Module Guide for Software Engineering

Team 6, Pitch Perfect
Damien Cheung
Jad Haytaoglu
Derek Li
Temituoyo Ugborogho
Emma Wigglesworth

December 21, 2024

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change
[etc. —SS]	[... —SS]

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 specific hardware on which the software is running.

AC2: The format of the initial input data.

...

[Anticipated changes relate to changes that would be made in requirements, design or implementation choices. They are not related to changes that are made at run-time, like the values of parameters. —SS]

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

...

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: User Interface Module

M2: Authentication Module

M3: Team Management Module

M4: Game Management Module

M5: Announcements Module

M6: Standings Module

M7: Scheduling Module

M8: Waiver Module

M9: Notification Module

M10: Backend Module

M11: Scheduling Algorithm Module

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	User Interface Module Authentication Module Team Management Module Game Management Module Announcements Module Scheduling Module Standings Module Waiver Module
Software Decision Module	Database Module Notification Module Scheduling Algorithm Module

Table 1: 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 2.

[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. —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. *Software Engineering* means the module will be implemented by the Software Engineering 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 Behaviour-Hiding Module

7.1.1 User Interface Module (M1)

Secrets :

- The structure of inputs (e.g., username, password, team names, game scores, and scheduling information).
- User interface components, logic, and states.

Services :

- Rendering views, collecting user input, and displaying data retrieved from the backend.
- Backend interactions to fetch data and send requests.
- Manages user role-based views.

Implemented By: Team 6

Type of Module: Interface.

7.1.2 Authentication Module (M2)

Secrets: The implementation details of password encryption, token generation, and user session management.

Services: Handles the following:

- User authentication, including login, registration, and logout.
- Verification of user credentials and assignment of roles (player, captain, administrator).
- Management of access tokens for secure API interactions.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.3 Team Management Module (M3)

Secrets :

- Team creation for Captains and team joining joining processes and data for players.
- Waiver acceptance states and data.

Services :

- Handles captain input for team creation.
- Manages user-team associations (joining teams), and processing waiver sign-offs.
- Provides interface for captains to manage their teams (e.g., adding/removing players) and roster validation.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.4 Game Management Module (M4)

Secrets :

- The structure of the input data includes game details (game ID, teams, scores, and status).
- Game record verification.
- Game statuses processes (e.g., scheduled, completed, canceled).

Services :

- Enables game score reporting and updating game statuses for appropriate roles.
- Updates store and retrieve game results to trigger standing updates.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.5 Announcements Module (M5)

Secrets: The structure of announcements, including title, message content, timestamps, as well as how announcements are stored, retrieved, and delivered to users, notifying them.

Services :

- Allows admin to create, edit, and delete announcements for the league.
- Enables users to view announcements and ensures announcnement delivery via notifications.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.6 Standings Module (M6)

Secrets: The formulas for calculating rankings, win-loss records, and tie-breaking rules.

Services: Handles the following:

- Tracks and updates league standings based on game results.
- Displays rankings, win-loss records, and statistical summaries for all teams.
- Supports tie-breaking scenarios and rankings by division.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.7 Scheduling Module (M7)

Secrets: Schedule management, including data structures and communication with the scheduling algorithm.

Services :

- Handles game schedule creation and updates.
- Manages scheduling constraints like team preferences.

- Integrates with the Scheduling Algorithm Module for optimization.

Implemented By: Team 6

Type of Module: Abstract Object

7.1.8 Waiver Module (M8)

Secrets: Waiver completion data.

Services :

- Manages waiver access and completion.
- Tracks waiver status for players.
- Integrates with user accounts to validate players.

Implemented By: Team 6

Type of Module: Abstract Object

7.2 Software Decision Module

7.2.1 Notification Module (M9)

Secrets: The protocols for sending important league updates via email notifications.

Services: Sends notifications to users for important updates, such as schedule changes, game results, or captain messages.

Implemented By: External Library

7.2.2 Backend Module (M10)

Secrets: The architecture, database schema, and integration logic for backend services.

Services :

- Manages server-side logic and handles API requests and responses.
- Database to storage and data retrieval (e.g., user data, schedules, game results, etc).
- Secure communication between the frontend and backend.

Implemented By: Team 6

7.2.3 Scheduling Algorithm Module (M11)

Secrets: The algorithm used to generate and optimize schedules, including constraints like team preferences, field availability, etc.

Services :

- Generates game schedules based on team inputs and league constraints for admin use.
- Supports conflict resolution for rescheduling (e.g., double bookings, unavailable fields).
- Optimizes schedules to balance fairness and preferences across teams.

Implemented By: Team 6

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
R1	M1, M2, M10
R2	M1, M2, M10
R3	M1, M7, M11
R4	M1, M6, M10
R5	M1, M3, M10
R6	M1, M3, M8, M10
R7	M1, M3, M7, M11, M10
R8	M1, M7, M11
R9	M1, M5, M9
R10	M1, M7, M11
R10	M1, M7, M11
R12	M1, M4, M10

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M??
AC2	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 3: 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.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

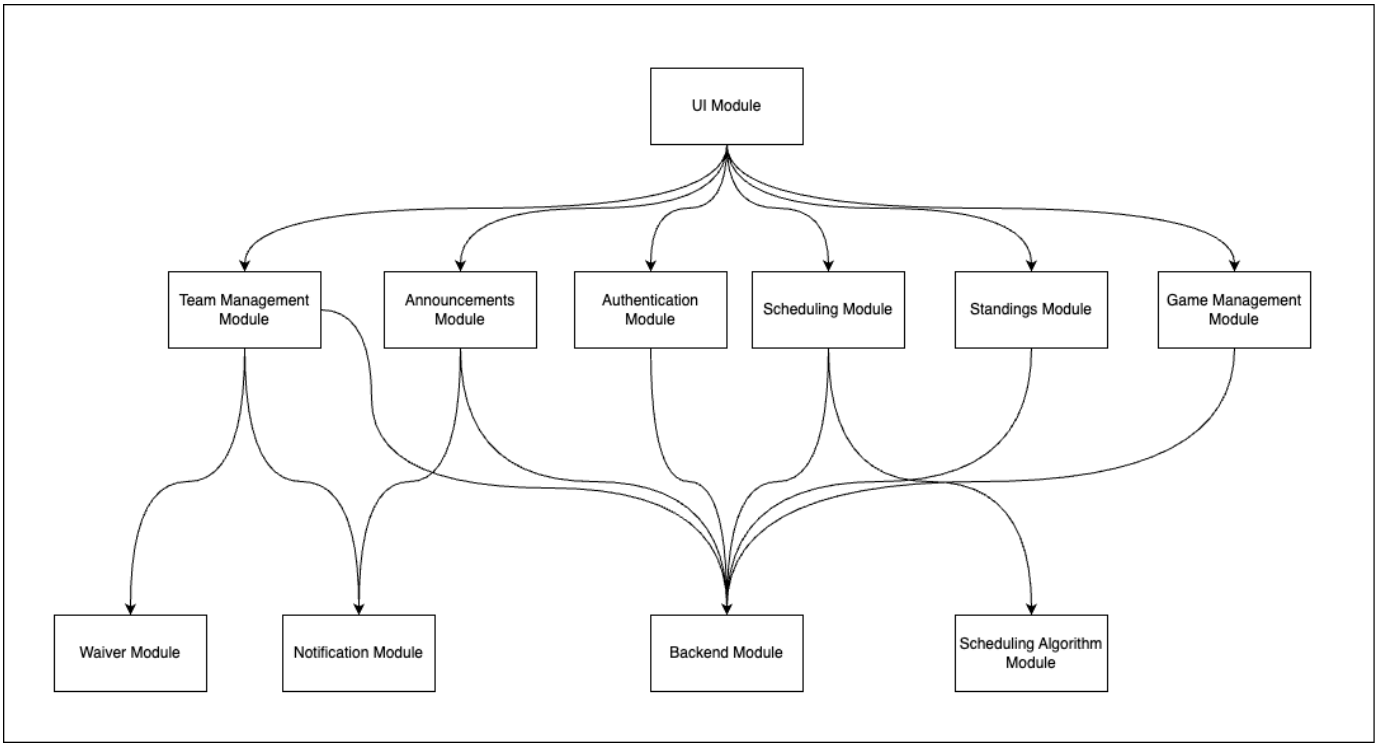

Figure 1: Use hierarchy among modules

10 User Interfaces

[Design of user interface for software and hardware. Attach an appendix if needed. Drawings, Sketches, Figma —SS]

11 Design of Communication Protocols

[If appropriate —SS]

12 Timeline

[Schedule of tasks and who is responsible —SS]
[You can point to GitHub if this information is included there —SS]

References

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.