# System design document for MemoryShape

Edenia Isaac
Nils Johnsson
Filip Linde
Kevin Svensson

Date: 2020-10-01
Version: 1

Contents:

**1. Introduction** **3**
1.1 Design goals 3
1.2 Definitions, acronyms and abbreviations 3
**2. System design** **4**
2.1 Overview 4
2.2 Software decomposition 4
2.3 Dependency analysis 5
2.4 Game states 5
2.4 Design patterns 5
2.5 Persistent data management 6
2.6 Encapsulation 6
2.7 Testing 6
2.8 Access control and security 6
**3. References** **7**

# 1. Introduction

In this document we will describe the technical aspects of our application. The goal of the project was to create an easily expandable and testable application. To achieve this the software has been developed in a way that minimises dependencies and coupling between different parts and by encapsulating information in classes.

Memoryshape is a single player game that aims to help the user improve his memory while being entertaining and challenging.

## 1.1 Design goals

Our goal is to create an application that follows an object oriented design. It should have low coupling between the different packages, high cohesion within our classes and packages, and be easily extended. Model dependencies on our graphical library must be avoided.  Our classes should be encapsulated and overall we should achieve a modular design.

## 1.2 Definitions, acronyms and abbreviations

- **Card** - A clickable figure that contains a shape
- **CardSelector** - A display of the next selected card for the player
- **Board** - The playfield with all the clickable cards and a CardSelector
- **MVC** - Model View Controller: A design pattern to structure the code on a large scale.
- **GUI** - Graphical User Interface: The visible part of the program for the user.
- **UML** - Unified Modeling Language: A diagram describing how different parts of the program work together.

- **Domain model** - A diagram describing the parts of the programs model.
- **User Stories** - A small story that describes the type of user, what they want and why.
- **FXML file** - Is used to create the user interface.

# 2. System design

(TODO)
You will to describe the `flow' of the application at a high level. What happens if the application is started (and later stopped) and what the normal flow of operation is. Relate this to the different components (if any) in your application.

-paket.
system deisgn går in i paket, vilka principer, hur vi jobbar objektorienterat helt enkelt

## 2.1 Overview

MemoryShape is written in java and designed with JavaFX. The application is structured in a way that follows MVC pattern. The model is not dependent on neither the view or controller and relevant information for the view is (WILL BE SENT THROUGH) sent through method parameters following Observer Pattern.

## 2.2 Software decomposition

The application consists of 4 packages.

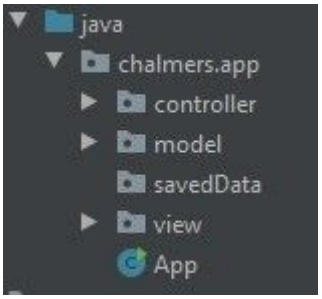

Figure 1: Screenshot of the packages

The controller package contains one controller for each fxml file. There are 3 controllers in total. Each control handles user-interaction with its fxml file and communicates with the model.

The model package contains all the data and the logic of the application. The model does not have any dependencies on external libraries. To make it flexible for future development and easy to use for another group of developers.

SavedData is a package with the data stored between sessions. It has a textfile containing information about the top ten high-scores scored on the machine.

The view package contains the fxml files and images.

## 2.3 Dependency analysis

(TODO)
Figure 1 (lägg till bild) contains all the dependencies of the application. No circular dependencies are (OR WILL BE) present in the latest iteration of the system.

Draw an UML package diagram for the top level for all components that you have identified above (which can be just one if you develop a standalone application). Describe the interfaces and dependencies between the packages. Describe how you have implemented the MVC design pattern.

Create an UML class diagram for every package. One of the packages will contain the model of your application. This will be the design model of your application, describe in detail the relation between your domain model and your design model. There should be a clear and logical relation between the two. Make sure that these models stay in `sync' during the development of your application.

The above describes the static design of your application. It may sometimes be necessary to describe the \emph{dynamic} design of your application as well. You can use an UML \emph{sequence diagram} to show the different parts of your application communicate an in what order.

## 2.4 Game states

Svara på hur användaren byter mellan de olika vyerna. The application uses different views between the menu and the gameplay. When viewing the menu.(TODO)

## 2.4 Design patterns

The application was programmed with intent to follow the MVC-pattern. This gives the code a solid structure on a larger scale, dividing the program into the three main sections. To be able to run the game in different modes strategy pattern (WILL BE ) used. The main function to run the gameplay of the application uses different strategies depending on what gamemode is selected. Other patterns used in the software includes Observer pattern and Facade pattern. Observer pattern is used to send relevant information from the model to the

view. This is an effective way to let the model communicate with other parts without depending on them. Facade pattern is used to increase cohesion within packages while decreasing coupling between them (HCLC). Instead of having many communicating parts within the packages only one class is used to communicate to other packages, lowering dependencies between them.

## 2.5 Persistent data management

The highscores are stored in a text file. The text file contains score, player name and time. It can be read and written to.

The images are stored in their own directory in the view package. They are loaded on application startup and when the player interacts with the cards on the board.

## 2.6 Encapsulation

One important principle of any Object-oriented-applications is encapsulation. Which is great because it protects the state of the object and reduces human errors [1]. We have followed this principle by always setting objects attributes to private. The attributes are modified by the public methods in the object instead. To encapsulate the model from other packages relevant information is always sent as a copy of the original object, so that modifications won't affect the model.

## 2.7 Testing

(TODO)
\begin{itemize}
  \item Describe how you test your application and where to find these tests. If
    applicable, give a link to your continuous integration.
  \item List all known issues.
  \item Run analytical tools on your software and show the results. Use for example:
    \begin{itemize}
      \item Dependencies: \href{http://stan4j.com/}{STAN} or similar.
      \item Quality tool reports, like \href{http://filehippo.com/download_pmd/}{PMD}.
    \end{itemize}
\end{itemize}

NOTE: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by ..., uses ..., etc.

## 2.8 Access control and security

NA, right?

(If you applications has some kind of access control, for example a login, of has different user roles (admin, standard, etc.), then explain how you application manages this.)

## 3. References

[1]:
Encapsulation 20-09-30 from
https://www.thoughtco.com/definition-of-encapsulation-958068

List all references to external tools, platforms, libraries, papers, etc. The purpose is that the reader can find additional information quickly and use this to understand how your application works.