Module Guide for Farming Matters

Team #14, The Farmers
Brandon Duong
Andrew Balmakund
Mihail Serafimovski
Mohammad Harun
Namit Chopra

January 18, 2023

1 Revision History

Date	Developer(s)	Change
1/18/2023	Namit Chopra, Brandon Duong Andrew Balmakund, Mohammad Harun Mihail Serafimovski	Finished First Version of Module Guide

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
FR	Functional Requirement
NR	Non-Functional Requirement
LF	Look and Feel Requirement
UH	Usability and Humanity Requirement
PR	Performance Requirement
OE	Operations and Environmental Requirement
MR	Maintainability and Support Requirement
SR	Security Requirement
LR	Legal Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
UI	User Interface
Farming Matters	Final Year Software Engineering Capstone Project name
UC	Unlikely Change

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (**?**). We advocate a decomposition based on the principle of information hiding (**?**). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by **?**, as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 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 9 describes the use relation between modules.

4 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 4.1, and unlikely changes are listed in Section 4.2.

4.1 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: Fluctuating prices for purchasing an item as well.

AC2: Adding more variety to the type of avatars for the user to interact with

AC3: Adding more variety to the type of items

AC4: Mechanism of events to alter the game state

AC5: Mechanism of consultant for providing probabilities of future events

AC6: Multiple soundtracks for different aspects of the game

AC7: Being able to sell from either market or inventory

AC8: The freedom of choosing where purchase insurance of an item

AC9: The algorithm behind of how insurance works

AC10: The placements of UI elements for different aspects of the game

AC11: The method of saving game state

AC12: The number of turns per season

4.2 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: User input in-game controls

UC2: The details need to create an account will remain the same

UC3: The environment in which the user will be able to play the game

UC4: The system will only one login session per user

UC5: The specific data used to collect for research purposes will remain the same

UC6: The ability to purchase and sell items

UC7: The ability to access a consultant

UC8: The authentication of a user is required before allowing them to perform actions in the game

UC9: The ability to delete user data

UC10: The ability to plant and harvest a crop

UC11: Changing of seasons

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Avatar

M2: AvatarMenu

M3: Consultant

M4: OtherAvatar

M5: Item

M6: Inventory

M7: Market

M8: DatabaseOperations

M9: MusicPlayer

M10: GameSettings

M11: AuthState

M12: Socket

M13: ClientFirebase

M14: User

M15: AuthError

M16: CreateAccount

M17: Login

M18: ServerFirebase

M19: RedisClient

M20: Server

M21: Seed

M22: FarmTile

M23: FarmGrid

M24: TurnSummary

M25: GameController

Level 1	Level 2
Hardware-Hiding Module	-
Behaviour-Hiding Module	GameController AvatarMenu Market Inventory FarmGrid GameSettings CreateAccount Login TurnSummary
Software-Hiding Module	Avatar Consultant OtherAvatars FarmTile Seed Item DatabaseOperations ServerFirebase ClientFirebase AuthState Socket RedisClient Server MusicPlayer User AuthError

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

7 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. *Farming Matters* means the module will be implemented by the Farming Matters 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.

7.1 Hardware Hiding Modules

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

7.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

7.2.1 GameController (M25)

Secrets: User input handlers and in-game logic.

Services: Responsible for calling all related submodules throughout a session.

Implemented By: Farming Matters

Type of Module: Library

7.2.2 AvatarMenu (M2)

Secrets: User input handling during menu.

Services: Interact with user selecting several avatars to choose to interact with.

Implemented By: Farming Matters

Type of Module: Library

7.2.3 Market (M7)

Secrets: The data structure of a Market.

Services: Provide functionality to interact with various items in a market, make purchases and updating the inventory.

Implemented By: Farming Matters

Type of Module: Library

7.2.4 Inventory (M6)

Secrets: The data structure of an Inventory.

Services: Provide the ability to get the current state of the inventory, add items to an inventory and remove items from an inventory.

Implemented By: Farming Matters

Type of Module: Library

7.2.5 FarmGrid (M23)

Secrets: The data structure of a Farm Grid.

Services: Provide the ability to get the current state of the grid tiles and add tiles to the current grid tiles.

Implemented By: Farming Matters

Type of Module: Library

7.2.6 GameSettings (M10)

Secrets: User adjustable setting values.

Services: The ability to change volume and opt out of study.

Implemented By: Farming Matters

Type of Module: Library

7.2.7 CreateAccount (M16)

Secrets: User account details.

Services: Provides functionality for registering an account to create the game and handling invalid user inputs.

Implemented By: Farming Matters

Type of Module: Library

7.2.8 Login (M17)

Secrets: User account details.

Services: Provides functionality for logging into the system and handling invalid user inputs.

Implemented By: Farming Matters

Type of Module: Library

7.2.9 TurnSummary (M24)

Secrets: Users decisions made in a single turn.

Services: Provides functionality for logging into the system and handling invalid user inputs.

Implemented By: Farming Matters

Type of Module: Library

7.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: –

7.3.1 Avatar (M1)

Secrets: The data structure of an Avatar.

Services: Provide functionality to get and set various properties of an Avatar.

Implemented By: Farming Matters

Type of Module: Abstract Object

7.3.2 Consultant (M3)

Secrets: Avatar statements.

Services: Provides functionality to generate a statement that assesses the current state of the game and provide a statistic based on a type of decision (probabilistic or deterministic).

Implemented By: Farming Matters

Type of Module: Record

7.3.3 OtherAvatars (M4)

Secrets: Avatar statements.

Services: Provides functionality to generate a statement related to a specific other avatar.

Implemented By: Farming Matters

Type of Module: Record

7.3.4 FarmTile (M22)

Secrets: Data structure of a farm tile.

Services: Provides functionality to plant a seed, harvest a crop, determine what has been planted on a given tile and how long a seed has been growing for.

Implemented By: Farming Matters

Type of Module: Abstract Object

7.3.5 Seed (M21)

Secrets: Data structure of a seed.

Services: Provides functionality to determine the seed growth length, the seasons in which the seed can grow and the sellable price ranges .

Implemented By: Farming Matters

Type of Module: Abstract Object

7.3.6 Item (M21)

Secrets: Data structure of an item.

Services: Provides functionality to get and set the price of an item, get the name of an item, get the quantity of an item and get the type of the item.

Implemented By: Farming Matters

Type of Module: Abstract Object

7.3.7 DatabaseOperations (M8)

Secrets: Establishing database connection.

Services: Provides functionality for terminating the database connection, adding and removing users from database, logging a given data, saving and loading the current state of the game.

Implemented By: Farming Matters

Type of Module: Library

7.3.8 ServerFirebase (M18)

Secrets: Firebase API credentials.

Services: Connects the server to a Firebase API.

Implemented By: Farming Matters

Type of Module: Library

7.3.9 ClientFirebase (M13)

Secrets: Firebase API credentials.

Services: Connects the client to a Firebase API.

Implemented By: Farming Matters

Type of Module: Library

7.3.10 AuthState (M11)

Secrets: User session details.

Services: Validates if a user is authenicated session and handles unauthorized sessions.

Implemented By: Farming Matters

Type of Module: Library

7.3.11 Socket (M12)

Secrets: User session details.

Services: Grants the user access to play the game.

Implemented By: Farming Matters

Type of Module: Library

7.3.12 RedisClient (M19)

Secrets: API credentials needed to access Firebase instance.

Services: Handles invalid credentials.

Implemented By: Farming Matters

Type of Module: Library

7.3.13 Server (M20)

Secrets: Internal routing and database setup.

Services: Provides communication protocols to connect the front-end and the back-end.

Implemented By: Farming Matters

Type of Module: Library

7.3.14 MusicPlayer (M9)

Secrets: Audio files.

Services: Provides the ability to load, start and stop playing a audio file.

Implemented By: Farming Matters

Type of Module: Library

7.3.15 User (M14)

Secrets: User credentials stored in Firebase system.

Services: Retrieve user information.

Implemented By: Firebase

Type of Module: Record

7.3.16 AuthError (M15)

Secrets: Authentication specific details for a user.

Services: Raises in error for unverified users.

Implemented By: Firebase

Type of Module: Library

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes. Here is the Software Requirements Specification document to find specific details to the corresponding requirements listed below in Table 2.

Req.	Modules
FR1	M16, M11, M17, M8
FR2	M17, M8
FR3	M7, M6
FR4	M6, M5
FR5	M11, M16, M17
FR6	M7, M6, M5
FR7	M23, M22, M21, M5, M6
FR8	M7, M6, M5
FR9	M6, M5, M23, M22, M21
FR10	M7, M23, M22
FR11	M2, M1, M3, M25
FR12	M7, M5, M6
FR13	M8, M25, M18, M20
FR14	M8, M25, M18, M20
FR15	M8, M25, M18, M13, M20
FR16	M23, M22
FR17	M23, M22, M6, M5
FR18	M17
FR19	M6, M23, M25
FR20	M16, M8, M25
FR21	M25
FR22	M25
LF1	M7, M6, M2, M10
LF2	M7, M6, M2, M10
LF3	M9
LF4	M7, M6, M2, M23, M22, M3, M4, M5
LF5	M25
LF6	M7, M6, M2, M23, M22, M3, M4, M5, M25
UH1	M7, M6, M2, M10, M25
PR1	M8, M20, M25
PR2	M8, M20, M25
OE1	M17, M25
OE2	M17, M25
MR1	M9, M5, M2
SR1	M17, M16
LR1	M8, M10

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M7, M5
AC2	M2, M4
AC3	M5
AC4	M25, M23, M22, M3
AC5	M3, M25
AC6	M9
AC7	M7, M6
AC8	M7, M6, M25, M5
AC9	M7,M5
AC10	M7, M6, M2, M10
AC11	M8, M25
AC12	M25

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

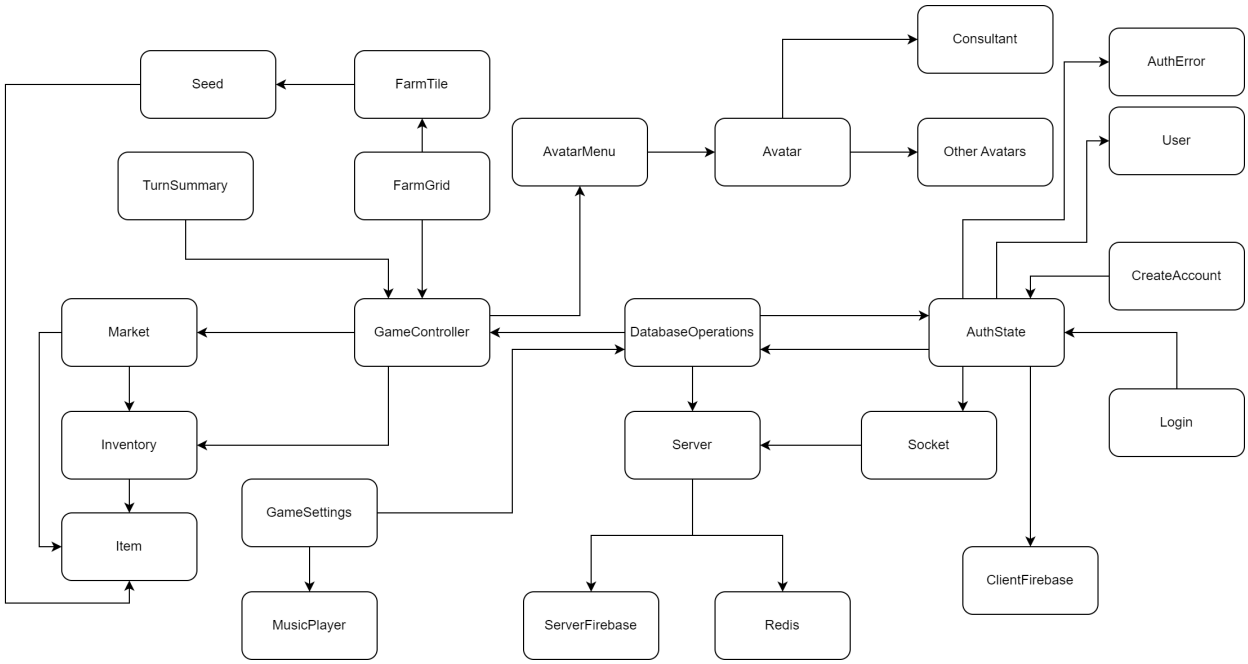

Figure 1: Use hierarchy among modules