Module Guide: Library of ODE Solvers

Paul Aoanan

December 17, 2017

1 Revision History

Date	Version	Notes
December 17, 2017	1.0	Initial 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 assumptions and constraints on the input parameters.

AC3: Moved to UC7. [There really isn't a way to hide this in the design. If the language changes, everything will have to be rewritten. —SS] [Moved to unlikely changes. —PA]

AC4: The implementation of Euler's Method.

AC5: The implementation of the Trapezoidal Method.

AC6: The implementation of Heun's Method.

AC7: The implementation of the Runge-Kutta Method.

AC8: The format of the initial input data.

AC9: The format of the input parameters.

AC10: The format of the 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 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: Function Call with correctly formatted arguments, Output: File, Memory, and/or Screen).

UC2: There will always be an expected program call with the assumed arguments to the library's modules.

UC3: Moved to AC8.

UC4: Moved to AC9.

UC5: Moved to AC10.

UC6: The goal of the system is to calculate the value of y_k in an ODE Initial Value Problem.

UC7: The specific language which the software is written in.

[Formats are easy to hide. You should classify these as likely (anticipated) changes —SS] [Moved to Anticipated Changes. —PA]

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: External Interface Module

M3: Equation String Parser Module

M4: Output Format Module

M5: Euler's Method Module

M6: Trapezoidal Method Module

M7: Heun's Method Module

M8: Runge-Kutta's Method Module

Note that the M1 is a commonly used module and is already implemented by the operating system. It will not be reimplemented. Similarly, M3 and M4 are already available in Matlab and will not be reimplemented.

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	External Interface Module Euler's Method Module Trapezoidal Method Module Heun's Method Module Runge-Kutta's Method Module
Software Decision Module	Equation String Parser Module Output Format Module

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

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: –

6.2.1 External Interface Module (M2)

Secrets: The interface exposed to the external world or wrapper function.

Services: Provides the external interface to the library for use by a driver program. The external interface module handles the external interface of LODES validating the inputs sending them downstream.

Implemented By: LODES

6.2.2 Euler's Method Module (M5)

Secrets: The implementation of the Euler's Method algorithm.

Services: Provides the calculation to obtain y_k using Euler's method.

Implemented By: LODES

6.2.3 Trapezoidal Method Module (M6)

Secrets: The implementation of the Trapezoidal Method algorithm.

Services: Provides the calculation to obtain y_k using the trapezoidal method.

Implemented By: LODES

6.2.4 Heun's Method Module (M7)

Secrets: The implementation of Heun's Method algorithm.

Services: Provides the calculation to obtain y_k using Heun's method.

Implemented By: LODES

6.2.5 Runge-Kutta's Method Module (M8)

Secrets: The implementation of the Runge-Kutta 4 Method algorithm.

Services: Provides the calculation to obtain y_k using the Runge-Kutta 4 method.

Implemented By: LODES

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 Equation String Parser Module (M3)

Secrets: The implementation of a string parser algorithm.

Services: Parses the equation string input and returns a valid and manipulatable ODE equation.

Implemented By: Matlab

6.3.2 Output Format Module (M4)

Secrets: The implementation of sending the desired output to its destination.

Services: Sends the output to be used by the outside environment or driver program.

Implemented By: Matlab

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**
IM1	M5
IM2	M6
IM3	M7
IM4	M8
C1	M2
C2	M3
C3	M5, M6, M7, M8
O1	M1, M4
O2	M1, M4
O3	M5, M6, M7, M8
O4	M5, M6, M7, M8, M4
O5	M5, M6, M7, M8, M4

Table 2: Trace Between Requirements and Modules

AC	**Modules**
AC1	M1
AC2	M2
AC3	M2
AC4	M5
AC5	M6
AC6	M7
AC7	M8

Table 3: Trace Between Anticipated Changes and Modules

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.

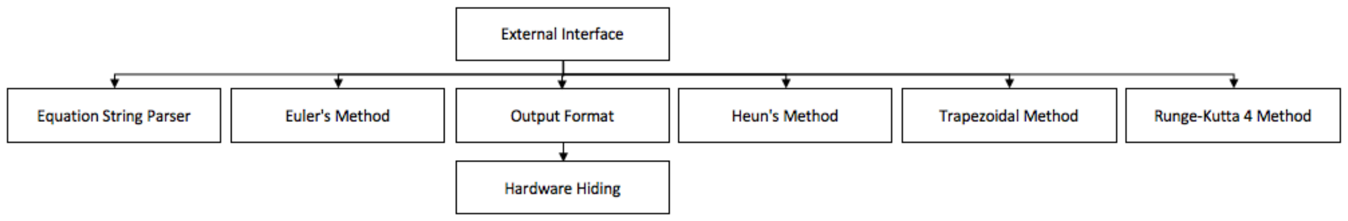

Figure 1: Use hierarchy among modules

[If module are on the same level of the hierarchy, the picture should reflect this. That is, the Output Format module should be on the same line as the other modules. —SS] [Figure has been modified. —PA]

[Good start for the design. Remember that you may have to modify the design as you work through the MIS and gain a deeper understanding of how your modules interact. —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.