SE 3XA3: Module Guide
Poker Project

Team 12
Safwan Hossain and hossam18
Eamon Earl and earle2
Tyler Magarelli and magarelt

April 12, 2022

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	Version	Notes
March 16, 2022	1.0	Initial Draft
March 18, 2022	1.1	Revised Draft
April 12, 2022	2.0	Revision 1

1 Introduction

1.1 Summary of Project

Our Poker Project is aptly named; we aim to build upon a relatively primitive base code for evaluating poker hand states and create a fully playable online poker experience. Our main motivation behind developing this software is to remove the barriers that other similar products have neglected to in the past, namely the requirement of financial commitment from the user. We firmly believe in the educational and developmental value of a strategic, high-stakes game like poker, but we also believe that losing money and promoting detrimental habits and addiction are not inherent to that high-stakes feeling. Our project will make this game accessible to students and other young people who are not in the financial situation to regularly go to a casino or use a monetary gambling app, and will give them the opportunity to develop valuable risk analysis skills and have fun with their friends, without the chance of harming their future.

1.2 Context of Module Guide

This document underlines the distinctions made using the information gathered from the requirements document, and how they will go on to partition the elements of the software into distinct modules, which are described in detail in the MIS below (Section 8). We have also developed traceability matrices to ensure that we have both encapsulated all of our required behaviour somewhere within the specification of the current architecture, and that our anticipated changes will have a "future home" as well. This document will provide various views on where we are currently at in the development process, where we were and where we intend on going, such that it will have some inherent value to each stakeholder involved. These include:

- Designers: This document will act as an oracle for the design and dependencies of the programs to be written by the development team, showing both what is expected and likely areas of faults or stress, so that the designers can reinforce the importance of those areas and ensure that the developers are implementing them correctly. It also acts as information hiding, so when verifying the final product the design team can simply refer to the overarching hierarchy and behaviour described by the MIS, as opposed to reading through the code or its associated documentation. Additionally, as designing for generality is a specific property that we have prioritized given the possible application of the software to additional card game simulations, and we have already begun the process of considering anticipated changes, it allows the designers to freely consider what areas of the program may have the flexibility to accommodate for these heuristics.

- Developers: The viewpoint of this document that is most applicable to the developers is that of the MIS; the technical specification of what each module and their encapsulated routines must achieve. If the designers are vigilant and the requirements are fully and

unambiguously specified, the developer should not have to consider anything else. Regardless, the abstraction of the architecture is there for their consideration, and interfacing directly with the source code may also give them unique insight into areas for achieving the previously discussed heuristics of generality and modifiability.

- Clients: The clients can use this document as a metric of the degree to which their business requirements are being met, specifically with Sections 4 and 6. They can also validate whether or not any anticipated changes conflict with any of their prior expectations, or if any unlikely changes highlight new behaviour that they may want to reconsider. However, it must be acknowledged that prioritizing a change after it has already been deemed unlikely may inherently cause some code overhaul and some setback in the lifetime of the development.

1.3 Design Principles

As previously discussed, the main principles we are hoping to achieve with our design are generality and modifiability / designing for change. This is based off of the principle that the lifespan of our project is relatively small and consequently so is the scope, and as such there is very likely to be further development on this code in the future with the addition of further features and game modes, just as this project was based and developed on top of some previous source code; such is the nature and the beauty of open source projects. These principles encapsulate other ones, however, which can be seen throughout our design. Modularity is a principle that boils down parts of the functional source code into the smallest, distinct sections, which we have designed to have high internal coupling and low cohesion with other such modules. This allows for a module to be swapped out with another in the future, with minimal effects on the rest of the program. We also designed our program following the MVC (Model-View-Controller) architecture, which inherently partitions the program into three main sections, such that concerns can be separated when working in each distinct region. It also distinguishes areas of the code that are most important to different stakeholders and for different purposes. The data model will be used as a base for the current game mode and all future game modes, while the view is the most important section when considering the user. The controllers act as individual scripts for each game mode, and the Server works to trigger the controllers and link communication between multiple clients. All of these abstracted sections connect with the rest of the system minimally, ensuring that changes to the system can be made easily.

1.4 Outline

The rest of the document is organized as follows. Section 2 lists the anticipated and unlikely changes of the software requirements. Section 3 summarizes the module decomposition that was constructed according to the likely changes. Section 4 specifies the connections between the software requirements and the modules. Section 5 gives a detailed description of the modules. Section 6 includes two traceability matrices. One checks the completeness of the

design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 7 describes the use relation between modules.

2 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 2.1, and unlikely changes are listed in Section 2.2.

2.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: Using a higher fidelity server.

AC2: Implementing the chip betting system with preset chip values (5, 10, 20, 100, etc.)

AC3: The format of the data that is exchanged between the clients and the server.

AC4: The method of how all clients synchronize gameplay.

AC5: The addition of a controller template, that all controller classes will implement, to make running various game modes from the same starting menu easier.

AC6: GameView will be developed to have more useful display functionality, to relieve some complication of scripting gameplay in the controller, improving readability and fixing some cohesion issues.

2.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

UC2: There will always be a source of input data external to the software.

UC3: A server will always be used for every game.

UC4: The model for the deck will not change.

3 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 2. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Hardware-Hiding Module

M2: Model Module

M3: View Module

M4: Controller Module

Level 1	**Level 2**
Hardware-Hiding Module	
Model Module	Client GameInfo Server Game Card Player HandEval Deck PlayerAction
View Module	MainMenuView GameView
Controller Module	MainController ClientController ClientHandler

Table 2: Module Hierarchy

4 Connection Between Requirements and Design

The design for this system is intended to meet all the requirements established in the first SRS document. The system is broken down into modules that will have their own requirements and when the modules are put together, the system will satisfy all of the requirements. Table 1 shows the relationship between needs and modules. 4.

5 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by **?**. The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. Also indicate if the module will be implemented specifically for the software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented. Whether or not this module is implemented depends on the programming language selected.

5.1 Hardware Hiding Modules (M1)

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

5.2 Model Module

Secrets: Decides how elements of game data will stored and updated.

Services: This module will determine the structure of majority of the system. It will consist of all the data elements that will need to be stored in order to run the program and modules that will implement functions that determine how the data can be updated.

Implemented By: –

5.2.1 Client Module (M2)

Secrets: The information and components for a user to interact with the server.

Services: Stores information about the user that is needed by the server to identify and manipulate each unique client.

Implemented By: ClientController

5.2.2 GameInfo Module (M2)

Secrets: Decides how to store game data.

Services: Stores important game data that will be communicated between a server and client.

Implemented By: Client, ClientController, ClientHandler

5.2.3 Server Module (M2)

Secrets: Decides how to store and manage client and server data.

Services: Stores server and client data, allowing users to establish a connection to the server.

Implemented By: MainController

5.2.4 Game Module (M2)

Secrets: Decides to store important elements of the game.

Services: Provides a system to store and manage elements of the game that will keep track of the game's progress.

Implemented By: ClientController

5.2.5 Card Module (M2)

Secrets: Stores information about a card.

Services: Provides a way to store a card suite and rank in a single data type.

Implemented By: Game, Deck

5.2.6 Player Module (M2)

Secrets: Stores information about a player.

Services: Provides a way to store and represent each player of the game.

Implemented By: Game

5.2.7 HandEval Module (M2)

Secrets: Decides the card rankings of each player's hand.

Services: Provides a way to assign each player a hand ranking for comparison.

Implemented By: Game

5.2.8 Deck Module (M2)

Secrets: Stores information about a deck of cards

Services: Provides functions and storage for a collection of cards.

Implemented By: Game

5.2.9 PlayerAction Module (M2)

Secrets: Stores information about valid game moves.

Services: Provides a way to validate user inputs by confirming if their input is a valid game move.

Implemented By: Game, GameInfo, ClientHandler, Client

5.3 View Module

Secrets: Decides how elements of game data will be displayed to the user.

Services: This module will provide the visual aspect of the program, providing modules that will allow the data in the program to be displayed to the user in a meaningful way. Almost all elements of the view module will be easily modifiable with minimal collateral to other code.

Implemented By: –

5.3.1 MainMenuView Module (M3)

Secrets: Decides how the main menu of the program will be displayed.

Services: Provides a way to display and navigate the main menu. Also contains error and success messages.

Implemented By: MainController

5.3.2 GameView Module (M3)

Secrets: Decides how the game will be visually represented to the user.

Services: Provides a way to display all the elements of the game to the user in a meaningful way. Also contains error and success messages.

Implemented By: ClientController

5.4 Controller Module

Secrets: Decides and handles the software decision making process. When and what data will be manipulated by user input.

Services: This module will act as a mediator between the view, model and the user. It will utilize the view and the model to display data to the user and take in user input and decide how the user input will manipulate the data.

Implemented By: –

5.4.1 MainController Module (M4)

Secrets: Decides how to enter and setup the game.

Services: Provides an interface to users on how to set up and join a game.

Implemented By: Hardware-Hiding Module

5.4.2 ClientController Module (M4)

Secrets: Decides how the user can manipulate game data according to a central server consisting of multiple players.

Services: Acts as the "brain" of the program by providing a synchronous way to play the game with other online players.

Implemented By: Hardware-Hiding Module

5.4.3 ClientHandler Module (M4)

Secrets: Decides how the server should respond to user input.

Services: Provides a way for users to communicate with the server behind the scenes

Implemented By: Hardware-Hiding Module

6 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Func. Req.	Modules
FR1	M2, M3
FR2	M2, M3
FR3	M2, M3
FR4	M2, M3, M4
FR5	M2, M4
FR6	M2, M4
FR7	M2
FR8	M2
FR9	M2
FR10	M2, M3, M4
FR11	M2, M3
FR12	M2, M3
FR13	M2, M3, M4
FR14	M2, M3, M4
FR15	M2, M3, M4
FR16	M2, M3, M4
FR17	M2, M4
FR18	M2, M4
FR19	M4
FR20	M2, M3, M4
FR21	M2, M3, M4
FR22	M2, M3, M4
FR23	M2, M3
FR24	M2, M3
FR25	M2, M3, M4
FR26	M2, M3, M4
FR27	M2, M3, M4
FR28	M2, M4
FR29	M2, M4
FR30	M2, M4

Table 3: Trace Between Functional Requirements and Modules

Non-Func. Req.	Modules
NFR1	M1, M3
NFR2	M1
NFR3	M1
NFR4	M1, M4
NFR5	M1
NFR6	M2, M4
NFR7	M1
NFR8	M1
NFR9	M1
NFR10	M1
NFR11	M1
NFR12	M1, M4
NFR13	M1, M4

Table 4: Trace Between Non-Functional Requirements and Modules

AC	Modules
AC1	M1
AC2	M2
AC3	M1
AC4	M1
AC5	M4
AC6	M3, M4

Table 5: Trace Between Anticipated Changes and Modules

7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. A uses relation between module A and module B is defined to be that module A is dependant of module B to function correctly. For example, if module A uses module B then the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules for this program. It can be seen that the graph is a directed acyclic graph. Each level of the hierarchy provides a testable and functional component of the system, and higher-level modules are effectively simpler since they rely on lower-level modules.

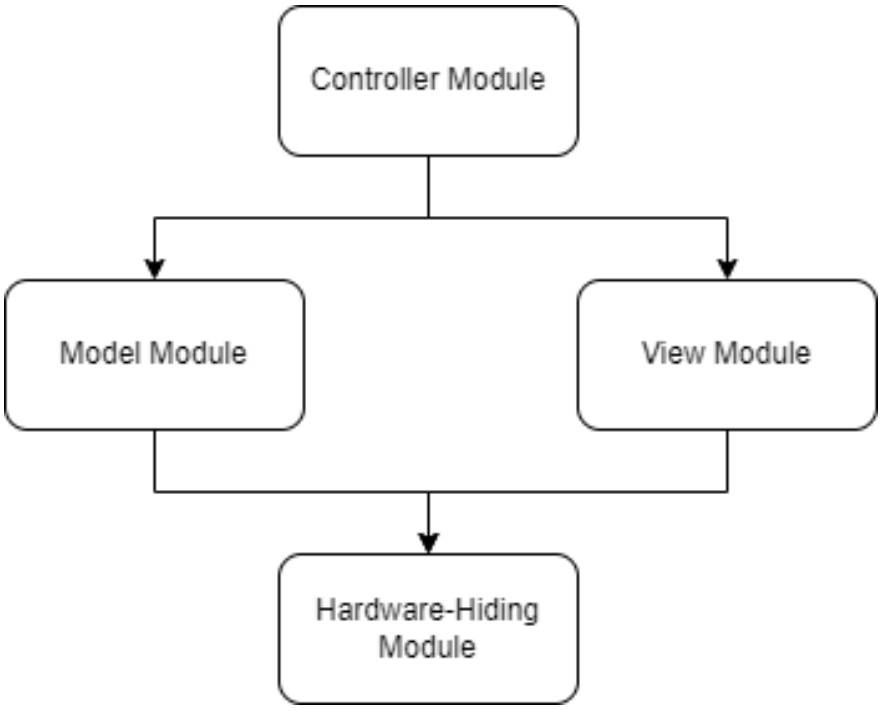

Figure 1: Use hierarchy among modules