SE 3XA3: Module Guide
DNA Says

Team #10, Team Name: DNA
Kareem Abdel Mesih - abdelk2
John-Paul Dakran - dakranj
Shady Nessim - nessimss

November 8, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
3/11/2016	1.0	Addition of section 2
7/11/2016	1.1	Addition of section 3
8/11/2016	1.2	Addition of section 1

1 Introduction

1.1 Overview

This project is a redevelopment of the famous game Simon Says, with a slight modification that makes DNA Says unique while keeping the integrity of the game consistent with the original version. The game consists of three distinct modes - Kareem Says, JP Says and Shady Says.

1.2 Context

This document consists of the Module Guide (MG) for the project DNA Says. This is the second portion of the design documentation along with the Module Interface Specification (MIS) - which explains the semantics of the code in natural language.

The Module Guide is a decomposition of the software system into modules. A module is an independent self-contained unit that makes up a complex software system. Decomposing a problem into modules is an extremely important aspect of software design as it promotes the principle of information hiding. Each module is completed concurrently by a programmer and houses a secret of the modules functionality.

The Module Guide (MG) reveals how the software system will carry out the functionality that is described in the Software Requirements Specification (SRS) document. The potential readers of the document are listed below:

- Designers/Developers: This document is extremely important for the designers of the software system. It provides a means for the designers to easily identify different parts of the software and relate the implementation to the requirements.
- New Project Members: This document allows new project members to easily identify the components of the software system. It is a simple way to understand and locate information that relates to specific parts of the software.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.

1.3 Design Principles

The first major design principle utilized by the developers of DNA Says is modularity. This design principle was achieved by dividing the software into independent self-contained units. After the conncurrent development of the individual modules they are integrated together to full the functional and non-functional requirements in the SRS.

The second major design principle used in this software system is abstraction. After division of the software system into modules, each module was viewed at an abstract level. In other words, the design team neglected the implementation of the module and focused on the most essential information which is relevant to that specific module.

The third major design principle utilized by the developers of DNA Says is information hiding. This design principle was employed during the development of the individual modules. The theory behind this principle is that the modules were designed in such a way that the information and secret of a module cannot be exposed to other modules that do not need to know that information.

1.4 Document 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.

1.5 Naming Conventions & Terminology

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
MG	Module Guide
SRS	Software Requirements Specification
MVC	Model View Controller

Table 3: **Table of Definitions**

Term	**Definition**
Mode	Different subsections of the game
Gantt Chart	Chart outlining the timeline of the project
Python	A programming language
Pygame	Cross-platform set of Python modules designed for writing video games

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 routine that the user follows to start the program. There will be an attempt to create a standalone application rather than require the user to install Python and Pygame.

AC2: The main menu.

AC3: The flow of the program. There will be a button within each mode to go back to the main menu.

AC4: The colours of labels and buttons of the main menu along with all the modes.

AC5: The size of the text in each mode.

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 MVC structure will be transformed to accommodate for the separation of concerns, along with information hiding.

3 Module Hierarchy

This section provides an overview of the module design. 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.

M1: Hardware-Hiding Module*

M2: Main Module

M3: Menu Module

M4: JP Module

M5: Kareem Module

M6: Shady Module

M7: Setup Module

M8: Update Module

M9: ShowInst Module

M10: ShowScore Module

M11: ShowGoBack Module

M12: DrawKeys Module

M13: FlashKeyAnimation Module

M14: ChangeBackgroundAnimation Module

M15: GameOverAnimation Module

M16: CheckForQuit Module

Level 1	Level 2
Hardware-Hiding Module	
Behaviour-Hiding Module	M3 M7 M8 M9 M10 M11 M12
Software Decision Module	M2 M4 M5 M6 M13 M14 M15 M16

Table 4: Module Hierarchy

Note*: The Hardware-Hiding Module is not implemented in the hierarchy as there is no hardware involved in this software system.

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 5.

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 (M16)

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	M16, 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 5: Trace Between Requirements and Modules

AC	Modules
AC??	M16
AC5	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 6: 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