System Design Document for MemoryShape

Edenia Isaac
Filip Linde
Nils Johnsson
Kevin Svensson

Date: 2020-10-23
Version: 3

Contents

1. Introduction

This report will discuss the technical aspects of the application MemoryShape. The project's purpose was to create an easily expandable and testable application, with an object oriented design. To achieve this the software was developed with minimal dependencies and minimal coupling between different modules by encapsulating data as thoroughly as possible.

Memoryshape is a single player game that aims to help users improve their memory in a fun, engaging and challenging way.

1.1 Definitions, acronyms and abbreviations

- **FXML file** - File that defines the user interface of a JavaFX application in a user interface markup language.
- **Travis** - A hosted, distributed continuous integration service used to build and test projects in GitHub.[1]
- **MVC** - Model View Controller: A design pattern to structure the code on a large scale.[2]
- **GUI** - Graphical User Interface: The visual interface of the program, seen by the user.[3]
- **UML** - Unified Modeling Language: A diagram describing how different parts of the program work together.[4]
- **Domain model** - A domain model is a visual representation of conceptual classes in the model.[5]
- **User Stories** - A small story that describes a type of user, what they want, why they want it, and how to achieve it.[6]
- **Persistent data management** - information that's stored when the application is shut off.
- **JSON** - JavaScript Object Notation: A lightweight format for storing and transporting data.[7]

2. System architecture

The application is written in the Java programming language and JSON is used for storing data. The MVC design pattern is used as a base for the structure of the project.

On startup the application creates a maincontroller which initialises the menucontroller. The menucontroller loads data from the JSON file and the graphical user interface displayed on screen is a menu. When the user interacts with the menu buttons, the menucontroller updates the scene. On game start the menucontroller calls it parentcontroller, the maincontroller, which initialises a boardcontroller. The boardcontroller displays a board full of clickable cards on screen. If the user decides to return to the menu the boardcontroller calls it's

parentcontroller, the maincontroller, which initialises a new menucontroller. On shutdown the maincontroller updates the JSON file with the new high scores.

3. System design

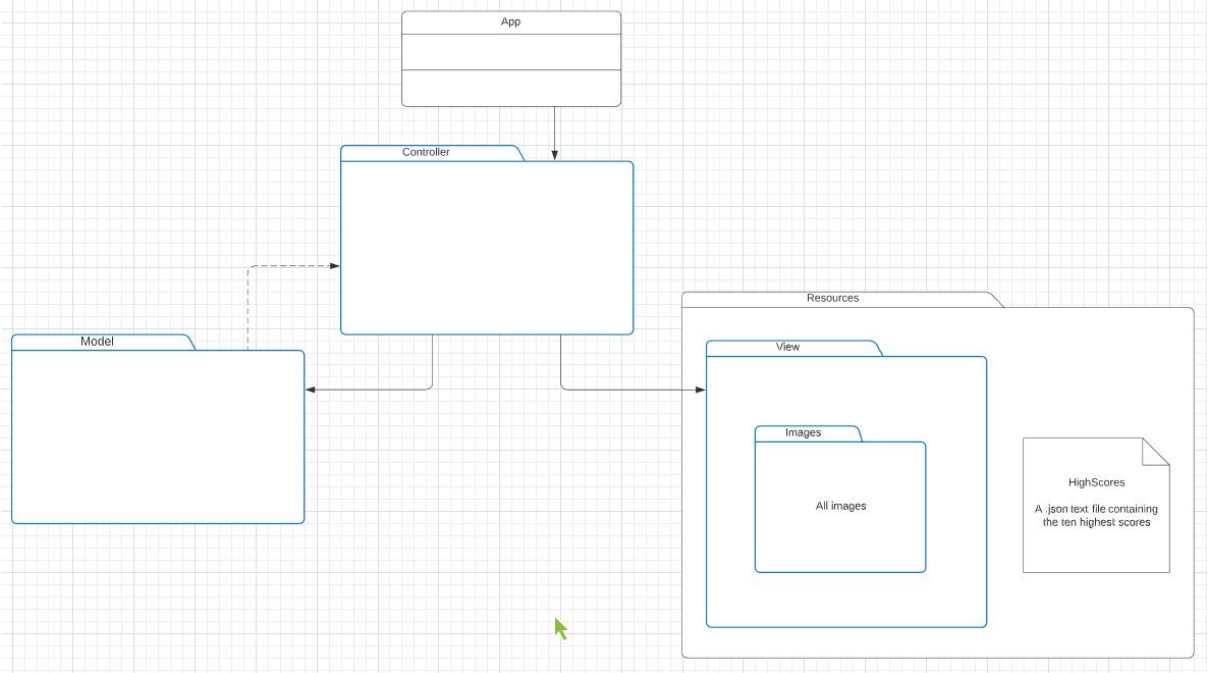

Figure 1: Top level UML-diagram of the MemoryShape application.

3.1 Dependencies and MVC

The MVC pattern was implemented by dividing the codebase into three packages: model, view and controller (see figure 1). Seeing as javaFX was used for the graphics, the view package only contains FXML files and no java files. The controllers to the FXML files were placed in the controller package. Using javaFX it is actually these controllers that manipulate the view presented to the user. This clashes with the single responsibility principle since the controller classes manage input from the user while simultaneously updating the view. Since the view does not contain java files the whole view package was placed inside of a resource folder so that Maven would be able to build the project. The design model is completely independent from the view and the controllers.

3.2 Encapsulation

An important principle for all object oriented applications is encapsulation. Encapsulation protects the state of objects and reduces human errors [8]. The principle has been followed throughout this project by always setting objects' attributes to private when possible. The attributes are instead modified via public methods within the objects class. To encapsulate the

model from other packages relevant information is always sent as a copy of the original object, so that modifications will not affect the model.

3.3 Design model and domain model

The domain model closely resembles the design model of the program. It displays the main parts within the application and some connections between them (see figure 2). Since the domain model is shown to all the parties of the development, including the customer, it is not a very technical description compared to a UML class diagram (see UML-ClassDiagram.pdf in the documents folder). Still, the domain model resembles the design model in that it looks like a simplification of it. This is because they both depict the same thing. While the design model includes all of the logical technical parts of the model, the domain model shows only what is actually apparent in the application, be that visually or audibly. You might say that the domain model is how the customer imagines the application works, whereas a developer imagines the more technical design model.

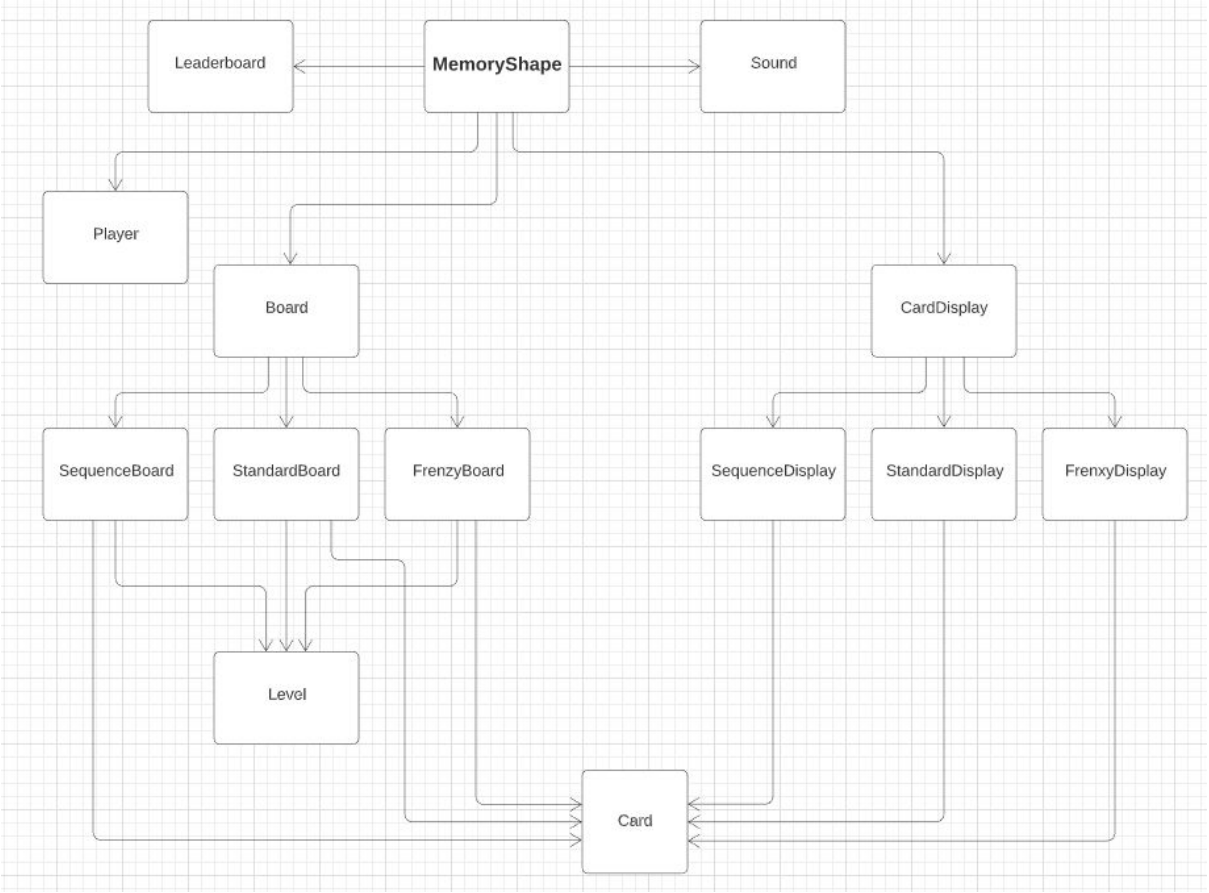

Figure 2: MemoryShape's domain model.

3.4 Design patterns

The game class is a facade for the design model used by the maincontroller class. Inputs from the user are sent to the main controller then into the design model through the facade to update the conditions within the model. To update the controllers, which in turn update the view, the observer pattern is used. The observer sends an updated version of what should be displayed on the screen. This was achieved using the iterator pattern. An iterator of all the parts that should be displayed is sent to one of the controller classes so that the view can be updated. In most cases the view would observe the model but since the controller classes also control the view, it is one of the controller classes that observes the design model. Besides communicating through the observer or to the game class directly there is one specific dependency from the controller package to the model. One of the controller classes has access to the card class. Although the controller package is not meant to have a dependency on the model other than the facade, the use of the card class has a very practical use and saves us from some duplicating a lot of code in the controller package. Strategy pattern was used so that the application can run three different modes of the game. A template method in the game class uses a board and a card display which can each be dynamically set to three different types. This allowed for a lot of code reuse for the different modes since they have much in common.

4. Persistent data management

MemoryShape uses persistent data management for saving highscores on the leaderboard between sessions. The highscores are stored in a JSON text file, which contains the player's name, score and the played game mode. The application can read the file to display the high scores on the leaderboard, and write in the file when a new high score is being added.

The shapes on the cards, the background and the rest of the images are also stored while the application is shut off. They are all kept in a directory named “images” in the view package, from where they are loaded to the application when it starts and when the player interacts with the cards.

5. Quality

MemoryShape uses analytic tools to ensure good quality in the project.

5.1 Continuous Integration

In order to discover bugs quickly MemoryShape uses continuous integration with Travis. Travis was easily integrated into github where all group members can see whether or not a build was successful. Link to our Travis CI.

5.2 Dependency Analysis

To help analyse the dependencies within the project STAN was used. Because the view package does not have any classes, it only has FXML files, its dependencies are not shown. The project has no circular dependencies and the model is completely independent.

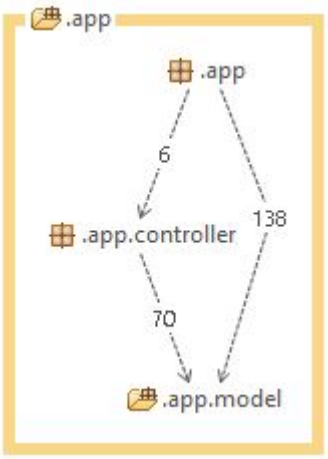

Figure 3: Dependencies between MemoryShape's packages.

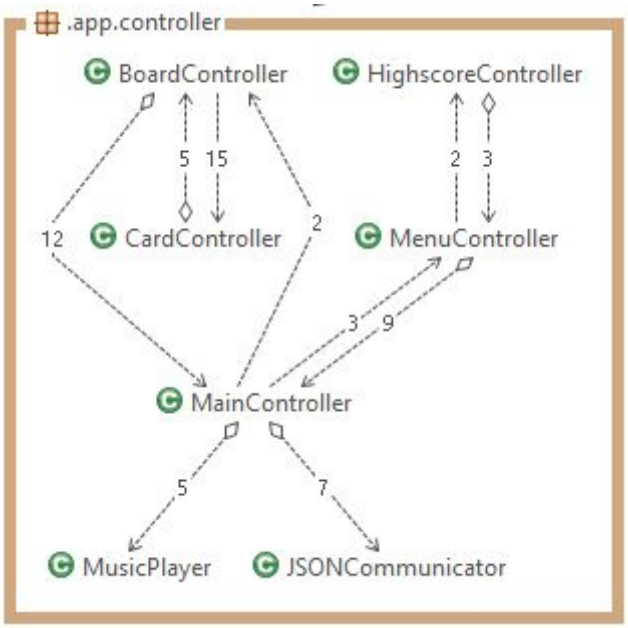

Figure 4: Dependencies within MemoryShape's controller package.

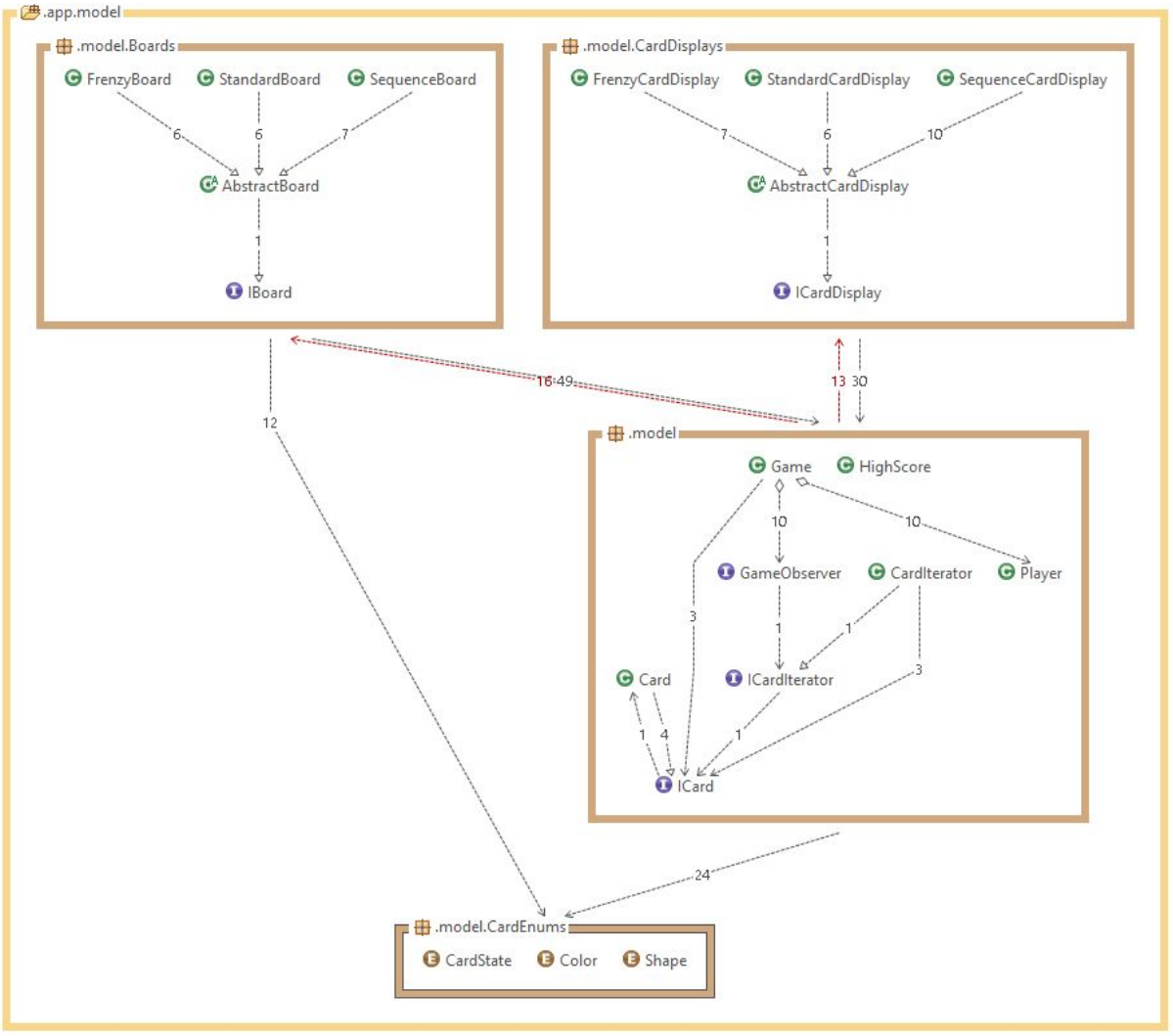

Figure 5: Dependencies in MemoryShape's model package.

5.3 Access control and security

N/A

References

[1] Wikipedia, *Travis CI*. Edited 25 September 2020. Retrieved 10 October 2020. [Online]. Available: https://en.wikipedia.org/wiki/Travis_CI

[2] T. Reenskaug, J. Coplien, *The DCI Architecture: A New Vision of Object-Oriented Programming*. Artima Developer, 20 March 2009. Retrieved 1 October 2020. [Online]. Available: https://web.archive.org/web/20090323032904/https://www.artima.com/articles/dci_vision.html

[3] Nationalencyklopedin, *GUI*. Retrieved 3 October 2020. [Online]. Available: https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/gui-(grafisk-yta)

[4] Source Making, *UML: Introduction*. Retrieved 3 October 2020. [Online]. Available: https://sourcemaking.com/uml/introduction

[5] C. Larman, *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Second edition. 2001.

[6] Visual Paradigm, *What is user story?*. Retrieved 3 October 2020. [Online]. Available: https://www.visual-paradigm.com/guide/agile-software-development/what-is-user-story/

[7] JSON, *Introducing JSON*. Retrieved 22 October 2020. [Online]. Available: https://www.json.org/json-en.html

[8] D. Bolton, *Definition of Encapsulation in Computer Programming*. 10 February 2019. Retrieved 30 September. [Online]. Available: https://www.thoughtco.com/definition-of-encapsulation-958068