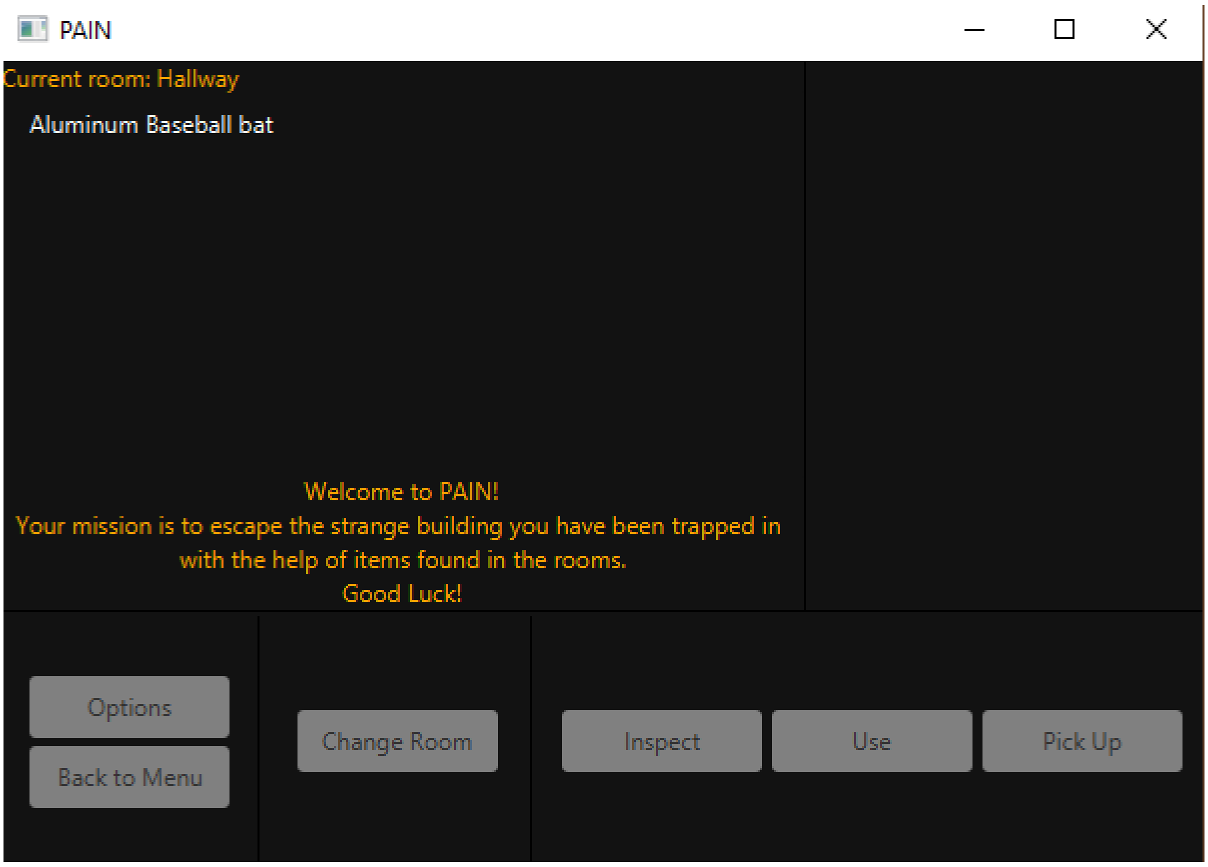
Sommersemester 2022

Softwareentwicklung 2

MI7

PAIN



Current room: Hallway

Aluminum Baseball bat

Welcome to PAIN!
Your mission is to escape the strange building you have been trapped in with the help of items found in the rooms.
Good Luck!

Rometh Umic-Senol (ru006)

Manith Mam (mm334)

André Schwabauer (as439)

Kevin Cipric (kc028)

https://gitlab.mi.hdm-stuttgart.de/as439/se-2-pain

HOCHSCHULE
DER MEDIEN

Table of Contents

Short description

Our first enthusiastic idea was to create a whole big text-adventure. Obvioulsy, that was way too ambitious and too time consuming. After the first few meetings and first coding sessions we realized, we must cut down our imaginations. We have agreed to concentrate on fulfilling the given criteria first.

We now have a click-game where you need to escape the building by finding essential items to progress. The game has a UI which is easy to understand. You click yourself through the rooms and pick up, inspect, or drop the items lying in rooms by clicking the related buttons with the mouse. But you need to hurry because a timer could deny your escape.

Starting class

The Main-Method is placed in *src/main/java/de.stuttgart_hdm.hdm.mi.se2* and is called **Main**.

Note

We specialized on the Open-Close principle, so we have an easy way to add further items and rooms but keep our quintessential information private. This was important for us. Nevertheless, our program could have some downsides like no additional graphics or bugs we didn't notice yet.

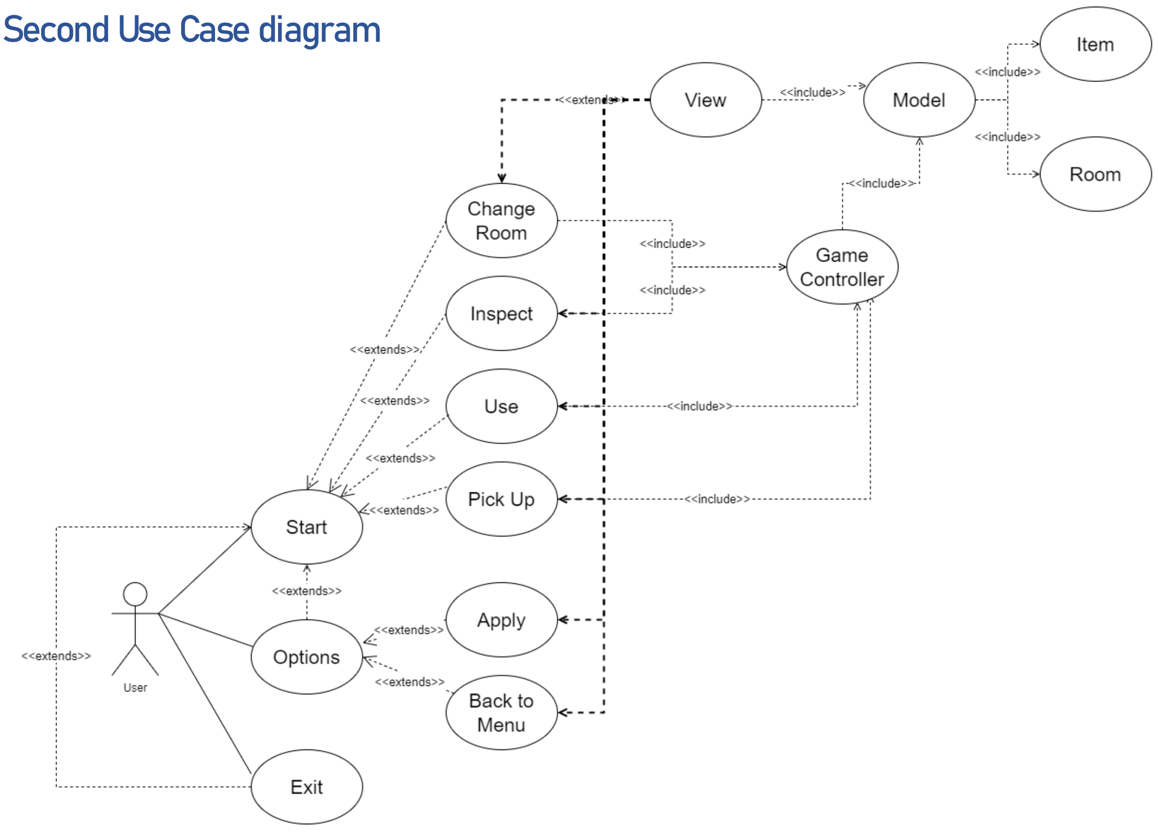
Before we wrote any code, we tried to have a basic plan. So, we created our first Class Diagram. We knew which objects we needed surprisingly quick. We needed a player, items, rooms therefore we had already a foundation of our plan.

Important Note:

We had some issues with our UI running on **Linux** otherwise it works perfect on **Windows**. We couldn't solve this problem.

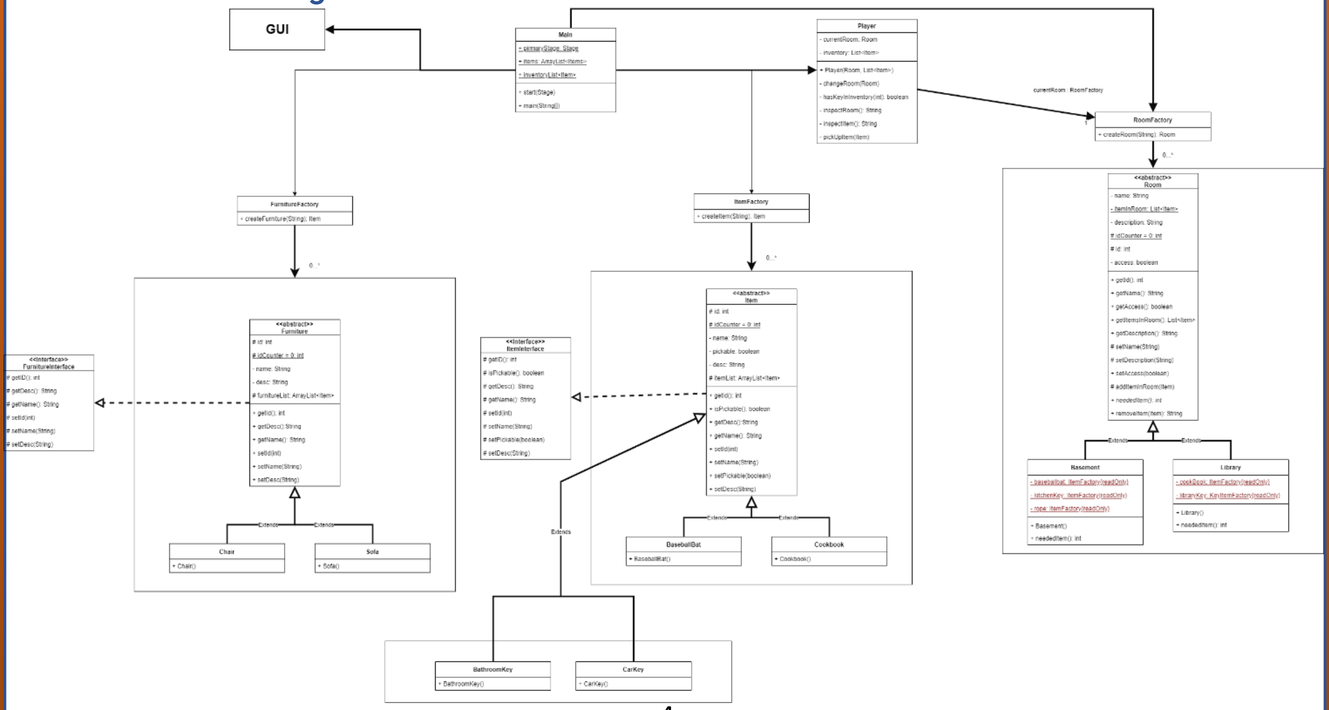
Diagrams

Second Use Case diagram

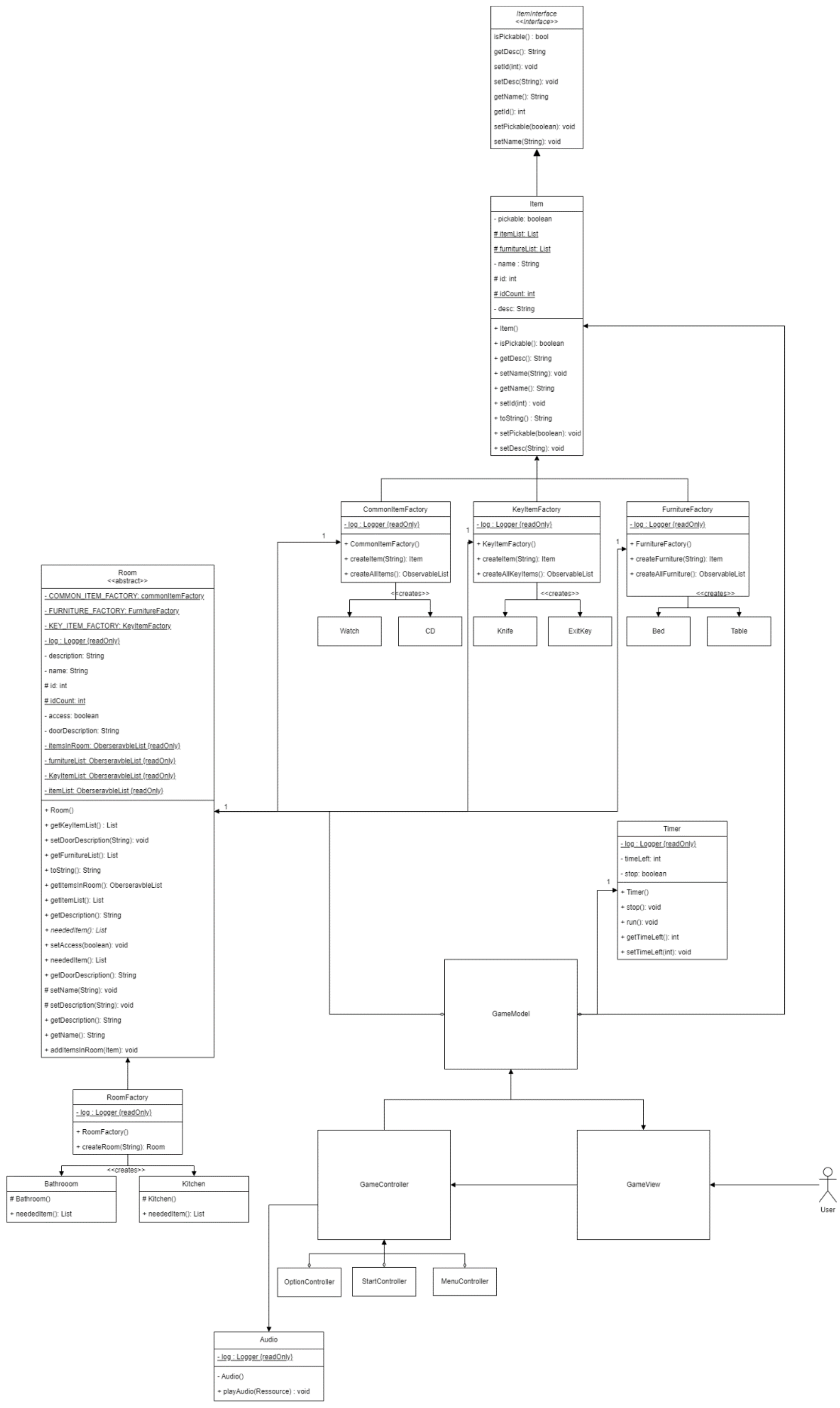


Note: After the first Use Case diagram we learned how to create a Use Case diagram properly. So, we created a right one and transferred our new learned knowledge into it.

Second Class diagram



Final Class Diagram



Statement

Architecture

- Created our own Interface **ItemInterface** (implemented in the abstract class **Item**). Because if we want to add a new type of items it should partly behave like the others.
- **Timer** implements **Runnable**
- Two abstract classes **Room** and **Item**
- Subclasses like **Basement, Bathroom** inheritance from **Room**
- Subclasses like **Glasses, Hammer, Chair** inheritance from **Item**
- Factories for Rooms (**RoomFactory**), factories for all our item types (**FurnitureFactory, KeyItemsFactory, CommonItemFactory**). We did this to easily add new CommonItems, KeyItems, Furnitures without changing the base code and like this we keep more privacy.
- If we create rooms, items, etc. we access it only through factories
- We are using the Model View Controller pattern for our GUI (**package controller, model, view** represent the MVC pattern)
- We have Controller for the **game itself**, **the menu**, **the options** and **for losing the game**
- *Fxml-files* are in **resource/fxml**

Clean Code

- We tried to set variables to final if it was possible
- Tried to keep as many properties and variables private as possible or at least protected
- To read properties we are only using getters and we are using setters in Subclasses to set their own property to keep encapsulation
- loadFxml in **gui/Utils,** playerAudio in **Audio** had to be static
- same goes for getKeyItemList in **Room**
- static methods in **GameView** and in **GameModel**
- it was inevitable to not have a reference to the inventory **List**

Tests

- **RoomTest** tests *needItem(), addItemsInRoom(), properties of Rooms* and a negative test for *properties of Rooms*
- **ExceptionTest** tests if **IlegalArgumentException** of *createRoom(String roomType)* is thrown like expected and the same for **InterruptedException** from our thread
- **ItemFactoryTest** tests the unique Ids and the *properties of Item*

GUI

- Our GUI is created with *fxml*-files except for error screens which are created with java code
- Everything for our UI is located under the directory *gui*

Exceptions

- **RoomFactory** throws **IlegalArgumentException** if the parameter in *createRoom(String roomtype)* is an non-existing room type and all factories follow this principle. **GameModel** handles this thrown Exception.
- **Timer** catches an unchecked **InterruptedException**

Threads

- **Timer** uses a thread to have a displayed time in game, also the time is getting logged.
- If the timer hits 0 the player loses the game
- **Timer** with its thread is started in **Main**

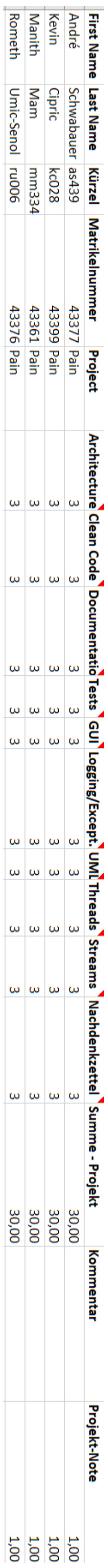
Streams and Lambda-Functions

- With the **Item Watch** we constantly check the remaining time. We implemented it with a **Stream** in **Timer**
- To check which **Item** is needed for opening a door, we coded it also with stream, filter and map
- Generally, we mostly use **Streams** to check the **Items´** Id
- Factories use a ,,**Lambda-like" syntax** to create the objects
- Button events exclusively handled with **Lambda expressions**

Logging

- Our logging.xml is found under resources/**log4j2.xml**
- **Exception** are logged with log.warn()
- **Timer** Thread is logged
- Log statements are saved under logs/**logFile.log**

Filled evaluation



First Name	Last Name	Kürzel	Matrikelnummer	Project	Architecture	Clean Code	Documentatio	Tests	GUI	Logging/Except.	UML	Threads	Streams	Nachdenkzettel	Summe - Projekt	Kommentar	Projekt-Note
André	Schwabauer	as439	43377	Pain	3	3	3	3	3	3	3	3	3	3	30,00		1,00
Kevin	Cipric	kc028	43399	Pain	3	3	3	3	3	3	3	3	3	3	30,00		1,00
Manith	Mam	mm334	43361	Pain	3	3	3	3	3	3	3	3	3	3	30,00		1,00
Rometh	Umic-Senol	ru006	43376	Pain	3	3	3	3	3	3	3	3	3	3	30,00		1,00