Module Guide for Chess Connect

Team #4,
Alexander Van Kralingen
Arshdeep Aujla
Jonathan Cels
Joshua Chapman
Rupinder Nagra

January 18, 2023

1 Revision History

Table of Revisions

Table 1: Revision History

Date	Developer(s)	Change
1/17/2023	1.0	Detailed Modules used by Arduino Mega 2560
2023-01-16	Jonathan Cels, Rupinder Nagra	Web Application Modules
2023-01-18	Jonathan Cels, Rupinder Nagra	Finalized Web Application Modules

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
UC	Unlikely Change
FEN	Forsyth-Edwards Notation

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- 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.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 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 9 describes the use relation between modules.

4 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 4.1, and unlikely changes are listed in Section 4.2.

4.1 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 pin configuration for Hall-effect sensors

AC2: The pin configuration for LEDs

AC3: Transition logic between the various game states

AC4: Handling different user actions

4.2 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: Valid chess move logic.

UC2: Data type definition locations may be shifted between different modules.

5 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: Arduino Controler Module

M3: Arduino Module

M4: Piece Identification Module

M5: Chess Board Module

M6: Communication Module

M7: Software Serial Module

M8: Web Application Input Module

M9: Display Module

M10: Web Application Output Module

M11: User Mode Module

M12: Board Module

M13: Web Application Game State Module

M14: Engine Module

The Hardware-Hiding module is implemented by the microcontroller as firmware that controls all of the hardware components. This is implemented by the kernel to also run embedded software loaded onto the controller. The Arduino and SoftwareSerial modules are present in the Arduino framework and will not be recreated for this project.

Level 1	**Level 2**
Hardware Hiding Module	Arduino Controller Module
	Arduino Module
	Software Serial Module
Behaviour-Hiding Module	Web Application Input Module
	Display Module
	Web Application Output Module
	Piece Identification Module
	Communication Module
Software Decision Module	User Mode Module
	Board Module
	Web Application Game State Module
	Engine Module
	Chess Board Module

Table 2: Module Hierarchy

6 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.

[The intention of this section is to document decisions that are made "between" the requirements and the design. To satisfy some requirements, design decisions need to be made. Rather than make these decisions implicit, they are explicitly recorded here. For instance, if a program has security requirements, a specific design decision may be made to satisfy those requirements with a password. In scientific examples, the choice of algorithm could potentially go here, if that is a decision that is exposed by the interface. —SS]

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). 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. *Chess Connect* means the module will be implemented by the Chess Connect 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.

7.1 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

7.1.1 Arduino Controller Module (M2)

Secrets: The logic for changing game state and reading user actions.

Services: Reads inputs from sensors and writes to outputs to turn on/off LED's. Main overarching module that runs and controls all other modules to coordinate the logic and messages with the game state.

Implemented By: Arduino Mega 2560 Rev3 Controller

7.1.2 Arduino Module (M3)

Secrets: The Arduino API that allows access to the Hardware Abstraction Layer (HAL).

Services: Provides an interface to the hardware through pin assignments and hardware ID's.

Implemented By: Arduino Controller Module

7.1.3 Piece Identification Module (M4)

Secrets: The values read by the Hall-effect sensors that are translated into the Piece data type.

Services: Translates sensor values into pieces and positions on the chess board. Provides Piece identification data types to the rest of the program.

Implemented By: Arduino Controller Module, Chess Board Module

7.1.4 Chess Board Module (M5)

Secrets: Contained within this module are:

- the valid move logic for chess pieces;
- The translation of the piece configuration on the chess board into a FEN string
- Checking for checkmate

Services: Handles all of the logic used in the game of chess. This module processes commands recieved from the Web Application and Teensy Controller into actions to be implemented within the game and gives feedback related to valid moves and the game/board state.

Implemented By: Arduino Controller Module

7.1.5 Communication Module (M6)

Secrets: Message encoding string used to send board state, game mode and resignation signal. Message decoding string to interpret best moves, the check/mate/stalemate warning and commands from the Teensy controller.

Services: Reads and writes to the communication pins, processes and sends commands to various parts of the embedded software.

Implemented By: Arduino Controller Module, Chess Board Module

7.1.6 SoftwareSerial Module (M7)

Secrets: The methods in which the data is encoded, flags used to acknowledge connection, successful data transmission, incoming signal, etc.

Services: The Arduino API that allows for sending and receiving various data types through serial communication.

Implemented By: Communication Module

7.2 Behaviour-Hiding Module

7.3 Behaviour-Hiding Modules

7.3.1 Web Application Input Module (M8)

Secrets: Input data.

Services: Takes in input data of current board state to provide to other modules.

Implemented By: Chess Connect (Node.js libraries, Bluetooth)

7.3.2 Display Module M9

Secrets: Graphics output data.

Services: Allows users to view the current board configuration of the system.

Implemented By: Chess Connect (React.js framework)

7.3.3 Web Application Output Module M10

Secrets: None.

Services: Takes game state and engine moves, encodes the data, and transmits it.

Implemented By: Chess Connect (React.js framework, Bluetooth)

7.4 Software Decision Module

7.4.1 User Mode Module M11

Secrets: User mode logic and data.

Services: Handles switching between user modes and communicating with mode-specific modules.

Implemented By: Chess Connect (React.js framework)

7.4.2 Board Module M12

Secrets: Board data.

Services: Stores and modifies board state information.

Implemented By: Chess Connect (React.js framework)

7.4.3 Web Application Game State Module M13

Secrets: Game state and data.

Services: Handles checking the game state (none, check, checkmate, stalemate).

Implemented By: Chess Connect (React.js framework, Node.js libraries)

7.4.4 Engine Module M14

Secrets: Chess engine moves.

Services: Uses the board state to calculate 3 possible moves.

Implemented By: Chess Connect (Node.js libraries and Stockfish)

8 Traceability Matrix

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

Req.	Modules
GA1, GA2, EB1, BB1, BB2	M5
NB1	M4
NB2, EB2, BD1	M5
ND1, ED1, ED2	M6
NA1	M8, M6
NA3	M9, M13
BA1, BA2	M9
EA2, EA3, EA4	M14
EA5, EA6	M9
BB1, BB2, BB3	M5

Table 3: Trace Between Requirements and Modules

AC	Modules
AC1	M2
AC2	M2
AC3	M2, M5
AC4	M2, M5, M6

Table 4: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) 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

References

FEN. Fen (forsyth-edwards notation) - chess terms. https://www.chess.com/terms/fen-chess. Accessed: 2023-01-18.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.