

**SE 3XA3:
Module Guide
CraftMaster**

Group Number: 307
Group Name: 3 Craftsmen
Members:
Hongqing Cao 400053625
Sida Wang 400072157
Weidong Yang 400065354

Contents

1	Introduction	1
1.1	Project Overview	1
1.2	Context of Module Guide	1
1.3	Design Principles	2
1.4	Content Structure	2
2	Anticipated and Unlikely Changes	3
2.1	Anticipated Changes	3
2.2	Unlikely Changes	3
3	Module Hierarchy	4
4	Connection Between Requirements and Design	4
5	Module Decomposition	4
5.1	Hardware Hiding Modules (M1)	5
5.2	Behaviour-Hiding Module	5
5.2.1	Input Format Module (M??)	5
5.2.2	Etc.	5
5.3	Software Decision Module	5
5.3.1	Etc.	6
6	Traceability Matrix	6
7	Use Hierarchy Between Modules	7

List of Tables

1	Revision History	ii
2	Module Hierarchy	4
3	Trace Between Requirements and Modules	6
4	Trace Between Anticipated Changes and Modules	6

List of Figures

1	Use hierarchy among modules	7
---	---------------------------------------	---

Date	Editor(s)	Change
Mar 9	Sida	General Content added
...

Table 1: **Revision History**

1 Introduction

1.1 Project Overview

CraftMaster is a re-implementation of [Michael Fogleman's Simple Minecraft-inspired Demo](#) (referred to as **original project** in the following content), which is developed by Python and Pyglet. The CraftMaster design team has initiated and completed multiple requirements (specified in the [Software Requirement Specifications\(SRS\)](#)) to add new features, including new block types, day and night mode shift, game saving, and game menu frames, to the original project. During the software design process, the team has also applied software architecture design patterns and software design principles to increase the quality of the software design. The specification of those design methodology will be described in this document.

In terms of the software product characteristics, CraftMaster is a 3D Sandbox Game that allows players to control the character and build the game world based on their imagination. We believe that CraftMaster will be beneficial to teenagers and children in the way that it inspires them to unleash their creativity.

1.2 Context of Module Guide

The [SRS](#) document shows the features, functionalities and desired properties that the system should have. The **Module Guide(MG)** is generated based on the [SRS](#), which further evaluates how requirements are achieved and also specifies the modular structure decomposition of the software system. It will be distributed to help potential readers easily identify the decomposed parts of the software. The potential readers are as follows:

- **New Project Members:** The **MG** acts as a guideline for the new project members to easily and quickly understand the modular structure of the system and its decomposition specifications. With this document, those new members can search for relevant modules more efficiently.
- **Designers:** The **MG** is used to help software system designers check for the consistency among modules, the flexibility of the design, and the feasibility of the modular decomposition.
- **Developers:** The hierarchical structure specified in the **MG** will give the developers a better understanding of the system decomposition and use relationships between different modules.
- **Maintainers:** The hierarchical structure of the system improves the maintainers' understanding of either the system as a whole or individual modular parts when they need to make changes to the system and the documents.

The **Module Interface Specifications(MIS)** is another section of the Design Specifications Documents other than the **MG**. The **MIS** shows the semantics and syntax of exported functions for each module in details. The **MG** should be an entry document for the design specifications and the readers should read the **MG** first to get an overview of the system, then browse the **MIS** for further references once they identify which module(s) they are searching for.

1.3 Design Principles

- **High Cohesion and Low Coupling:** This principle has been applied to the project in the way that the modules are designed to be strongly related and the dependency has been minimized.
- **Open-Closed Principle:** The modules are designed to be closed to modification and the system is designed to easily extended. For instance, the **Screen.py** module is implemented as a template and all other scenes(such as **gameScene.py**, **mainScene.py** and **settingScene.py**) inherit it. There might be new scenes to be implement in the future to easily extend the system.
- **Liskov Substitution Principle:** The inheritance relationships among modules follows the Liskov Substitution Principle.
- **Dependency Inversion Principle:** The inheritance design pattern is used to support the abstraction of the system. With the abstraction, dependency inversion principle can be followed.
- **Interface Segregation Principle:** The inheritance design pattern minimizes the interfaces of subclass modules, which follows the Interface Segregation Principle.
- **Law of Demeter:** The abstraction of the system supports the principle of Law of Demeter. The communications only depend on the interfaces as limited knowledge.
- **Information Hiding:** There are many private methods implemented to support the information hiding strategy.

1.4 Content Structure

The rest of the document is organized as follows:

- Section 2 lists the anticipated and unlikely changes of the software requirements.
- Section 3 summarizes the module decomposition that was constructed according to the likely changes.
- Section 4 specifies the connections between the software requirements and the modules.

- Section 5 gives a detailed description of the modules.
- Section 6 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 7 describes the use relation between modules.

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 format of the initial input data.

...

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: Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

UC2: There will always be a source of input data external to the software.

...

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 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.

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 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

5.2 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: -

5.2.1 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]

5.2.2 Etc.

5.3 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: –

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
AC2	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