Module Guide for Aorta

Author Name

October 2, 2022

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
Aorta	Aorta geometry reconstruction
UC	Unlikely Change
DAS	Descending Aorta Segmenter
AAS	Ascending Aorta Segmenter
ASS	Aorta Sagital Segmenter
[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.

...

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: Hardware-Hiding Module

M2: Input Format Module

M3: Input Parameter Module

M4: Control Module

M5: Display Module

M6: Aorta Segmentation Module

M7: Descending Aorta Segmentation Module

M8: Ascending Aorta Segmentation Module

M9: Aorta Sagital Segmentation Module

...

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	Input Parameter Module Control Module Display Module ? ? ? ? ?
Software Decision Module	Aorta Segmentation Module Descending Aorta Segmentation Module Ascending Aorta Segmentation Module Aorta Sagital Segmentation 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. 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. *Aorta* means the module will be implemented by the Aorta 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 Input Format Module (M2)

Secrets: The format and structure of the input data.

Services: Converts the input data into the data structure used by the input parameters module.

Implemented By: Aorta

Type of Module: [Abstract Data Type] [Information to include for leaf modules in the decomposition by secrets tree.]

7.2.2 Input Parameter Module (M3)

Secrets: The data structure for input parameters, how the values are input and how the values are verified. The load and verify secrets are isolated to their own access programs.

Services: Gets input from user (including the starting slice of the segmentation, the center of the circle , segmentation factor that is used to determined a good segmentation slice), stores input and verifies that the input parameters comply with physical and software constraints. Throws an error if a parameter violates a physical constraint. Throws a warning if a parameter violates a software constraint. Stored parameters can be read individually, but write access is only to redefine the entire set of inputs.

Implemented By: Aorta

7.2.3 Control Module (M4)

Secrets: The algorithm for coordinating the running of the program.

Services: Provides the main program.

Implemented By: Aorta

7.2.4 Display Module (M5)

Secrets: The methods by which it displays input data and output data, along with any other info to the screen.

Services: Displays the program.

Implemented By: Aorta

7.2.5 Etc.

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 Aorta Segmentation Module (M??)

Secrets: The abstract class for an Aorta segmenter.

Services: Includes the common attributes and function signatures of any Aorta segmenter.

Use case: When declare a new segmenter class, AS is used as the base class for reference to know what an Aorta segmenter must have. It must have an original image, a segmented image and a *begin_segmentation* method public to the control module to start the segmentation.

7.3.2 Descending Aorta Segmentation Module (M7)

Secrets: The algorithm to segment the descending Aorta from the rest of the CT scan image.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Use case: Pass in the segmentation seeds and *original image* and call AortaDescendingSegmenter constructor to initialize an object. Call the method *prepared_image* from DAS objecet to crop the image into desired size. Finally, call *begin_segmentation* function and read *segmented_image* to get the segmented result, a SITK image object.

7.3.3 Ascending Aorta Segmentation Module (M8)

Secrets: The algorithm to segment the ascending Aorta from the rest of the CT scan image.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Use case: Pass in the segmentation seeds and *segmented image* from previous step and call AortaAscendingSegmenter constructor to initialize an object. Finally, call *begin_segmentation* function and read *segmented_image* to get the segmented result, a SITK image object.

7.3.4 Aorta Sagital Segmentation Module (M9)

Secrets: The algorithm to segment the Aorta from sagital plane, this can segment out the curve part of the Aorta.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Use case: Pass in the segmentation seeds and *segmented image* from previous step and call AortaSagitalSegmenter constructor to initialize an object. Finally, call *begin_segmentation* function and read *segmented_image* to get the segmented result, a SITK image object.

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, M3, M4
R2	M2, M3
R3	M??
R4	M??, M4
R5	M??, M??, M4, M??, M??, M??
R6	M??, M??, M4, M??, M??, M??
R7	M??, M??, M4, M??, M??
R8	M??, M??, M4, M??, M??
R9	M??
R10	M??, M??, M4
R11	M??, M??, M??, M4

Table 2: Trace Between Requirements and Modules

AC	**Modules**
AC1	M1
AC2	M2
AC??	M3
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M4
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.

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.