Module Guide: A Library of Simplex Method Solvers
[Work the name of your library into your MG title —SS]

Hanane Zlitni

November 05, 2018

1 Revision History

Date	Version	Notes
November 05, 2018	1.0	First Draft

Contents

List of Tables

List of Figures

2 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 used 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 3 lists the anticipated and unlikely changes of the software requirements. Section 4 summarizes the module decomposition that was constructed according to the likely changes. Section 5 specifies the connections between the software requirements and the modules. Section 6 gives a detailed description of the modules. Section 7 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 8 describes the use relation between modules.

3 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 3.1, and unlikely changes are listed in Section 3.2.

3.1 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 library's external interface.

AC3: The format of the initial input data.

AC4: The assumptions related to the input data discussed in Section 6.2 in the CA (Zlitni (2018)).

AC5: The data structure used to store the input data.

AC6: The linear programming algorithm used to obtain the optimal solution.

AC7: The format of the final output data.

3.2 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 decisions 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 objective of LoSMS will always be to find and output the optimal solution(s) of linear programs that satisfy all constraints and sign restrictions.

4 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: Behaviour-Hiding Module

M3: External Interface Module

M4: Input Module

M5: Input Conversion Module

M6: Data Structure Module

M7: Simplex Method Solver

M8: Output Module

Note that M1 is a commonly used module and is already implemented by the operating system. Therefore, it will not be reimplemented. Furthermore, for the current scope of the project where minimization linear programs are solved by converting them to maximization problems, it makes more sense to have one simplex method solver module that provides services for solving both types of problems. In case of further expansion of the project and the usage of other variations of the algorithm to solve minimization problems, it would then make more sense to have two separate simplex method modules: one for maximization problems and one for minimization problems.

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	
Software Decision Module	External Interface Input Input Conversion Data Structure Simplex Method Solver Output

Table 1: Module Hierarchy

5 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.

6 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. 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.

6.1 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

6.2 Behaviour-Hiding Module (M2)

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: –

6.3 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: –

6.3.1 External Interface Module (M3)

Secrets: The library's inner modules.

Services: Provides the external interface which is the communication link between the client programs and the inner LoSMS modules. [I'm interested in this approach. I wonder if you are just going to be renaming the access programs for the modules that this module uses. In that case, this module wouldn't likely be that useful. We'll have a better idea as you develop your MIS. —SS]

Implemented By: LoSMS

6.3.2 Input Module (M4)

Secrets: The format and structure of the input data.

Services: Gets the inputs from the client, verifies that they are valid, throws an error if the inputs violate any constraints.

Implemented By: LoSMS

6.3.3 Input Conversion Module (M5)

Secrets: The way the inputs are converted to the forms needed for the simplex method solver module.

Services: Converts the inputs to the standard and canonical forms which are needed for the simplex method solver module.

Implemented By: LoSMS

6.3.4 Data Structure Module (M6)

Secrets: The data structure for a matrix (i.e. the simplex method tableau).

Services: Converts the values in the input conversion module to a matrix (2D array) and communicates with the simplex method solver for matrix operations.

Implemented By: LoSMS

6.3.5 Simplex Method Solver (M7)

Secrets: The simplex algorithm implementation for solving maximization and minimization linear programming problems.

Services: Solves maximization and minimization linear programs using the simplex method.

Implemented By: LoSMS

6.3.6 Output Module (M8)

Secrets: The format and structure of the output data.

Services: Outputs the optimal solution of the linear program and the point where it occurs which are obtained from the simplex method solvers.

Implemented By: LoSMS

7 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, M3, M4
R2	M4
R3	M4
R4	M5
R5	M5
R6	M6, M7, M8
R7	M6, M7, M8
R8	M6, M7, M8

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M1
AC2	M3
AC3	M4, M5
AC4	M4, M5
AC5	M6
AC6	M7
AC7	M8

Table 3: Trace Between Anticipated Changes and Modules

[Is it okay that I didn't include M2 (BH Module) in the tables? Since there aren't any BH modules in the hierarchy, I didn't know how to map it with the Rs and the ACs —HZ] [Yes, this is fine. The only modules you need to include in the traceability are the leaf modules. —SS]

[You have two cases where anticipated changes map to more than one module. You should explain why this has occurred. The normal goal is one module, one change. The modules around the input do seem odd. Could the input module have a service (access program) related to conversion? Do you really need a separate module. Maybe in this case you feel like you have two secrets, but if the two secrets always happen together, then module consolidation might make sense. As you write your MIS, it should become clear whether you really have two modules. —SS]

8 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.

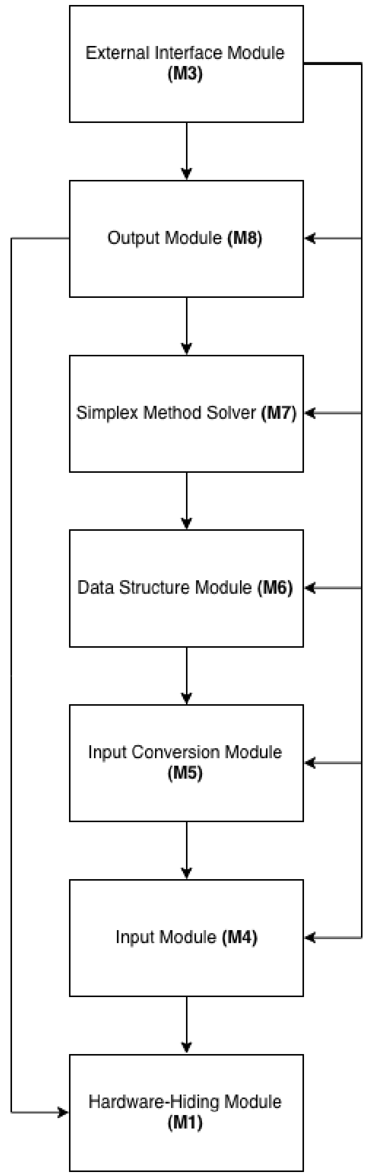

Figure 1: Use Hierarchy Among Modules

[You indicate that the external interface modules uses the output module with two different arrows. This isn't wrong, but it is confusing. Does the output module really use the simplex method solver? It seems tome that the external interface module would use the simplex solver and then relay the results to the output module. Will the output module actually call the simplex solver itself? I would think that the input module would use the data structure module, not the other way around? —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.

Hanane Zlitni. Commonality analysis of a library of simplex method solvers, 2018. URL https://github.com/hananezlitni/HZ-CAS741-Project/blob/master/docs/SRS/CA.pdf.