Blaze Brigade

- Module Guide -

SFWR ENG 3XA3 - Section L02
007 (Group 7)

Jeremy Klotz - klotzjj
Asad Mansoor - mansoa2
Thien Trandinh - trandit
Susan Yuen - yuens2

November 10, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
Date 1	1.0	Notes

1 Introduction

1.1 Overview

Blaze Brigade is a tactical simulation role-playing game that combines the strategic challenges as a form of interactive entertainment for its users. This turn-based game allows users to advance their units into enemy territory, participate in combat and strive to eliminate all of the opposing units. In adaption to the open-source freeware, Tactics Heroes, Blaze Brigade will incorporate new functional and design enhancements that may not be available to the existing open-source project. Such enhancements include the implementation of new features within the unit movement, combat and strategy aspect of the game as well as improved graphical representations of the menu menu and gameplay to improve the overall experience of the game.

1.2 Context

The system is fully devised and presents all of its functional and non-functional requirements in the Software Requirement Specification (SRS). As these requirements state the desirable properties of the system, the design documents will further evaluate on how these requirements are identified and achieved. The Module Guide (MG) will serve as a tool to decompose the following system into a modular structure adhering to the principle of information hiding. Upon reaching the finalized version of this document, the Module Guide can be distributed amongst various groups in order to learn and identify parts of the software that is being presented. These various groups are as follows:

- Developers and maintainers: Decomposing the system and documenting into the Module Guide will aid the developers and maintainers to understand the system-as-is and recognize what areas of the software are likely to be changed. In addition, a sense of the overall design will be structured and will be maintained in the following developments phases yet to come.

- Designers: In addition to the design pattern being documented, designers are able to determine whether the designs are constructed as initially specified. With the following anticipated changes to be happening, which areas of software is flexible and feasible to accommodate new design changes.

- New recruits or outsourced resources: The documentation will onboard the new recruits in familiarizing the overall structure of the implementation adhering to a specific design principle. This will reduce the downtime of debugging and have an advantage of multiple groups working on the system at once. Furthermore, if an external team were to implement the system or would like to carry out further improvements after the project timeline, this document will serve as an aid to determine the existing framework and how further implementation can take place.

In addition to the Module Guide, the Module Interface Specification (MIS) is also a product of the design documentation. The specification defines the syntax and semantics that are associated with the functions provided in the source code. Tools like Doxygen have been utilized to generate a set of documentation that will indicate the characteristics of the functions in terms of the corresponding inputs, outputs, assumptions, exceptions, state and environment variables. These characteristics will further aid in observing how the implementation is taken place and how the design constitutes from these functions.

1.3 Design Principle

The design principle taken into consideration revolves around the decomposition of the overall system into a modular structure of subsystems. These subsystems are observed in an abstract manner, hiding any details that may complicate the process. This act of information hiding and encapsulation ensures that each modules hides some design aspect from the rest of the system and analyzing which areas are expected to change. This process plans for the following subsystems to be changed, hence protecting other subsystems of the software from major changes if the design decision is changed.

1.4 Outline

The Module Guide is organized in the given order. Section 2 lists the anticipated and unlikely changes of the software requirements. Section 3 decomposes the system into a module hierarchy of the likely changes. Section 4 establishes the connection between the software requirements with the modules. Section 5 gives a detailed insight on how the modules have been decomposed with their corresponding descriptions. Section 6 includes two traceability matrices comparing the modules with the software requirements and anticipated changes. At last, section 7 pinpoints the use hierarchy between the modules.

2 Anticipated and Unlikely Changes

This section identifies possible changes to the software system. These changes to the design choices are organized into two categories. Anticipated changes, and unlikely changes. Anticipated changes are listed in Section 2.1, and the unlikely changes are listed in Section 2.2.

2.1 Anticipated Changes

The design decisions in this section are likely to change because they are hidden in modules. When these changes are made, they can be done easily and not affect other modules of the project.

AC1: All classes that implement the Class interface are likely to have their stats change for balancing reasons.

AC2: All weapons that implement the Weapon interface are likely to have their stats change for balancing reasons.

AC3: The getHitRate() and getCritRate() methods inside the DamageCalculations class are likely to change for balancing reasons.

AC4: All sprites from outside sources. It has been determined that the project should contain all original content.

2.2 Unlikely Changes

The following design decisions are unlikely to change because they affect many modules. Since they affect multiple modules, changing these decisions may result in multiple changes in the overall design of the project. Unless these changes are necessary, they will not occur.

UC1: Input/Output devices (Input: Mouse, Output: Updated Model and Screen).

UC2: The software implements the MVC (Model-View-Controller) architecture.

UC3: The Graph of nodes that represents the playable grid.

UC4: Nodes are identified by their x and y coordinates.

UC5: The path finding algorithm.

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

M2: Behaviour-Hiding Module

M3: Software Decision Module

M4: Input Formatting Module

M5: Output Formatting Module

M6: Model Module

M7: Game State Module

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	Input Formatting Module Output Formatting Module Model Module
Software Decision Module	Game State Module

Table 2: Module Hierarchy

Since Blaze-Brigade consists of purely software, M1 does not apply to the system. The software never interfaces with the hardware itself. The lowest level of interfacing with the software is the OS.

4 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 and Table 4.

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 the module 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 Module

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

5.2 Behaviour-Hiding Module(M2)

Secrets: The behavioural process of the software.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) document. This module serves as a representation of the software decision module. The programs in this module will need to be modified if there are changes in the SRS document.

Implemented By: M4, M5 , M??.

5.2.1 Input Formatting Module (M4)

Secrets: The format and structure of the input data that manipulates the software.

Services: Converts the input data into the data structure used by the Game State Module.

Implemented By: MouseHandler.cs, Button.cs

5.2.2 Output Formatting Module (M5)

Secrets: The result of input data.

Services: Receives data from the Game State Module, and updates the software accordingly.

Implemented By: DamageCalculations.cs, GameFunction.cs

5.2.3 Model Module (M6)

Secrets: The design decisions that implement the structure of the software.

Services: Includes data structures that update based on user interaction.

Implemented By: Game.cs, Graph.cs, Node.cs, Unit.cs, Weapon.cs, Player.cs

5.3 Software Decision Module

Secrets: The design decisions that determine *how* the software updates.

Services: Includes data structure used in the system that do not directly interact with the user.

Implemented By: M7

5.3.1 Game State Module

Secrets: The design decisions that hold all possible states of the software.

Services: Retrieves user input and updates the current state of the software. From here, the module sends the current state to the output formatting module.

Implemented By: GameState.cs

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
FR1	M1, M4, M5
FR2	M6, M7
FR3	M4, M5
FR4	M4, M5, M6
FR5	M6

Table 3: Trace Between Functional Requirements and Modules

Req.	Modules
NFR1	M1, M2, M5
NFR2	M3, M4
NFR3	M3, M6
NFR4	M3, M6
NFR5	M3, M6, M7
NFR6	M1, M2
NFR7	M5
NFR8	M1, M3

Table 4: Trace Between Non-Functional Requirements and Modules

AC	Modules
AC??	M1
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 5: 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