed to assign unique identifiers to the weapons and keeping track of their condition. This task will be supported by a database holding the needed information.

1.1.3 Goals of the project

The overarching goal is to develop a system that allows for keeping track over items present in a given inventory. In this context these items are epees for fencing or equipment to setup the fencing piste. The goal is to be able to know who borrowed the weapon and the working status of each item. The system should also facilitate the job of the armourer or the person currently in charge of fixing the item, as the assessment of each weapon will simply be done already.

1.1.4 Solution to the problem

The solution for the problem presented in the previous section consists of two parts. At first, all epees in possession of the fencing department need to be collected and registerd. For that, each weapon is given a uinique identifier, which not only allows to identify the weapon and make it storable in a computer based system, it allso will provide information about the weapon itself. The ID will have three components: #bs-h-id, where

- *bs* notes the size (length) of the blade. Technically, this can be any integer between 0 and 5, but for this project, only sizes 0 and 5 are needed.
- *h* notes the handedness of the weapon, either left or right.
- *id* is an ongoing number. It will increase for every new object registered in the database.

The index for every weapon needs to be created manually.

Secondly, the weapon is registered in a database, containing information about the status of the weapon, who currently owns it and what maintenance has been done when and by whom. This database application is the subject of this document.

1.2 Requirements Engineering

1.2.1 Requirements

These requirements provide a foundation and minimal set of basic functionality of the application. Implicitly, this section also servers as a documentation for the version v1.0.0 of the project and requirements for further versions of the project will contain their own documentation.

The section will be devided into *Functional Requirements* (Table 1.1) and *Nonfunctional Requirements* (Table 1.2). Each requirement will have an identifier, a description and a priority. *Must Have Requirements* (MH) are non-negotiable and need to be implemented first, as they ensure the correct and planned behaviour and functionality of the application. *Should*

Have Requirements (SHR) can be postproned in favour of a MHR, but should be considered when implementing other requirements and features. *Nice To Have Requirements* NtHR are negotiable and can be implemented at later point in time.

Functional Requirements

ID	**Description**	**Priority**
FR01	The application shall provide a database scheme to store weapon objects	MHR
FR02	The application shall implement the CRUD operations for every database scheme	MHR
FR03	The application shall store the following information: ID of the weapon, current owner of the weapon, a list of maintenance done on the weapon	MHR
FR04	The application shall provide a mechanism to filter the list of results	SHR
FR05	The maintenance entry shall include the date of maintenance, the maintenance work done and the maintainer	MHR
FR06	The application shall provide a graphical user interface (GUI) to interact with the database	MHR

Table 1.1: Functional Requirements

Nonfunctional Requirements

ID	**Description**	**Priority**
NFR01	The application shall be constructed in such a way that new types of material can easily added in the database	MHR
NFR02	Backend and frontend should not be coupled and be interchangable from oneanother	MHR

Table 1.2: Nonfunctional Requirements

1.2.2 Architectural Design

The following section defines the architecural design. The section the general idea behind the structure of the project and an overview of the designchoises.

Designoverview

Figure 1.1 shows the design of the application. It follows the layer pattern, which is used to decouple frontend from backend. There are three layers: The presentation layer, the API Layer and the Data Access Layer.

The presentation layer provides a GUI for the user to interact with the system and to display the data provided by the API. The API layer defines endpoints which can be used by the frontend to request data from the database. It servers as a connector which connects prensentationlayer and database. It uses components from the data access layer to forward the data requests from the presentation layer. The data access layer defines the operations which can be done on the database.

The driving factor behind every descition made in the development process is the flexibility to adapt to future changes. Because the system is not yet tested and requirements were not gathered through user-centric processes, unforseen requirements can arise in the future and during usage of the system. For this reason, the system was designed to be modular and flexible for change. By using the layer architecture, the system provides a modular way to change database and frontend technologies independant from oneanother.

Database services should not be accessed driectly, as it should be possible to switch between technologies as needed. For this reason, the data access layer defines data base adapters, which implement the specific querries for each database. The API should only know of the classic CRUD operations (Create, Read, Update and Delete) and delegate, which to use when and with which parameters.

1.3 Languages and Frameworks

The application will be written using the Python programming language. This is done for two reasons. First, performance is not (yet) a requirement to be considered, and secondly it allows for rapid prototyping. As the first version of the amo system serves as a proof of concept, the tradeoff „less performance for faster development“ can be safely made. If at some point in the future, performance will get an issue and a faster programming language will do the trick, then a switch can be made.

The framework of choise is Flask. Flask is a minimal backend framework for the Python language. It is used to provide the API and handle requests / responses, routing, user authentication etc. Because of it simplicity it is flexible and allows the developer to implement custom soulutions.

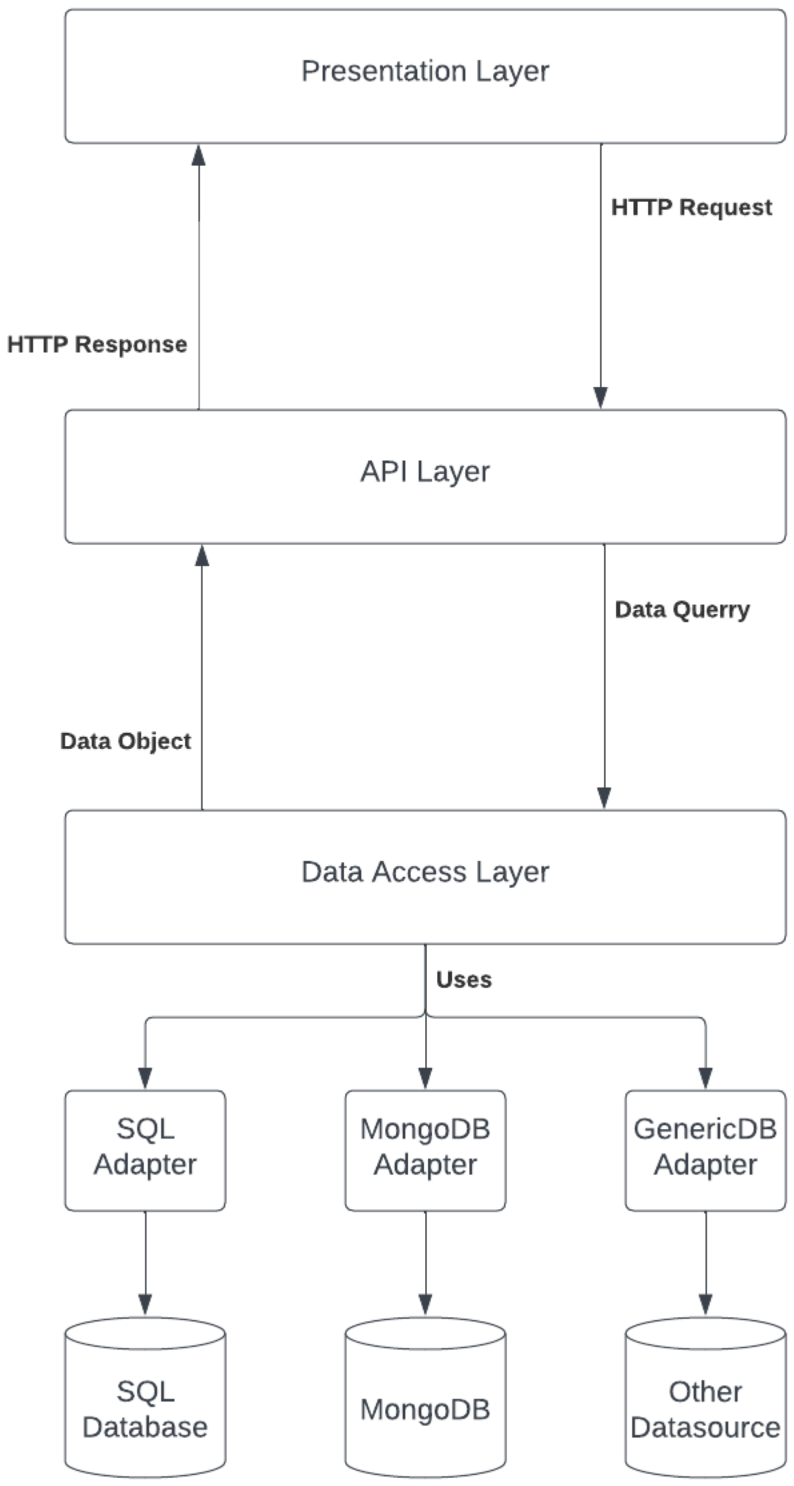

Figure 1.1: Archtecutral Overview of the AMO System

2 Sprint and Backlog Documentation

This chapter describes how the development workflow and the versioning system is setup. It serves as an explainer to understand how the project is maintained and a documentation to standardise and speed up the planing process. The process is described in three parts, preparation, implementation and the documentation of the sprint after it happens (section 2.1, 2.2 and 2.3 respectively). It should be noted that this project follows many common practices of modern software development, like semantic versioning and dividing the development process into sprints. These practices are more of a guideline however and will not be followed exactly, as they are simply *overkill* for this project. They more or less serve as an inspiration and provide loose frameworks to make development more manageable.

2.1 Semantic Versioning and Sprint Planing

2.1.1 Semantic Versioning

Semantic versioning is a practice used to standardize the way versions get assigned to software by giving each digit meaning. A version is a three digit number of integers, e.g.: v1.2.0 . Each digit indicates a different kind of change / version: *major*, *minor*, *patch*. In this project, a new version can be called an *increment*. A *major* increment breaks backwards compatibility (e.g: changing the API). *Minor* increments stay backwards compatible but add new features to the software. *Patch* increments fix bugs or add / change smaller details.

2.1.2 Sprint Planing

The term *sprint* is not exactly fitting, it will be more of a marathon. A sprint defines a short time span predefined by the development team. Due to external factors, the timespan for each sprint won't be predefined. New features will be added once they are implemented and tested. A sprint in this project should be interpreted as a set of requirements, that the developer tries to implement next for a given version.

For each sprint, the developers in charge are free to chose which and how many requirements they implement. Try to select requirements that affect roughly the same area of code (modules, classes, layers, etc...). Before each sprint create a document (less than one page)[1] that lists the issues / requirements and why the need to be implemented. If possible, describe the solution that implements these requirements using bullet points or full sentences. This is not a form submitted for approval and is only for documentation purposes. In case of really small patches or bug fixes, there is no need to create a separate page. As a rule of thumb: If you would increment the version, create a documentation.

2.2 Implementation

2.2.1 Coding Guidelines

Flexibility and extendability are the main qualities the code should fulfill (apart from actually *functioning*, that is). For this reason it is advised to not take to many new features for a new sprint to implement and taking time to fully implement and test.

When coding, adhere to the best practices of the used language to handle style. Please refer to an official style documentation. Also, look at the following chapter **??** to see how the style is defined or refer to section **??** to get a list of formatters, linters and other tools to ensure a uniform code style. Please consider the architecture of your code. Softwaredesignpatterns offer standard implementations and solutions for many common problems. Use them where sensible. Ask the question, if the code is open for extension when another developer needs to come back to this code and add a feature.

Provide docstrings for the code you write. Write them, so that another developer can use the code only with the informations provided by the signature and the docstring. Docstrings are used to automatically create and update the documentation for the code. More on the style of docstrings in chapter **??**. Unittest your code. Especially the functionality and any thrown errors should be verified by unittests. Unittests generally adhere to the same guidelines as production code.

2.2.2 Branching Strategy

The repository will provide two branches: The main branch containing the production code and the development code, where new features will be integrated and tested. To run a new sprint, create a branch from development and start implementing. Once implementation is finished, the code will be tested and integrated into development using a pull request. Pull requests to the main and development branch need approval from the owner of the repository.

[1]NOTE: I will create and link a template for this.

2.3 Sprint Retrospective

2.3.1 General Retrospective

2.3.2 More Information

After finishing the implementation, create a pullrequest to integrate your changes with the development branch. A pullrequest needs to be reviewed and approved by the owner of the repository. If the pullrequest has been approved and integrated into the development branch, test the application manually to verify that the rest of the application can execute normally.

Create a small retrospective of your development. This is a document[2], less than one page in size, that shortly explaines the changes / implemented features and lists the implemented requirements / issues. It is possible to discard some of the requirements that were listed in the planning document. Reasons may be that the issue was closed unexpectedly, is not relevant or is not possible to implement. Be carefull when discarding requirements and consider, if there is a way to implement them. Not enough time *is not* a valid reason to discard an issue and be done early with the pull request. Take more time and open the pull request later, when the issue is implemented and tesed. If there have been any, please list some lessons learned at the end of the document.

Please create a document regardless of whether or not a planing document has been created to start the implementation (e.g. a bugfix)[3]. This serves as a documentation and an overview over what has been done.

[2]NOTE: I will create and link a template for this.
[3]If possible, I will provide a smaller document template for bugfixes