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	Added Template
11/10/16	0.2	Finished part 2

1 Introduction

Hello World!

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.

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

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

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

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 themodule 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)

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.

5.2 Behaviour-Hiding Module

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: External.

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: –

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

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: 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: 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

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: 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: 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

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: 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: 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: 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: NNNNNNNNNN.

Secrets:

Services:

Implemented By:

5.2.1 Input Format Module (M??)

Name: NNNNNNNNNN.

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.

Implemented By: Libtcod

Name: NNNNNNNNNN.

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]

5.2.2 Etc.

5.3 Software Decision Module

Name: NNNNNNNNNNN.

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: –

5.3.1 Etc.

6 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
AC1	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

7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. **?** 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

8 References