Module Guide for SubLiMat

Uriel Garcilazo Cruz

April 3, 2025

1 Revision History

Date	Version	Notes
Mar 17th 2025	1.0	Notes
Apr 1st 2025	1.0	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
SubLiMat	Explanation of program name
UC	Unlikely Change

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 (e.g., sequences, substitution matrices).

AC3: The alignment algorithm used (e.g., Needleman-Wunsch, Smith-Waterman).

AC4: The set of substitution matrices supported (e.g., DNA, protein matrices).

AC5: The format of the output data (e.g., plain text, JSON, graphical visualization).

AC6: The method for calculating gap penalties that could go from fixed to variable, changing dynamically and requiring its own module.

AC7: The performance optimization strategy (parallel processing, GPU acceleration).

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).

UC2: The type of alignment problem (e.g., global alignment using Needleman-Wunsch).

UC3: The type of sequences handled (e.g., DNA sequences).

UC4: The scoring system used for alignment (e.g., substitution matrices and gap penalties).

UC5: The format of the output data (e.g., plain text for alignment results).

UC6: The programming language used for implementation (e.g., Python).

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: Hardware-Hiding Module

M2: Control Manager Module

M3: Alignment (Needleman-Wunsch) Interface Module

M4: Substitution Matrix Module

M5: File Manager Module

M6: Sequence Data Structure Module

Level 1	**Level 2**
Hardware-Hiding Module (M1)	
Behaviour-Hiding Module	Control Manager Module (M2) Alignment (Needleman-Wunsch) Interface Module (M3)
Software Decision Module	Substitution Matrix Module (M4) File Manager Module (M5) Sequence Data Structure Module (M6)

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.

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. *SubLiMat* means the module will be implemented by the SubLiMat 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

7.2 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: –

7.2.1 Control Manager Module (M2)

This module is responsible for managing the overall control flow of the program, including the initialization of the program, reading input files, and invoking the necessary modules for sequence alignment. It serves as the main entry point for the program and coordinates the interaction between different components.

Secrets: Controls the flow of information in the program during the benchmarking process.

Services: Computes the optimal alignment score between two sequences.

Implemented By: SubLiMat

Type of Module: Library

Uses: M4 (Substitution Matrix Module), M6 (Sequence Data Structure Module), M3 (Alignment Interface Module)

7.2.2 Alignment (Needleman-Wunsch) Interface Module (M3)

This module implements the Needleman-Wunsch algorithm for global pairwise sequence alignment, generating a comparative matrix F from the input sequences, and given a substitution matrix.

Secrets: The implementation of the Needleman-Wunsch algorithm for global sequence alignment.

Services: Computes the optimal alignment score between two sequences.

Implemented By: SubLiMat

Type of Module: Library

Uses: M4 (Substitution Matrix Module), M6 (Sequence Data Structure Module)

7.3 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: –

7.3.1 Substitution Matrix Module (M4)

This module is responsible for managing the substitution matrices used in the alignment process. It provides functions to load, validate, and access different substitution matrices, as well as their associated scoring rules. The module also handles the gap penalties and other parameters related to the scoring system.

Secrets: The structure and scoring rules for substitution matrices.

Services: Provides access to different substitution matrices (e.g., JC, K80, HKY85) and their penalizing costs.

Implemented By: SubLiMat

Type of Module: Abstract

Uses: M5 (File Manager Module)

7.3.2 File Manager Module (M5)

This module is responsible for managing the input and output files used in the program. It handles file reading, writing, and validation, ensuring that the data is correctly formatted and accessible. The module also provides functions for exporting benchmark results to CSV format.

Secrets: The format and structure of input and output files, including FASTA sequence files, substitution matrix files, and benchmark results. Path management for accessing and storing data.

Services: Provides functions to read nucleotide sequences from FASTA files, load substitution matrices from text files, and export benchmark results to CSV format. Handles file validation and error reporting for invalid inputs.

Implemented By: SubLiMat; pandas (Python library)

Type of Module: Abstract

7.3.3 Sequence Data Structure Module(M6)

This module is responsible for managing the representation and validation of biological sequences. It provides functions to validate sequences and ensure they contain only valid characters (A, T, G, C, or _). The module also handles the storage and manipulation of sequences during the alignment process.

Secrets: The representation and validation of biological sequences.

Services: Provides functions to validate sequences and ensure they contain only valid characters (A, T, G, C, or _).

Implemented By: SubLiMat

Type of Module: Abstract Object

Uses: M5 (File Manager Module)

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
FR1 (Input sequences and parameters)	M1, M6, M4, M5
FR2 (Build comparative matrix)	M3, M4, M2
FR3 (Calculate optimal alignment scores)	M3, M2
FR4 (Verify inputs)	M6, M5
FR5 (Output aligned sequences)	M3, M5
NFR1 (Accuracy)	M3, M4, M2
NFR2 (Usability)	M1, M3, M2
NFR3 (Maintainability)	M4, M3, M6, M5
NFR4 (Portability)	M1, M3
NFR5 (Performance)	M3, M2

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M1
AC2	M6, M4, M5
AC3	M3, M2
AC4	M4
AC5	M5
AC6	M3, M4
AC7	M3, M2

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the 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.

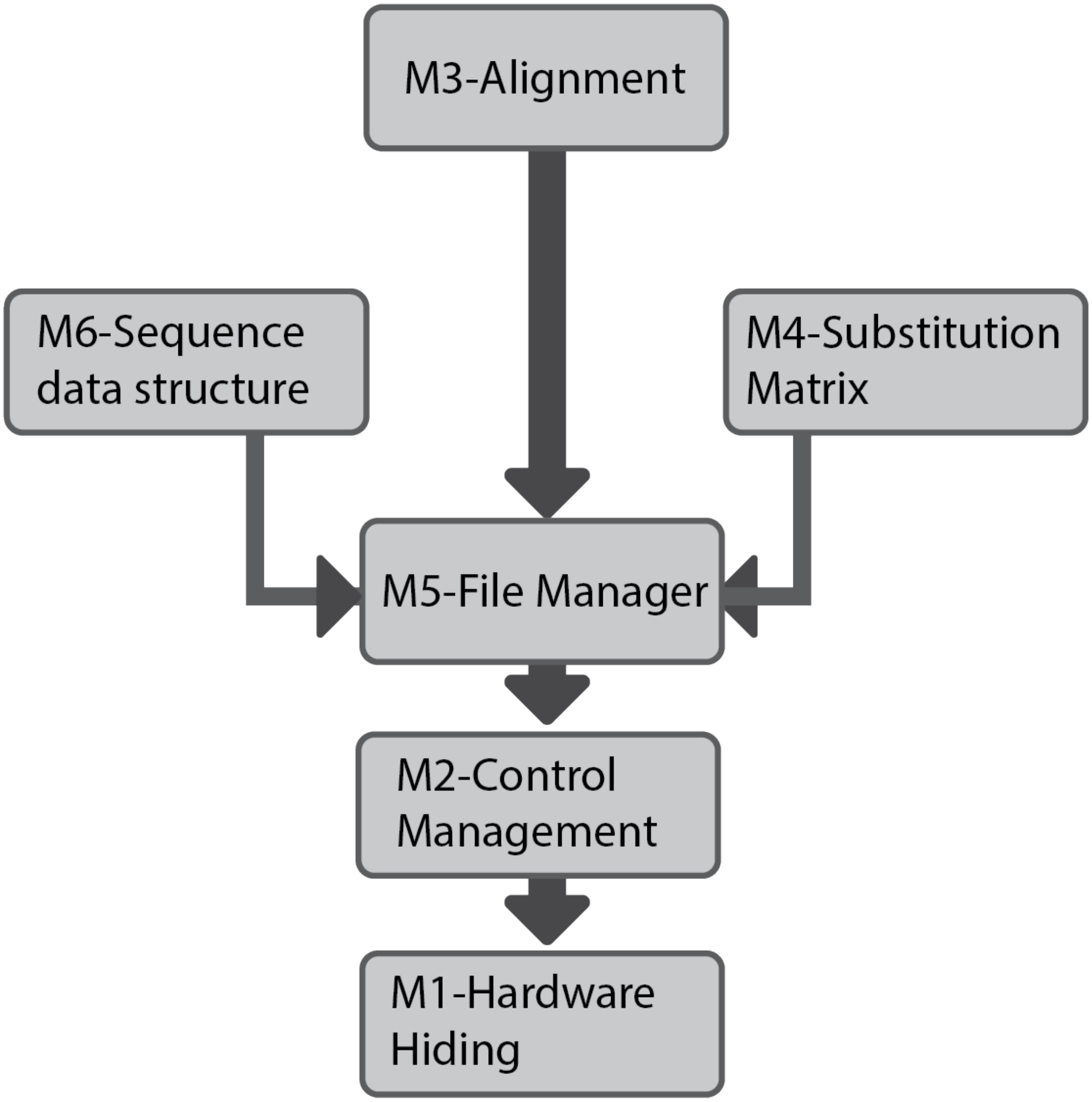

Figure 1: Use hierarchy among modules

References

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.