SE 3XA3: Module Guide
Rogue Reborn

Group #6, Team Rogue++

Ian Prins	prinsij
Mikhail Andrenkov	andrem5
Or Almog	almogo

Due Friday, November 11th, 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

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 Analysis of the system architecture design 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
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 through the levels and complete the game
Encapsulation	Grouping together of methods and fields
Frame	An instantaneous “snapshot” of the GUI screen
Homomorphism	A mapping between two mathematical structures
Libtcod	Graphics library specializing in delivering a 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

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: The specific hardware on which the software is running.

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

AC3: The language in which the game is presented.

AC4: Accessibility modifications used to play the game.

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

AC6: The number, type, and visualization of monsters.

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

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: The way the game loop handles itself, in the context of in-game entities, state parameters, and high-level flow. Just about every single module in the program will have to change to accommodate for this change.

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

UC3: The way in which the coordinate system of the game functions. A change here will involve changing just about every single module in the system.

UC4: The number of rooms in a dungeon level. Modifying this will involve immersive changes to the Level module, as well as the entire Item chain, and of course LevelGen module.

3 Module Hierarchy

Modules are summarized in a hierarchy decomposed by secrets in Table 4. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

Table 4: **Module Hierarchy**

Level 1	**Level 2**
Hardware-Hiding Module	BasicIO Doryen
Behaviour-Hiding Module	External Item Level LevelGen MainMenu Monster PlayerChar RipScreen PlayState UIState
Software Decision Module	Coord Feature Itemzone MasterController Mob Random Terrain

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/non-functional requirements. The decisions that went into the design of the system were made based on two major factors:

- Flexibility: Should any feature be decided as non-feasible, out-of-scope, or otherwise unattainable, the system should not collapse due to its lacking.
- Logical partitioning: We tried to keep related logic as close together as possible. One could argue that this was done to adhere with the concept of "one module, one secret", but that in fact is not the case. The OMOS concept is simply a consequence of the duality of "High cohesion, low coupling". This was, succinctly, the deciding factor in our system design stage.

The requirements are numerous, and identifying each requirements' correspondence to a module would be quite the bore. Instead, we will look at important requirements and how they interact interestingly with their corresponding modules.

Functional requirement 11 specifies: "The player character shall be able to pass their turn". Had this not been the case, a simple queue could be built to keep track of entity move order and that will be all. But this is not the case, and in fact this also ties along with functional requirement 35, which specifies "Each monster shall be able to calculate a plan of action during their turn." Different actions take different amounts of time, and it would be a poor design decision to assume the opposite. A smart, divisive system was built to take this into account and enable turn-skipping. A primary motive that is enabled by this feature is allowing the player to skip their turn, allowing a nearby monster to make a move, hopefully into the region into which the player can fire off projectiles, like arrows and vials. The module that handles all of this functionality is titled *MasterController*.

Functional requirement 37 specifies: "The player character shall be able to defeat every monster." It may be assumed, at this point, that a monster may be defeated if its health drops below zero. The entire functionality of absorbing player damage and reporting the results is handled by the *Mob* module. Why not *Monster*, you ask? Well, the Monster module is designed to customize monster behavior, such as attack patterns, and combative positioning. The Mob module, on the other hand, provides an interface for movement and entity statistics, which includes hit points.

Finally, we will look at functional requirement 31, which specifies: "Scrolls, rings, and wans shall be usable". This requirement represents one of the most difficult aspects of the game. Every different scroll, ring, and wand has a different functionality. And while all may be used in very similar ways, the core function is vastly different. While a purple wand may zap a goblin out of existence, a topaz ring might cause the player to suddenly begin levitating in the air. Thoroughly inter-connected aspects of the game are affected by items, and they represent a very central aspect of the project. This is why items are given their own module, *Item*, to manage items and shield their detailed functionalities.

5 Module Decomposition

5.1 Hardware Hiding Modules

Name: BasicIO

Secrets: Input and output devices.

Services: Serves as a layer beneath our application hiding the specifics of the input and output devices such as keystrokes and the monitor. The application uses it to retrieve commands from the user and display the game state.

Implemented By: Libtcod library.

Name: Doryen

Secrets: Details of display device and OS window interface.

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

Implemented By: Libtcod library.

Name: Input Format

Secrets: Interface to input devices.

Services: Detects keystrokes and provides virtual keycodes to the application. These are then interpreted by the system as commands from the user. As a layer above the BasicIO module, it hides not only the hardware itself, but the design of the device, for example it could allow the use of a touch screen with virtual buttons.

Implemented By: Libtcod library.

5.2 Behaviour-Hiding Module

Name: External

Secrets: The method by which gamestate is translated into display.

Services: All externally visible behavior passes through this module, as it controls the rendering of the game world. Interfaces between the hardware hiding modules by rendering onto them, and the software decision modules by making use of their data structures.

Implemented By: uistate.h

Name: Item

Secrets: Item data structures and behavior.

Services: Provides a consistent interface across all items, shielding the application from the details of the various types' internals. The application can move, activate, destroy, identify, or otherwise manipulate items freely using this module.

Implemented By: item.h

Name: Level

Secrets: Data storage format of the dungeon level.

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

Secrets: Level generation algorithm.

Services: Used by the controller modules to generate levels without requiring them to know the details of the algorithm.

Implemented By: level.h

Name: MainMenu

Secrets: Valid user name enforcement.

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

Implemented By: mainmenu.h

Name: Monster

Secrets: Monster data and algorithms implementing monster behavior.

Services: Customizes the behavior of various monsters to be greedy, fly, be aggressive, regenerate, etc. Also stores the data that defines the various monsters that can be found in the dungeon. eg. a dragon has certain behavior, a name, a certain quantity of hitpoints, etc.

Implemented By: monster.h

Name: PlayState

Secrets: Player character action selection.

Services: Calls the appropriate PlayerChar action methods according to the user's input. Renders the primary game screen.

Implemented By: playstate.h

Name: RipScreen

Secrets: Scoring and score storage.

Services: Handles reading and writing from the score file, and the contents thereof. The nature and location of the score table is known only to this module.

Implemented By: ripscreen.h

Name: UIState

Secrets: State of game interface.

Services: Layer between the game user and the game world. Handles command interpretation and menu control. Passes user commands to implementing modules.

Implemented By: uistate.h

5.3 Software Decision Module

Name: Coord

Secrets: Coordinate representation and coordinate 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. Also implements related algorithms such as the taxicab distance between two coordinates.

Implemented By: coord.h

Name: Feature

Secrets: Data structures implementing various objects found in the dungeon.

Services: Stores the data for any object found in the dungeon which is not a mob. This includes various objects such as stairs, piles of gold, items, and traps.

Implemented By: feature.h

Name: ItemZone

Secrets: Data structures storing items.

Services: Stores items and provides an interface mapping key-codes to specific items. Shields the application from details such as key-code ,and item stacking.

Implemented By: itemzone.h

Name: MasterController

Secrets: Game loop and application context.

Services: Directs the high-level flow of rendering, input-handling, and transitions between lower-level controllers. The flow between uistate.h members is modeled as a finite state machine.

Implemented By: mastercontroller.h

Name: Mob

Secrets: Internal data structures and behavior of creatures.

Services: Base module for all creatures in the dungeon (including the character). Provides interfaces for movement, combat, and various statistics.

Implemented By: mob.h

Name: Random

Secrets: The details of random number generation.

Services: Provides utilities to the rest of the application for the generation of random values. The algorithm used or the state of the PRNG are not exposed.

Implemented By: random.h

Name: Terrain

Secrets: Data structures of a dungeon tile.

Services: Stores the data for a single dungeon tile. This includes the character representing it, it's passability, transparency, and whether it is mapped.

Implemented By: terrain.h

6 Traceability Matrix

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

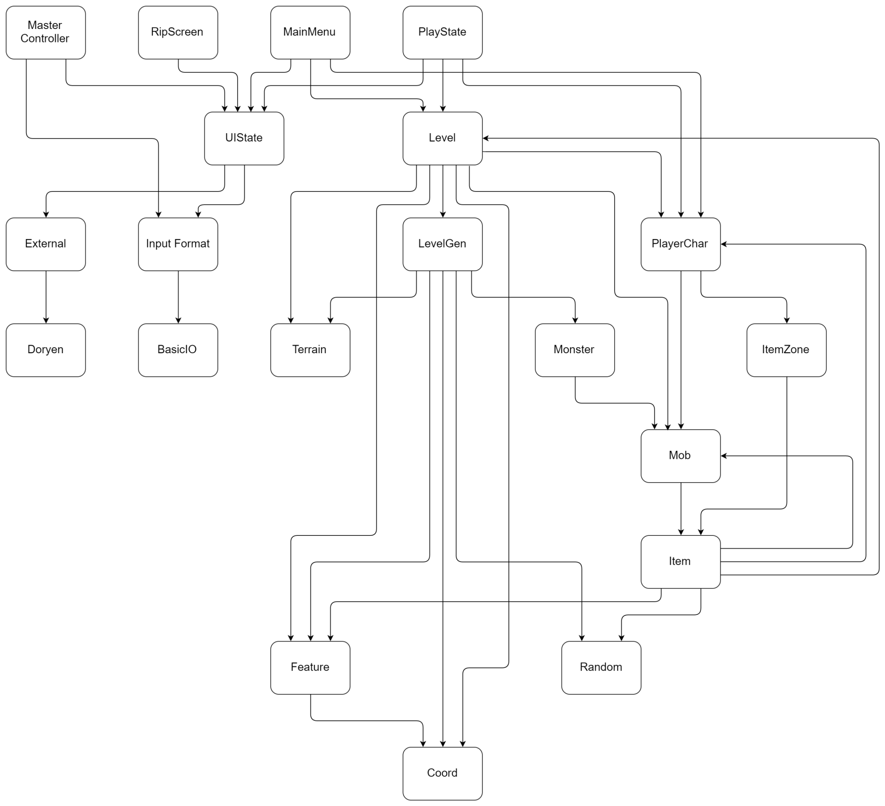
Table 6: **Trace Between Anticipated Changes and Modules**

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