Module Guide for SFWRENG 4G06: Dice Duels: Duel of the Eights

Team 9, dice_devs
John Popovici
Nigel Moses
Naishan Guo
Hemraj Bhatt
Isaac Giles

January 14, 2025

1 Revision History

Table 1: Revision History

Date	Developer(s)	Change
2025-01-08	Hemraj Bhatt	Added content to sections 4.1 & 4.2
2025-01-11	John Popovici	Formatted section 2; Added content to sections 4.1 & 4.2
2025-01-11	Hemraj Bhatt	Updated content on section 4
2025-01-14	Hemraj Bhatt	Updated content on section 3

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
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
SFWRENG 4G06	Explanation of program name
UC	Unlikely Change
[etc. —SS]	[... —SS]

See SRS Documentation for any additional.

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 (Parnas et al., 1984). For the Dual of the Eights project, we advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, as the ”secrets” that each module hides represent likely future changes. This approach is particularly valuable in the context of Dual of the Eights, where modifications to game rules, mechanics, or visual elements are likely during development and playtesting phases.

Our design follows the guidelines laid out by Parnas et al. (1984), as follows:

- System details that are likely to change independently, such as scoring algorithms, dice physics, or player health mechanics, should be the secrets of separate modules.
- Each data structure, such as those representing dice, players, or game states, is implemented in only one module.
- Any other module requiring 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.

*Template introduction refactored. Mostly kept the same, with few wording changes.

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: Operating System: Godot makes porting the game to other operating systems simpler, but specific interfaces may have to change and testing would be required.

AC2: Hardware: The specific hardware on which the software runs is expected to evolve, particularly as the online multiplayer functionality will require a server to enable long-distance gameplay. This aspect of the project is subject to change based on cost, performance, and scalability requirements.

AC3: User Input: The format of the initial input data is expected to evolve to accommodate users in operating the game and setting game rules. These changes will be made to minimize user errors and facilitate a smoother gameplay experience.

AC4: File Type: The file structure for saving game states and related data is expected to evolve. A file structure that ensures efficient storage and facilitates quick retrieval of game states is expected to be used.

AC5: Scoring: The scoring calculations may be modified based on usability and player testing.

AC6: User Interface (UI): A rudimentary UI is currently used for development. The UI will need to be updated to accommodate the addition of animations and will have to be refreshed in order to look more appealing once the game's major features are complete.

AC7: Dice: Dice types can continually be added and modified as they use a simple interface and if added are simple to integrate within the larger system.

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: Input Devices: The game is designed to play on a computer, i.e. it is designed to work with a keyboard and mouse. It is unlikely that the game will be playable through other input means such as controllers.

UC2: Output Devices: The game is designed to play on a computer, i.e. it is designed to display correctly on a computer screen. It is unlikely that the game will be playable through other devices who have vastly different screen sizes such as phones.

UC3: Game Types: Despite being likely changes in the SRS stage of development, once formed, the game types designed would require massive module changes if they were to further be modified.

5 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

...

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	? ? ? ? ? ? ? ?
Software Decision Module	? ? ?

Table 2: Module Hierarchy

This is the draft answer for this section:

The modules are categorized into three types: Hardware-Hiding, Behavior-Hiding, and Software Decision modules. Below is the hierarchy:

5.1 Hardware-Hiding Modules

- **CustomBaseDie Module:** Handles the 3D dice models, textures, and physics.
- **DynamicDiceContainer Module:** Manages dice interactions and rendering.
- **NetworkManager2P Module:** Manages connection and synchronization for two-player games.

5.2 Behavior-Hiding Modules

- **MultiGameManager Module:** Manages the sequence of multiple Yahtzee games and customization phases.
- **PlayerManager Module:** Tracks player states, scores, and upgrades.
- **GameManager Module:** Handles a single game of Yahtzee.
- **CustomizationMenu Module:** Implements dice and game customization between games.
- **DynamicScoreboard Module:** Tracks and displays scores dynamically.

5.3 Software Decision Modules

- **ScoreCalculator Module:** Calculates scores for dice rolls based on Yahtzee rules and custom modifiers.
- **GameSettings Module:** Loads and stores settings for this Yahtzee variant.
- **GameUI Module:** Provides the interface for interacting with the game and displaying relevant data.

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 3.

[The intention of this section is to document decisions that are made "between" the requirements and the design. To satisfy some requirements, design decisions need to be made. Rather than make these decisions implicit, they are explicitly recorded here. For instance, if a program has security requirements, a specific design decision may be made to satisfy those requirements with a password. —SS]

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). 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. *SFWRENG 4G06* means the module will be implemented by the SFWRENG 4G06 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 (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

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 Input Format Module (M??)

Secrets: The format and structure of the input data.

Services: Converts the input data into the data structure used by the input parameters module.

Implemented By: [Your Program Name Here]

Type of Module: [Record, Library, Abstract Object, or Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.2 Etc.

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 Etc.

7.4 Hardware-Hiding Modules

7.4.1 CustomBaseDie Module

Secrets: The interaction logic and rendering of custom dice, including dynamic face changes.
Services: Manages dice texture changes based on player customization and handles physics and collision for rolling.
Implemented By: SFWRENG 4G06.

7.4.2 DynamicDiceContainer Module

Secrets: The algorithms for managing a collection of dice and their rendering.
Services: Dynamically creates and organizes dice based on game settings and updates dice states during gameplay.
Implemented By: SFWRENG 4G06.

7.4.3 NetworkManager2P Module

Secrets: The underlying implementation of peer-to-peer communication and synchronization between two clients.
Services: Handles connection setup, data synchronization, and disconnection handling for a 2-player game.
Implemented By: SFWRENG 4G06.

7.5 Behavior-Hiding Modules

7.5.1 MultiGameManager Module

Secrets: The logic for managing multiple games, including customization and upgrades between games.
Services: Tracks progress and integrates player upgrades into gameplay.
Implemented By: SFWRENG 4G06.

7.5.2 PlayerManager Module

Secrets: The data structure for tracking player-specific details, including scores, dice, and upgrades.
Services: Tracks and updates player states, including consumables and modifiers.
Implemented By: SFWRENG 4G06.

7.5.3 CustomizationMenu Module

Secrets: The logic for presenting and applying customization options between games.
Services: Displays upgrade options, enforces selection rules, and updates player dice and

modifiers.
Implemented By: SFWRENG 4G06.

7.5.4 DynamicScoreboard Module

Secrets: The algorithms for dynamically generating score displays.
Services: Updates and displays scores in real-time during gameplay.
Implemented By: SFWRENG 4G06.

7.6 Software Decision Modules

7.6.1 ScoreCalculator Module

Secrets: The scoring algorithms based on Yahtzee rules and custom modifiers.
Services: Calculates scores for dice rolls and applies passive modifiers.
Implemented By: SFWRENG 4G06.

7.6.2 GameSettings Module

Secrets: Configuration storage and retrieval.
Services: Stores and loads game settings, including the number of games and customization options.
Implemented By: SFWRENG 4G06.

7.6.3 GameUI Module

Secrets: The logic for UI components and interactions.
Services: Displays information to players, including scores and customization options, and provides action buttons.
Implemented By: SFWRENG 4G06.

8 Traceability Matrix

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

Req.	**Modules**
R1	M1, M??, M??, M??
R2	M??, M??
R3	M??
R4	M??, M??
R5	M??, M??, M??, M??, M??, M??
R6	M??, M??, M??, M??, M??, M??
R7	M??, M??, M??, M??, M??
R8	M??, M??, M??, M??, M??
R9	M??
R10	M??, M??, M??
R11	M??, M??, M??, M??

Table 3: Trace Between Requirements and Modules

AC	**Modules**
AC2	M1
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 4: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) 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.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

Figure 1: Use hierarchy among modules

10 User Interfaces

[Design of user interface for software and hardware. Attach an appendix if needed. Drawings, Sketches, Figma —SS]

11 Design of Communication Protocols

[If appropriate —SS]

12 Timeline

[Schedule of tasks and who is responsible —SS]

[You can point to GitHub if this information is included there —SS]

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.