SE 3XA3: Module Guide
Rogue Reborn

Group #6, Team Rogue++

Ian Prins	prinsij
Mikhail Andrenkov	andrem5
Or Almog	almogo

Due Sunday, November 13th, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	Version	Notes
11/01/16	0.1	Initial Template
11/10/16	0.2	Added Anticipated Changes
11/12/16	0.3	Added Module Decomposition
11/12/16	0.4	Added Introduction
11/12/16	0.5	Added Use Hierarchy
11/13/16	0.6	Proofread and Editing
12/06/16	1.0	Revision 1 Update

1 Introduction

1.1 Project Overview

The Rogue Reborn project aims to develop a contemporary clone of the original *Rogue* game by leveraging modern programming paradigms, development tools, and software engineering principles. Aside from the executable, the Rogue++ team also plans to deliver a collection of commented source code, test suites, and formal documentation for both internal and public consumption. In doing so, Rogue Reborn will serve as an open-source repository for other developers to view, comment, and contribute to the code and documentation that characterize the project in its entirety.

1.2 Document Context

The primary purpose of the Module Guide (MG) is to describe, justify, and contextualize the module decomposition of the system. From a developer perspective, the MG facilitates the detection of overused or purposeless modules and helps identify architectural flaws. The document also provides a conceptual view of each module and its context in the broader system.

Before the MG is written, the Software Requirements Specification (SRS) document should be completed and thoroughly reviewed by all stakeholder parties. The SRS describes requirements for the software system that the MG should relate to the implemented design. By completing the SRS beforehand, it is more likely that the selected system design will comply with all compulsory requirements and the chosen module decomposition is appropriately structured.

After the MG is complete, the complementary Module Interface Specification (MIS) can be conceived to refine the interface (external behaviour) of each module. The MG produces a functional overview of each module, while the MIS delves into the actual implementation of the modules. Together, the MG and MIS are sufficient to allow programmers to concurrently develop the system in a modular fashion.

1.3 Design Principles

In an effort to develop code that is robust, maintainable, and more easily verifiable, the Rogue Reborn project attempts to enforce several well-regard design principles. In particular, the principles of Information Hiding (IH), High Cohesion (HC), and Low Coupling (LC) are consistently expressed throughout the code.

Information Hiding refers to the practice of concealing the implementation of modules; this is typically accomplished through encapsulation. Using this approach encourages developers to become more mindful of their assumptions and to produce code that does not depend on the specific implementations of various functions. Next, High Cohesion and Low Coupling describe the practice of partitioning code into modules that contain related operations and do not rely heavily on other modules. Modules that exhibit HC lend themselves to excellent error localization, while module systems that display LC are better suited for testing purposes. Structuring programs to emphasizes adaptability and modularity is vital to the success of any long-term software project.

1.4 Design Document Structure

A brief description of each section in the Module Guide is given below:

§1 Overview of the MG document

§2 Comprehensive list of system changes under consideration

§3 Conceptual hierarchy of implemented modules

§4 System design rationale with respect to the mandated requirements

§5 Detailed summary of the module decomposition

§6 Tabular depiction of the relationships between modules, requirements, and anticipated changes

§7 Illustration of the module functionality dependency graph

§8 References to external sources

1.5 Acronyms and Definitions

Table 2: **Acronyms**

Acronym	**Definition**
AC	Anticipated Change
ASCII	American Standard Code for Information Interchange
DAG	Directed Acyclic Graph
FR	Functional Requirement
GUI	Graphical User Interface
HC	High Cohesion
HP	Health Points
IH	Information Hiding
LC	Low Coupling
LoS	Line of Sight
MG	Module Guide
MIS	Module Interface Specification
NFR	Non-Functional Requirement
OMOS	One Module One Secret
OS	Operating System
PoC	Proof of Concept
PRNG	Pseudorandom Number Generator
SRS	Software Requirements Specification
UC	Unlikely Change

Table 3: **Definitions**

Term	Definition
Amulet of Yendor	Item located on the deepest level of the dungeon; enables the player character to ascend back through the levels and complete the game
Frame	An instantaneous "snapshot" of the GUI screen
Health Points	Quantity that represents the life of an entity; an entity with no life is considered to be dead.
Libtcod	Graphics library specializing in delivering an authentic roguelike experience
Permadeath	Feature whereby the death of the player character will conclude the game
Player Character	User-controlled character in Rogue Reborn
Rogue	Video game developed for the UNIX terminal in 1980 that initiated the roguelike genre
Roguelike	Genre of video games characterized by ASCII graphics, procedurally-generated levels, and permadeath

2 Anticipated and Unlikely Changes

2.1 Anticipated Changes

Anticipated changes (AC) are the source of the information to be hidden inside modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision.

AC1: The specific hardware on which the software is running.

AC2: The operating system on which the program will be executed.

AC3: The language presented in the GUI.

AC4: The accessibility modifications used to play the game.

AC5: The available items in the game (e.g., addition of a new potion).

AC6: The number, type, and visual representation of monsters.

AC7: The graphical overlay of the game (different graphical overlays could be used to alter the displayed images).

2.2 Unlikely Changes

Unlikely changes (UC) are design decisions that are deeply rooted in the system architecture and will require significant effort to modify. As such, it is not intended that the following decision will be changed.

UC1: The structure of the primary game loop in the context of the in-game entities, state parameters, and high-level flow. Virtually every single module in the program will have to be altered to accommodate this change.

UC2: The types of terrain in the game, expressed in terms of passability, transparency, and visibility. A change here will necessitate mutations in the Level, LevelGen, and PlayerChar code.

UC3: The definition of the coordinate system in the game. Given the degree of code that assumes this system behaves in a particular way, modifying this structure may require plentiful module changes.

UC4: The number of rooms in a dungeon level. Altering this parameter will mandate the need for changes to the Level module, the entire Item chain, and, of course, the LevelGen module.

3 Module Hierarchy

The modules listed below are categorized by secret type and serve as leaves in the hierarchy tree; they will realize a concrete implementation in the Rogue Reborn code.

Table 4: **Module Hierarchy**

Level 1	**Level 2**	
Hardware-Hiding Module	BasicIO	**M1**
	Doryen	**M2**
	Input Format	**M3**
Behaviour-Hiding Module	External	**M4**
	Item	**M5**
	Level	**M6**
	LevelGen	**M7**
	MainMenu	**M8**
	Mob	**M9**
	Monster	**M10**
	PlayerChar	**M11**
	RipScreen	**M12**
	PlayState	**M13**
	SaveScreen	**M14**
	UIState	**M15**
Software Decision Module	Coord	**M16**
	Feature	**M17**
	ItemZone	**M18**
	MasterController	**M19**
	Random	**M20**
	Terrain	**M21**

4 Connection Between Requirements and Design

The relationship between system design and requirements is delicate: while every module may contain one secret, this secret may involve multiple functional and non-functional requirements. The decisions that guided the design of the system were influenced by two major factors:

1. **Flexibility:** If any feature is deemed non-feasible, out-of-scope, or otherwise unattainable, the system should continue to function around this defect.

2. **Logical Partitioning:** Related logic should also be structurally proximate. Although it can be argued that this approach was motivated by the "one module, one secret" (OMOS) principle, it is actually a consequence of the duality of the high cohesion (HC) and low coupling (LC) principles.

Since the mandated requirements are numerous, identifying the correspondence of each requirement is likely to dilute the ensuing discussion. As such, only a handful of the more interestingly interactions are described below.

Functional requirement (FR) 11 specifies, "The player character shall be able to pass their turn." On a related note, FR 35 states, "Each monster shall be able to calculate a plan of action during their turn." If neither of these requirements were mandated, a simple queue could be implemented to keep track of entity turn priority. However, given that different actions should occupy different amounts of time, it would a poor design decision to assume that performing a task such as moving to an adjacent tile would result in the same turn delay as reading a scroll. As a result, a more robust implementation was constructed to factor the priorities of different actions and enable the turn-skipping feature. Although this introduced complexity to the code, the requirement was not challenged because players can strategically skip their turns to draw nearby monsters into a more favourable location.

Next, FR 37 comments, "The player character shall be able to defeat every monster." It may be assumed, at this point, that a monster is defeated

if its HP drops below zero. For this FR, the entire functionality of absorbing player damage and reporting the results is handled by the Mob module. This is deliberately excluded from the Monster module because the Monster module is designed to customize monster behavior, such as attack patterns and combative positioning. On the other hand, the Mob module offers an interface and implementation for movement and entity statistics such as HP.

Finally, FR 31 is examined: "Scrolls, rings, and wands shall be usable." This FR represents one of the most difficult aspects of the game; every type of scroll, ring, and wand must provide a different activation effect. While a purple wand may zap a goblin out of existence, a topaz ring might cause the player to suddenly levitate above the ground. Clearly, the sheer variety and sophistication of these effects poses an interesting decomposition challenge. As such, the logical representation of these items is divided into submodules (of the Item module) such as "Wand", "Weapon", and "Potion". However, given the similarity of their activation sequences, each of these submodules implements an interface from the Item module to enable external modules to use items without needing to consider their functional diversity.

5 Module Decomposition

5.1 Hardware Hiding Modules

Name: BasicIO (M1)

Secrets: Input and output devices

Services: Serves as a layer beneath the Rogue Reborn application that hides the hardware input and output devices such as keystrokes and the monitor. The application uses this module to retrieve commands from the user and display the game state.

Implemented By: *Libtcod*

Name: Doryen (M2)

Secrets: Details of display device and OS window interface

Services: Provides a virtual console interface for the application. The virtual console is displayed as a window in the host OS; Rogue Reborn uses this interface to display the game state.

Implemented By: *Libtcod*

Name: Input Format (M3)

Secrets: Interface to input devices

Services: Detects keystrokes and provides virtual keycodes to the application that can later be interpreted by the Rogue Reborn system as commands from the user. Conceptually, this module is a layer above the BasicIO module. In theory, this module could enable the use of a touch screen with virtual buttons as system inputs.

Implemented By: *Libtcod*

5.2 Behaviour-Hiding Modules

Name: External (M4)

Secrets: Method that translates the gamestate into a display

Services: Controls the rendering of the game world; all externally-visible behaviours pass through this module. This module also interfaces with the hardware hiding modules through the rendering process as well as the software decision modules by querying their data structures.

Implemented By: uistate.h

Name: Item (M5)

Secrets: Item data structures and behaviour

Services: Provides a consistent interface across all items, thereby shielding the application from the details of the internal representations. The broader system is able to move, activate, destroy, identify, and generally manipulate items through using this module.

Implemented By: item.h

Name: Level (M6)

Secrets: Dungeon level data structure

Services: Shields the rest of the application from the level data structure, providing methods that conveniently implement the details of operations. This also allows the underlying structures to change freely.

Implemented By: level.h

Name: LevelGen (M7)

Secrets: Level generation algorithm

Services: Generate levels without requiring the controller modules to know the details of the algorithm

Implemented By: level.h

Name: MainMenu (M8)

Secrets: Valid user name enforcement

Services: Accepts a legal player character name from the keyboard and initializes the player character entity

Implemented By: mainmenu.h

Name: Mob (M9)

Secrets: Internal data structure and behavior of creatures.

Services: Serves as the base module for all creatures in the dungeon (including the player character). This module provides interfaces for movement, combat, and various statistics, and also includes several implementations of generic actions.

Implemented By: mob.h

Name: Monster (M10)

Secrets: Monster type data structure and behaviour implementation

Services: Customizes the behaviour of various monsters act aggressively towards players, regenerate health, seek gold, etc. This module also stores the data to define various monster templates (e.g., all dragon share a common behavior pattern, name, HP quantity, etc.)

Implemented By: monster.h

Name: PlayerChar (M11)

Secrets: Player character data structure and action implementation

Services: Represents most aspects of the player character, including their inventory, equipped items, actions, and some obscure statistics.

Implemented By: playerchar.h

Name: PlayState (M12)

Secrets: Player character action selection

Services: Calls the appropriate PlayerChar action methods according to the user's input. This module also renders the dungeon-view game screen.

Implemented By: playstate.h

Name: RipScreen (M13)

Secrets: Score computation and storage

Services: Handles reading and writing from the score record file; the nature and location of the score table is only known to this module.

Implemented By: ripscreen.h

Name: SaveScreen (M14)

Secrets: Game state serialization and deserialization

Services: Handles reading and writing to the saved game files.

Implemented By: savescreen.h

Name: UIState (M15)

Secrets: State of the game interface

Services: Layer between the game user and the game world. This module interprets user command, handles menu control, and passes user commands to the corresponding modules.

Implemented By: uistate.h

5.3 Software Decision Modules

Name: Coord (M16)

Secrets: Coordinate representation and location-related behavior

Services: Provides a consistent interface for all other modules to use to communicate about the locations of objects within the level and on the screen. This module also implements related algorithms such as the taxicab (Manhattan) distance between two coordinates (Margherita Barile).

Implemented By: coord.h

Name: Feature (M17)

Secrets: Data structure that supports various dungeon entities.

Services: Stores the data for any dungeon object that is not a derivative of Mob; this includes stairs, piles of gold, items, and traps.

Implemented By: feature.h

Name: ItemZone (M18)

Secrets: Data structure storing items.

Services: Stores items and provides an interface that maps key-codes to specific items. This module also shields the application from the key-code intricacies and the item stacking phenomenon.

Implemented By: itemzone.h

Name: MasterController (M19)

Secrets: Game loop and application context

Services: Directs the high-level flow of rendering, input-handling, and transitions between lower-level controllers.

Implemented By: mastercontroller.h

Name: Random (M20)

Secrets: Random number generation details

Services: Generates random numbers as a utility service for the remainder of the application. The algorithm used and state of the PRNG are not exposed.

Implemented By: random.h

Name: BasicIO (M21)

Secrets: Data structure of dungeon tiles

Services: Stores the data for a single dungeon tile, including its symbolic character, passability, transparency, and whether or not it is mapped.

Implemented By: terrain.h

6 Traceability Matrices

Two traceability matrices are featured below: one between the modules and the requirements, and another between the modules and the anticipated changes.

Table 5: **Trace Between Requirements and Modules**

Requirements	**Modules**
FR 1, FR 4, FR 7	MasterController, Coord
FR 2, FR 3	RipScreen, SaveScreen
FR 5	MainMenu
FR 6, FR 8, FR 9, FR 10	UIState, Doryen, Coord, PlayState
FR 11, FR 12	PlayerChar
FR 13, FR 14, FR 15	Feature
FR 16	MasterController
FR 17, FR 18, FR 19, FR 25	LevelGen, Coord
FR 20, FR 21, FR 22, FR 24	Level, Coord
FR 23	PlayerChar
FR 26, FR 27, FR 28, FR 29	PlayerChar
FR 30, FR 31	Item
FR 32, FR 33, FR 34	Item, PlayerChar
Fr 35, FR 36	Monster
FR 37	Mob
FR 38, FR 39	PlayerChar
NFR 1	Doryen, UIState, PlayState
NFR 2	Fun
NFR 3	BasicIO
NFR 4, NFR 5	UIState, Doryen, PlayState
NFR 7	MasterController
NFR 8, NFR 13	RipScreen
NFR 9, NFR 12, NFR 15	External

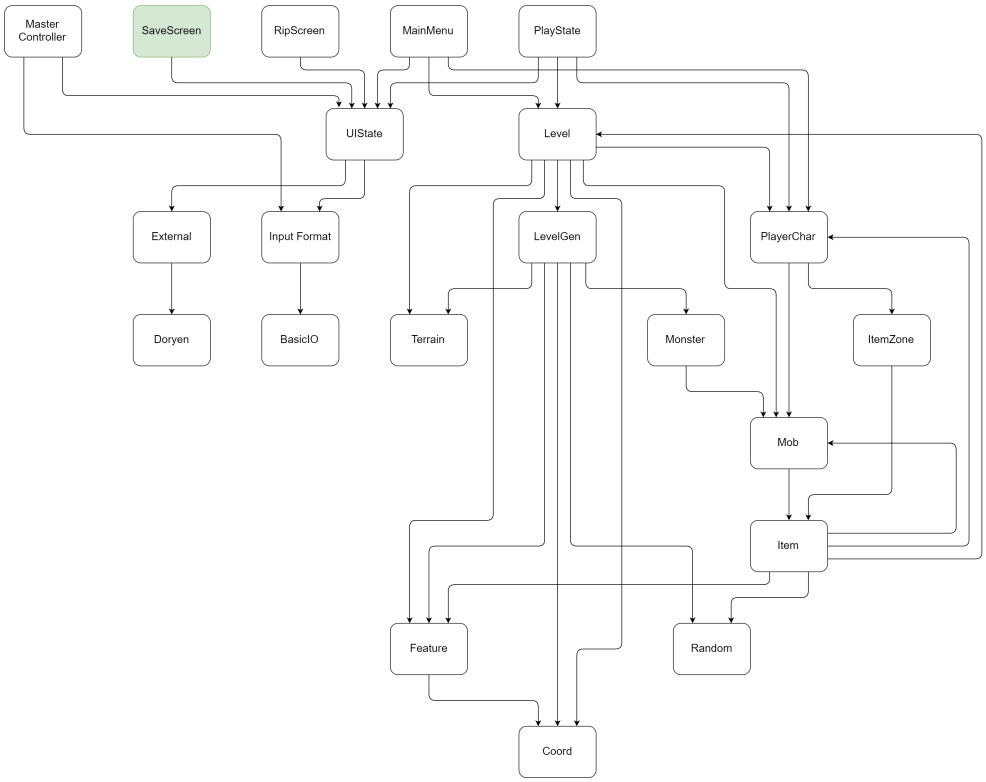
Table 6: **Trace Between Anticipated Changes and Modules**

AC	**Modules**
AC1	BasicIO, External, Random
AC2	BasicIO, External, Doryen
AC3	BasicIO, UIState
AC4	UIState, BasicIO
AC5	Item, Feature, PlayerChar, Mob
AC6	Monster, Mob, UIState
AC7	UIState, External

7 Use Hierarchy Between Modules

Within the software engineering discipline, there exists a widely-accepted belief that all Use Hierarchies must be homomorphic to directed acyclic graphs (DAGs) in order to avoid cyclic dependencies. However, throughout the development of the Rogue Reborn project, the Rogue++ team discovered that a cyclic use relationship across certain modules enabled an architecture that closely resembled the natural flow of the intended interaction. Although it would have been possible to correct the situation, the alternative designs would have an adverse effect on the high cohesion (HC) design principle and would sacrifice the maintainability and readability of the code. As such, the Use Hierarchy drawn in Figure 1 is illustrated with a cyclic dependency stemming from the Item module.

Figure 1: **Use Hierarchy**



8 References

References

Margherita Barile. Taxicab Metric. http://mathworld.wolfram.com/TaxicabMetric.html. Accessed: November 12, 2016.