MedievalText

MIDAS

Team Midas

Aaron Ackerman

Hicham Dioury

Neville Leung

February 17, 2020

Table of Contents

Introduction

MedievalText is a text-based adventure game that you can play from the command line. You wake up on the beach of a strange and unfamiliar island. With nothing but your wits, you must explore the island, solve its puzzles, and defeat the three guardians that rule over it. Each puzzle and enemy encounter requires the use of a specific item, which are found throughout the map, obtained by trades, or dropped by enemies.

The map is laid out in a two-dimensional grid, each square representing a different location, with the player being able to move to adjacent squares. Beginning with a small tutorial section, the map then branches into three main sections, each with its own boss monster.

The user has five in-world commands at their disposal:

- **Look** (e.g. `look room`): A brief description of a location or object is given
- **Move** (e.g. `move east`): The player moves between two locations
- **Take** (e.g. `take sword`): The player picks up an item off the ground
- **Drop** (e.g. `drop coin`): The player drops an item, leaving it on the ground
- **Use** (e.g. `use potion`): The player uses a specific item to solve a puzzle

In addition, there are four helper commands:

- **Map** (e.g. `check map`): A map with the player's location is displayed
- **Help** (e.g. `help`): Gives a list of the available commands
- **Save** (e.g. `save mySave`): Creates a save (.txt file) with the name `mySave`
- **Load** (e.g. `load mySave`): Loads the save with the name `mySave`

To make the game simpler to play, all commands and objects have a set of "aliases" they can also go by. For example, `move east`, `travel east`, and `go east` are all recognized by the game and accomplish the same task.

In the following pages, we will cover how the development of the project will be managed and the development process our team will follow. In the final pages, we will cover the initial design of our project, and its rationale.

Project Management

In this section, we will cover the organization of the team, including which roles each member will fill during the three phases, and how we plan to mitigate risk. Problems that the team might potentially face, and our plans for resolving them, are discussed.

Team Organization

Team members will rotate between the three roles (phase lead, librarian, and QA lead) during each phase to ensure that everyone has a turn in each role.

Team Members	**Design**	**Implementation**	**Testing / Maintenance**
Aaron Ackerman	Phase Lead	QA Lead	Librarian
Hicham Dioury	Librarian	Phase Lead	QA Lead
Neville Leung	QA Lead	Librarian	Phase Lead

Risk Management

Requirements and Design

In order to manage risk, we have chosen to prioritize functionality over scale. The goal of our design is to be simple but adaptable and buildable, like a LEGO piece. We will first build a functional game with only two or three rooms, then expand from there with the remaining time. Here, a self-contained tutorial section not only helps the player, but our development process as well. We have little functionality aside from the basic requirements of the project, so it is difficult to prioritize any parts in particular, but some peripheral functionality (like saving and loading) can be simplified at the expense of flexibility. If a specific task takes more time than expected, another member will be assigned to help.

The tasks and deadlines will be assigned by the phase lead, based on each individual's preferences, capability, and schedule. Weekly meetings will be planned by the phase librarian to chart the progress of these tasks and identify any issues.

If major changes to the design are needed, we will hold a team meeting as soon as possible to discuss options. Certain functionality may have to be left out.

People

If a team member leaves the project, then the scale of the finished state will be adjusted accordingly, and some functionality may have to be simplified. On the other hand, if we gain a new team member, then we will have a team meeting to walk them through the design and see if they have any new ideas. Before any task is assigned to them, they will shadow another member to help them understand how the current system functions.

If a team member lacks the technical skills for a specific task, it will be assigned to another member. A more suitable task will then be assigned instead. They may also shadow another team member to gain a better understanding.

If a team member is unproductive and not meeting deadlines, then the phase lead will speak with them. Any changes to the schedule, or changes to each member's duties, will be added to the agenda of the next weekly meeting. If the problem persists, or if the team member is inaccessible, then the rest of the team will set up a meeting with Dr. Anvik.

Learning and Tools

Though we do not plan to use any tools beyond those already introduced in the course, if a team member needs help, they may ask another member for assistance. If the team as a whole needs help, and research into solutions has turned up nothing, then we will make an appointment with the Lab Instructor or Dr. Anvik.

Development Process

Code Review Process

If any of the developers face a problem during a process, the Phase Lead will approve and resolve the issue . Given a merge conflict or a pull request, the leader will decide which code to use. If any inadequate code breaks the build or disables any of the game's functionality, the developer of that code will hold the responsibility to correct it afterwards or to bring back the previous version. All important adjustments will be discussed at the next weekly meeting, and will be pushed with the permission of the Phase Lead.

Communication Tools

So far, our team has used Google Docs to collectively record notes and ideas on a single page, and for developing the report. Going forward, this will be useful in planning the agenda of meetings, and for recording minutes. Because Google Docs has its own internal version control, it can also be a good way of tracking contributions. Issue tracking, as well as reporting contributions, will be handled through GitLab. Slack can be used for immediate communication, and email for any non-immediate communication.

Change Management

The Quality Assurance Lead of each phase will examine, triage, and resolve all bug reports. If the fault was introduced by a single developer, then they will be assigned to fix the bug. Depending on the urgency of the situation, decisions will be evaluated by the Phase Lead.

- If the bug report is a simple misunderstanding or an easy task, the bug will be fixed and the resolution closed immediately.
- If the bug report leads to an issue that might cascade into a bigger problem, our librarian will set up a meeting to decide which changes should be made.
- The Quality Assurance will be in charge of replying appropriately to any bug report.

Software Design

Design

World

Each interactable object in the world, including characters, items, and locations, all inherit from the superclass GamePanel. Each GamePanel has a unique identifier, the objectCode, that can be used for searching and matching objects. The first two digits of this object code also give its class information, or ClassCode, so that other objects can more easily manage it. There are four main GamePanel classes:

- **Items**: These are the fundamental unit of the gameplay loop, which is based around finding and using items in specific locations.
- **Locations**: Locations allow the user to navigate the map and encounter different puzzles or enemies.
- **Characters**: Divided into Player (the user's character) and NPCs. You can trade with friendly NPCs, or defeat Enemy NPCs, to obtain items. Defeating the three boss characters is the goal of the game.
- **Containers**: Both Characters and Locations hold items. To prevent redundant code, the exchange of items (including taking and dropping) is encapsulated in its own class.

The Map is not a GamePanel itself, but organizes the Locations. The final layout of the world was purposely made flexible, which the Map class accommodates. Given a list of Locations (each with unique coordinates), the Map builds and connects itself, providing each Location a list of pointers to its neighbors.

Game Control

Control of the game, which includes the User Interface as well as saving and loading, is separated from the World. The world objects are essentially just states, which are controlled and changed by the GameUI, or stored and retrieved by the SaveGame and LoadGame.

Design Rationale

All world objects have one thing in common: the user can interact with them. Because of this, they share characteristics and behaviours. They also have similar behaviour for behind-the-scenes functionality, such as saving and loading. We chose inheritance to accommodate these similarities. However, for flexibility, we abstracted control out of the objects themselves and into its own class: GameUI. This organizes the many elements of the game, and channels all of the interactions and changes into a single usable interface.

As each object is being saved (or loaded), a similar serialization process is carried out, though it varies depending on the type of value serialized. For this reason, the serialization and deserialization processes are encapsulated in their own classes, which GamePanels call to save each of their attributes.

Appendices

Appendix A: UML Diagrams

Figure 1: Condensed UML Diagram

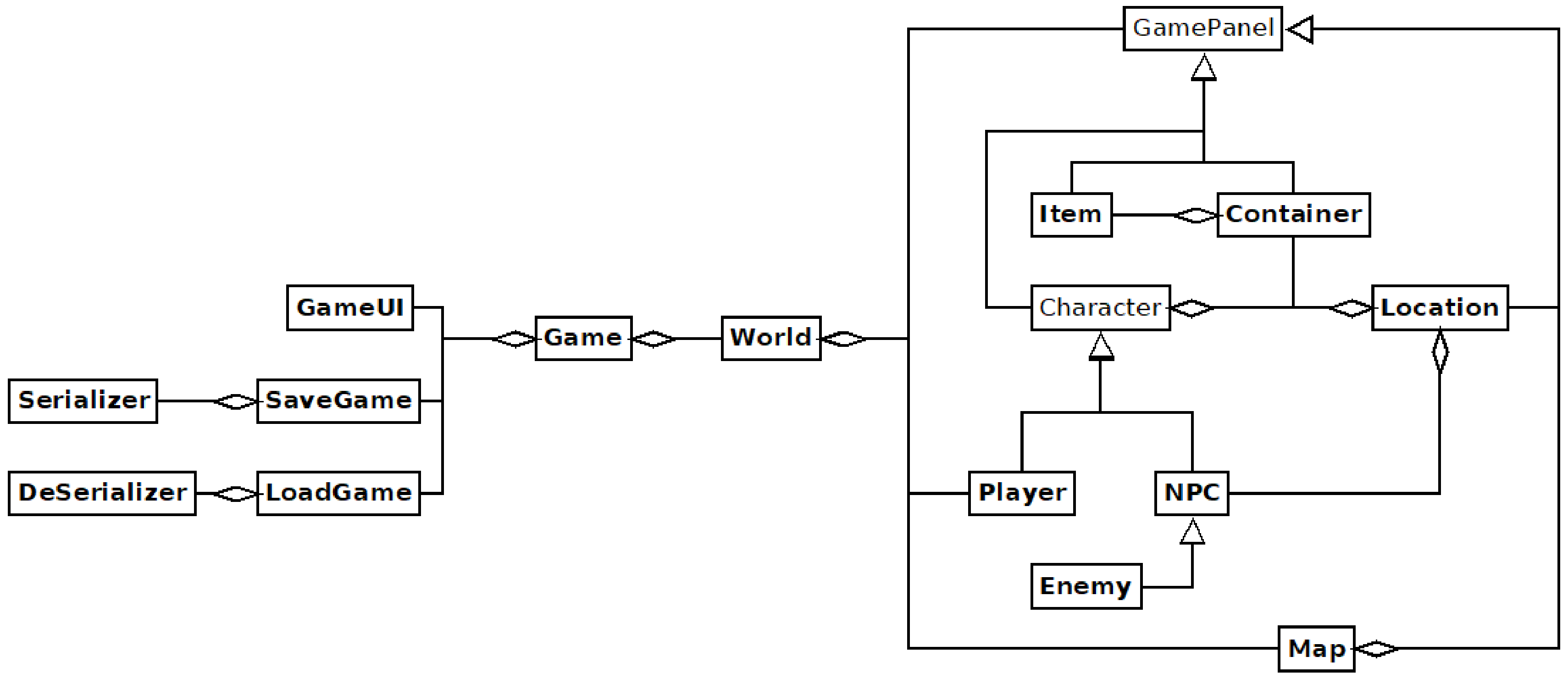


Figure 2: Part 1 - World UML Diagram

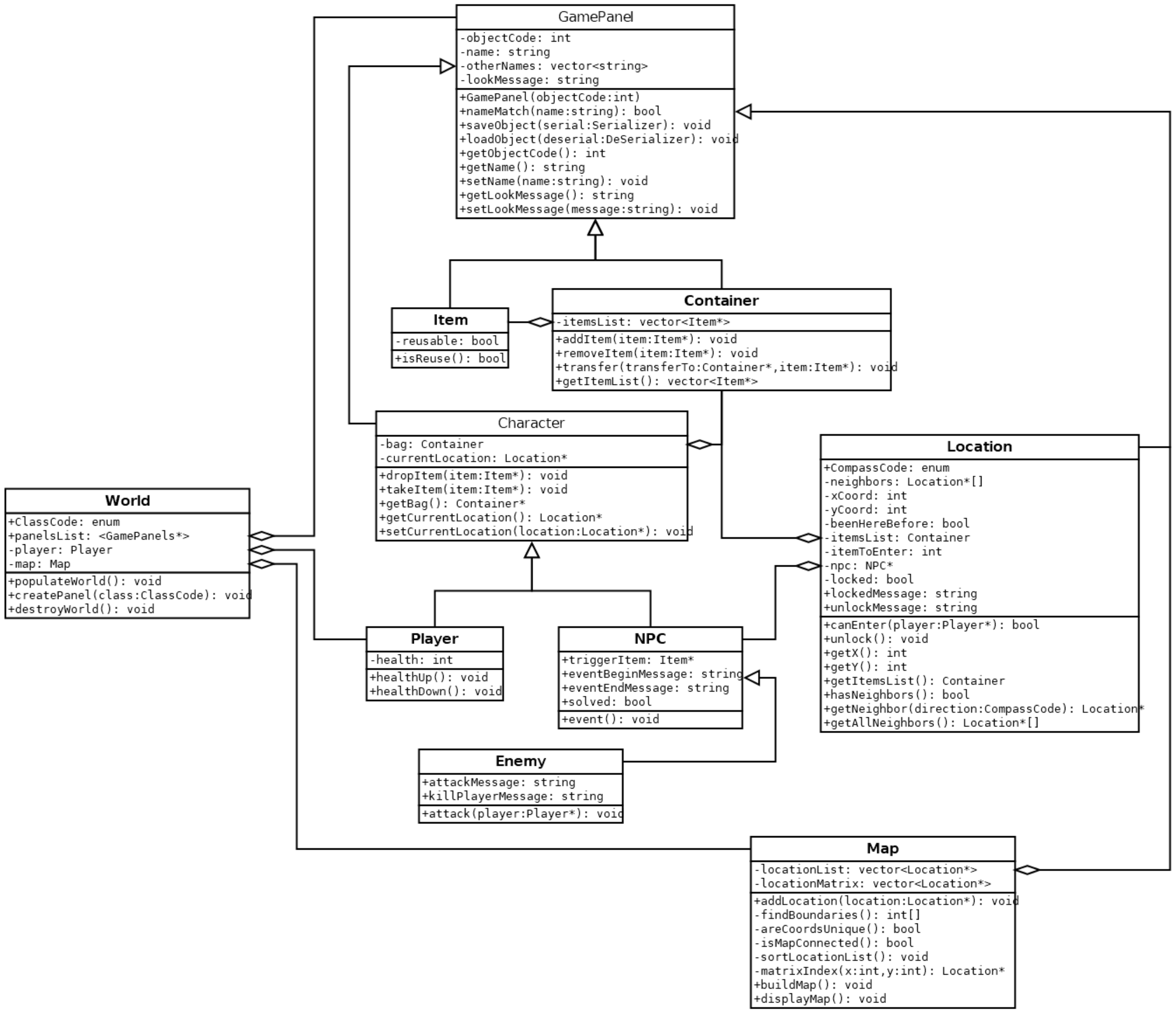
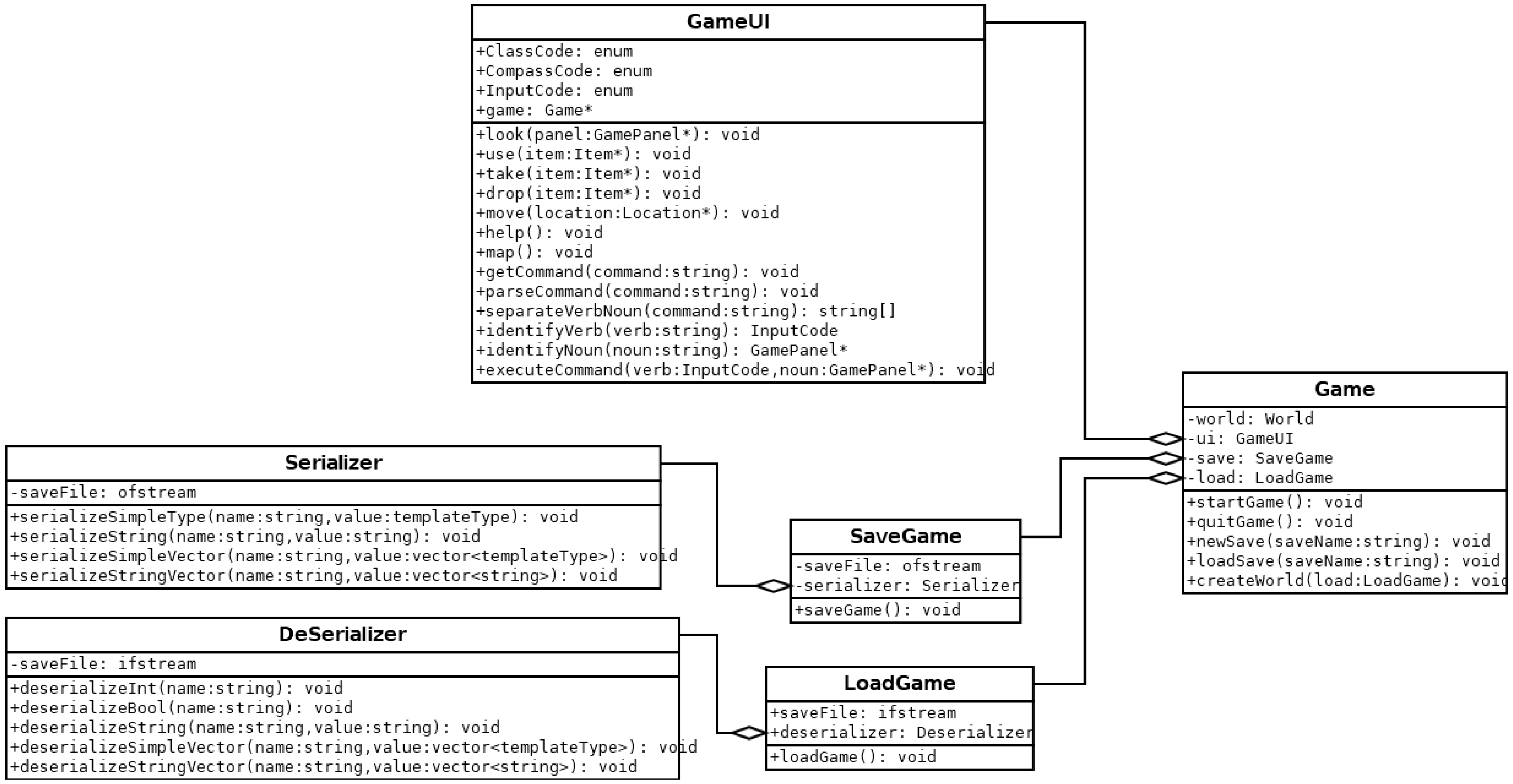

GamePanel
-objectCode: int
-name: string
-otherNames: vector<string>
-lookMessage: string
+GamePanel(objectCode:int)
+nameMatch(name:string): bool
+saveObject(serial:Serializer): void
+loadObject(deserial:DeSerializer): void
+getObjectCode(): int
+getName(): string
+setName(name:string): void
+getLookMessage(): string
+setLookMessage(message:string): void
Item
-reusable: bool
+isReuse(): bool
Container
-itemsList: vector<Item*>
+addItem(item:Item*): void
+removeItem(item:Item*): void
+transfer(transferTo:Container*,item:Item*): void
+getItemList(): vector<Item*>
Character
-bag: Container
-currentLocation: Location*
+dropItem(item:Item*): void
+takeItem(item:Item*): void
+getBag(): Container*
+getCurrentLocation(): Location*
+setCurrentLocation(location:Location*): void
World
+ClassCode: enum
+panelsList: <GamePanels*>
-player: Player
-map: Map
+populateWorld(): void
+createPanel(class:ClassCode): void
+destroyWorld(): void
Location
+CompassCode: enum
-neighbors: Location*[]
-xCoord: int
-yCoord: int
-beenHereBefore: bool
-itemsList: Container
-itemToEnter: int
-npc: NPC*
-locked: bool
+lockedMessage: string
+unlockMessage: string
+canEnter(player:Player*): bool
+unlock(): void
+getX(): int
+getY(): int
+getItemsList(): Container
+hasNeighbors(): bool
+getNeighbor(direction:CompassCode): Location*
+getAllNeighbors(): Location*[]
Player
-health: int
+healthUp(): void
+healthDown(): void
NPC
+triggerItem: Item*
+eventBeginMessage: string
+eventEndMessage: string
+solved: bool
+event(): void
Enemy
+attackMessage: string
+killPlayerMessage: string
+attack(player:Player*): void
Map
-locationList: vector<Location*>
-locationMatrix: vector<Location*>
+addLocation(location:Location*): void
-findBoundaries(): int[]
-areCoordsUnique(): bool
-isMapConnected(): bool
-sortLocationList(): void
-matrixIndex(x:int,y:int): Location*
+buildMap(): void
+displayMap(): void

Figure 3: Part 2 - Game Controllers UML Diagram


GameUI
+ClassCode: enum
+CompassCode: enum
+InputCode: enum
+game: Game*
+look(panel:GamePanel*): void
+use(item:Item*): void
+take(item:Item*): void
+drop(item:Item*): void
+move(location:Location*): void
+help(): void
+map(): void
+getCommand(command:string): void
+parseCommand(command:string): void
+separateVerbNoun(command:string): string[]
+identifyVerb(verb:string): InputCode
+identifyNoun(noun:string): GamePanel*
+executeCommand(verb:InputCode,noun:GamePanel*): void
Game
-world: World
-ui: GameUI
-save: SaveGame
-load: LoadGame
+startGame(): void
+quitGame(): void
+newSave(saveName:string): void
+loadSave(saveName:string): void
+createWorld(load:LoadGame): voi
Serializer
-saveFile: ofstream
+serializeSimpleType(name:string,value:templateType): void
+serializeString(name:string,value:string): void
+serializeSimpleVector(name:string,value:vector<templateType>): void
+serializeStringVector(name:string,value:vector<string>): void
SaveGame
-saveFile: ofstream
-serializer: Serializer
+saveGame(): void
DeSerializer
-saveFile: ifstream
+deserializeInt(name:string): void
+deserializeBool(name:string): void
+deserializeString(name:string,value:string): void
+deserializeSimpleVector(name:string,value:vector<templateType>): void
+deserializeStringVector(name:string,value:vector<string>): void
LoadGame
+saveFile: ifstream
+deserializer: Deserializer
+loadGame(): void

Appendix B: Sequence Diagrams

Figure 4: New Game Sequence

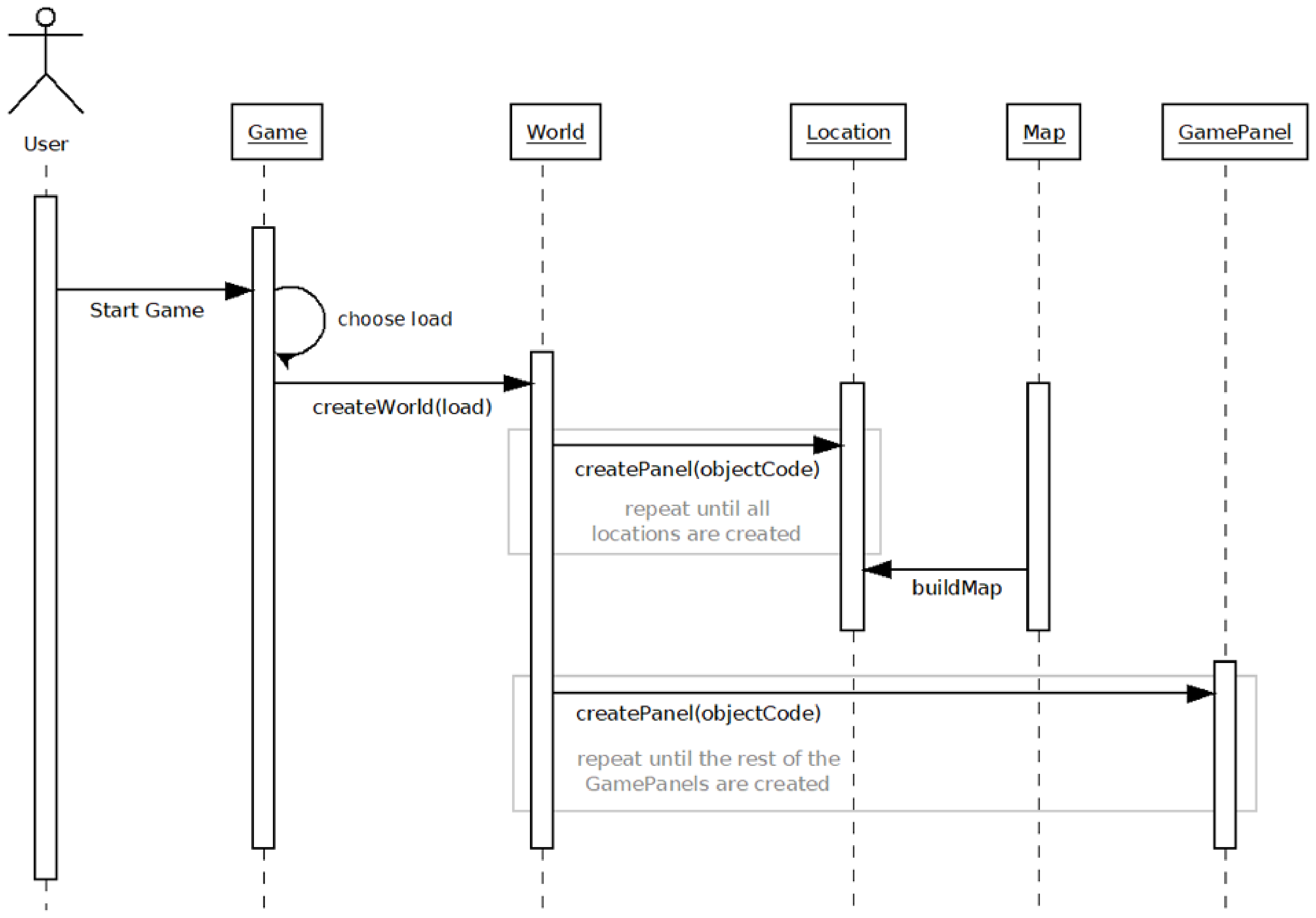


Figure 5: Sample User Command Sequence

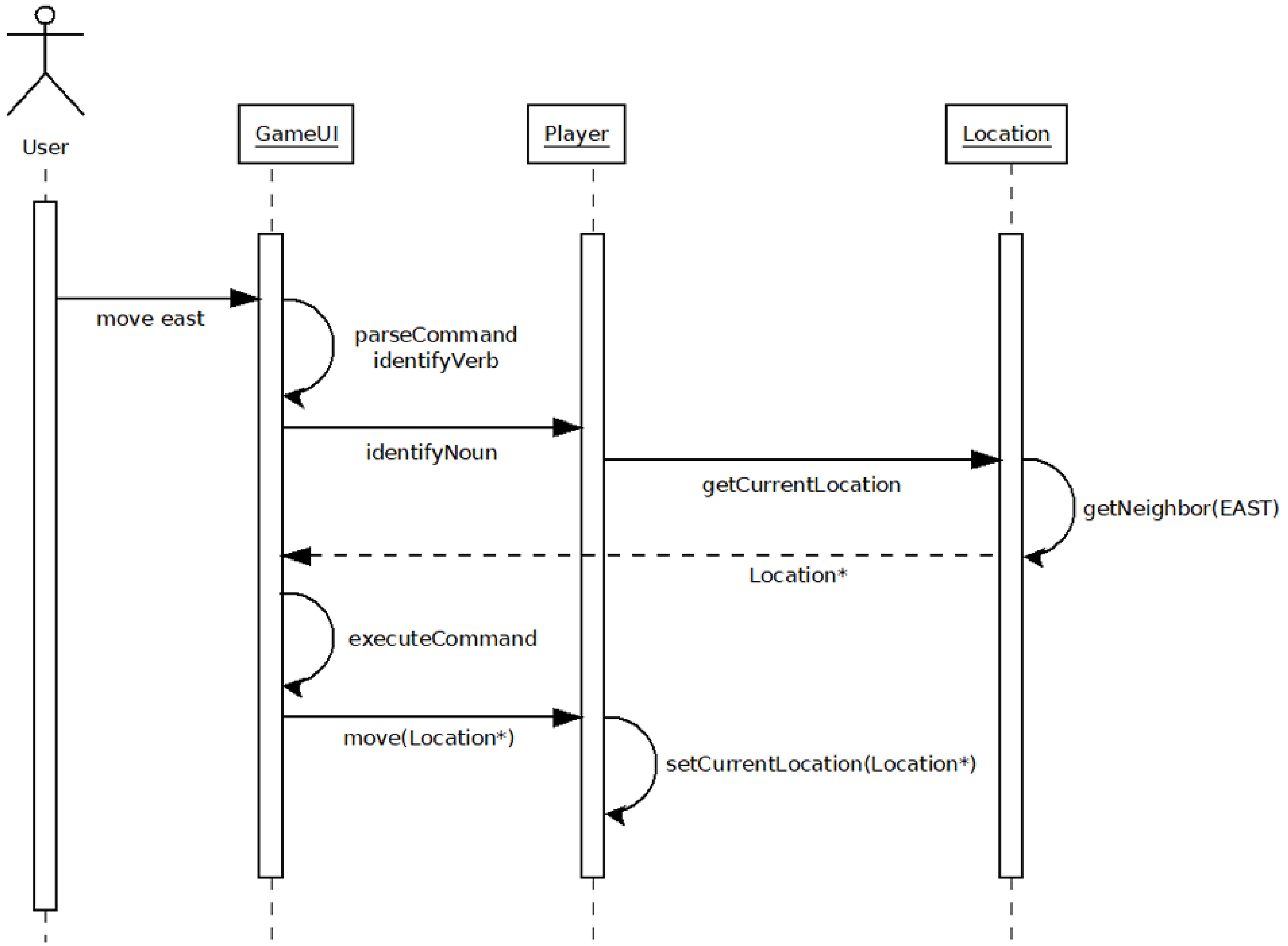


Figure 6: Save Game Sequence

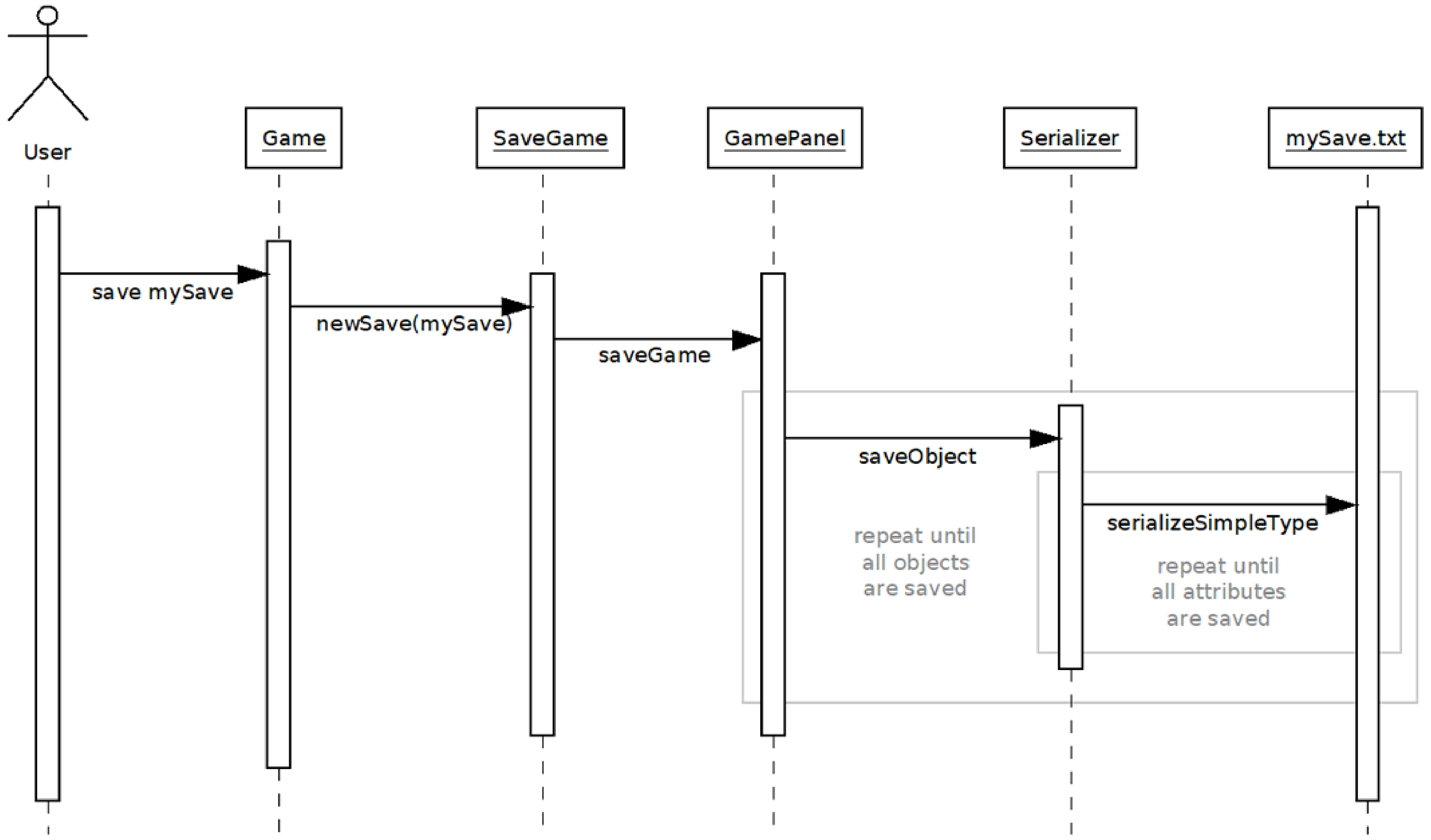
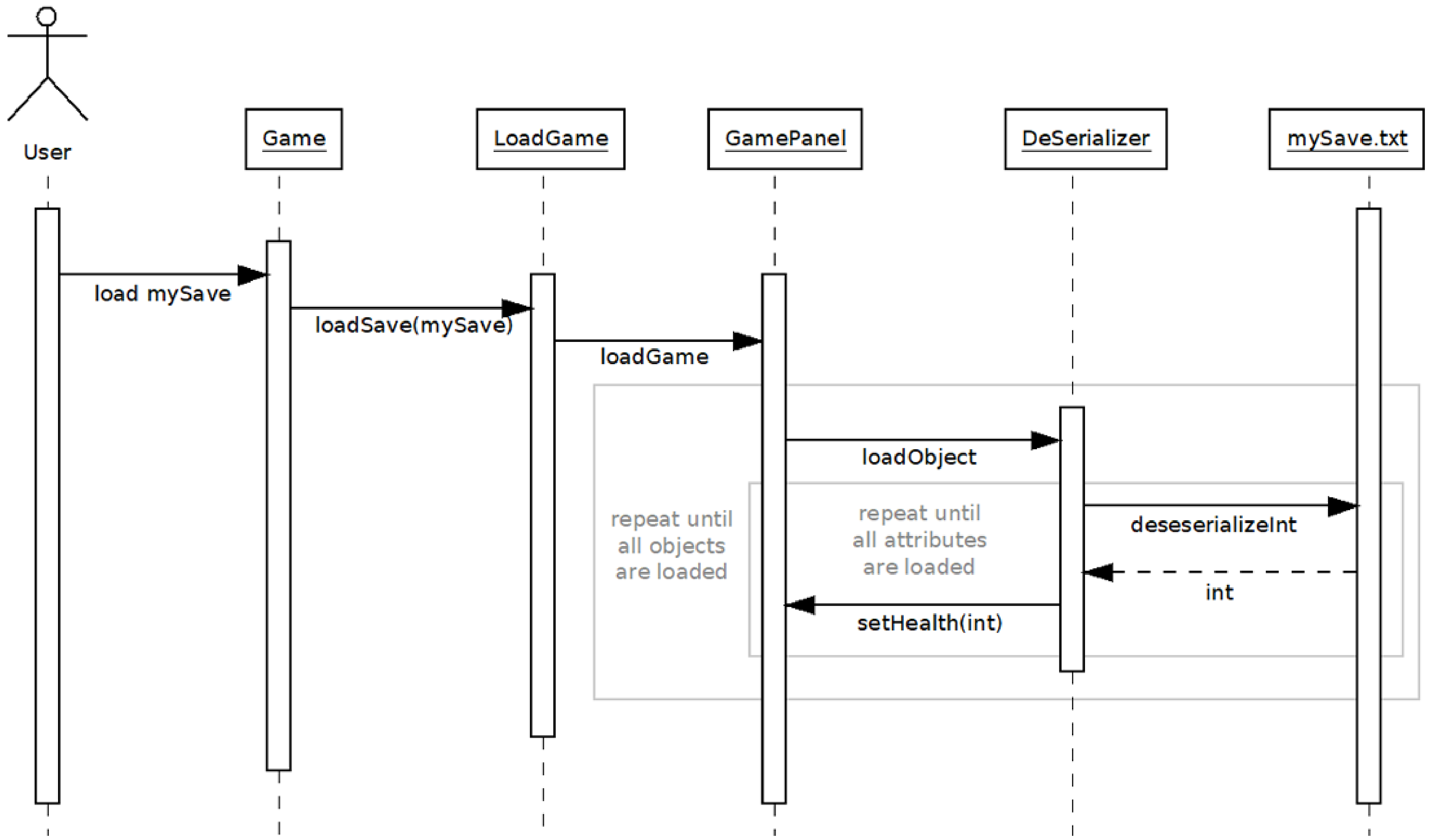


Figure 7: Load Game Sequence


User
Game
LoadGame
GamePanel
DeSerializer
mySave.txt
load mySave
loadSave(mySave)
loadGame
loadObject
repeat until all objects are loaded
repeat until all attributes are loaded
deseserializeInt
int
setHealth(int)