CAS 741: Module Guide

Aqueous Speciation Diagram Generator

Steven Palmer
`palmes4`

December 15, 2017

Revision History

Table 1: Revision History

Date	Developer(s)	Change
11.02.2017	S. Palmer	First revision of document
11.08.2017	S. Palmer	Revision 0

Contents

List of Tables

List of Figures

1 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 2 lists the anticipated and unlikely changes of the software requirements. Section 3 summarizes the module decomposition that was constructed according to the likely changes. Section 4 specifies the connections between the software requirements and the modules. Section 5 gives a detailed description of the modules. Section 6 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 7 describes the use relation between modules.

2 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 2.1, and unlikely changes are listed in Section 2.2.

2.1 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 exposed interface of the library.

AC3: The structure of the stored chemical system.

AC4: The structure and format of the input data with respect to chemical equations.

AC5: The structure and format of the input data with respect to chemical species.

AC6: How the input data is transformed into speciation data.

AC7: How the input is converted to a form usable by the non-linear solver.

AC8: The non-linear solver algorithm.

AC9: The plotting algorithm.

2.2 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).

3 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 2. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Hardware-Hiding

M2: External Interface

M3: Chemical System

M4: Species

M5: Reaction

M6: Input Conversion

M7: Calculation

M8: Non-linear Solver

M9: Plotting

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	External Interface Chemical System Species Reaction Input Conversion Calculation
Software Decision	Non-linear Solver Plotting

Table 2: Module Hierarchy

4 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 3.

5 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.

5.1 Hardware Hiding (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

5.2 Behaviour-Hiding

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: –

5.2.1 External Interface (M2)

Secrets: The exposed interface of the library.

Services: Provides the external interface of the library that can by used by programs.

Implemented By: SpecGen

5.2.2 Chemical System (M3)

Secrets: The structure of the stored chemical system.

Services: Provides the data structure for storing the input data. The data stored includes allof the chemical reactions and species associated with the system, as well as element totals.

Implemented By: SpecGen

5.2.3 Equation (M5)

Secrets: The structure and format of the input data with respect to chemical equations.

Services: Parses input (chemical equations) and converts to a format used by M3.

Implemented By: SpecGen

5.2.4 Species (M4)

Secrets: The structure and format of the input data with respect to chemical species.

Services: Parses input (chemical species) and converts to a format used by M3.

Implemented By: SpecGen

5.2.5 Calculation (M7)

Secrets: How the input data is transformed into speciation data.

Services: Calculates equilibrium concentrations of all species over a pH range.

Implemented By: SpecGen

5.2.6 Input Conversion (M6)

Secrets: How the input is converted to a form usable by the non-linear solver.

Services: Converts input data into a format that can be passed to the non-linear solver.

Implemented By: SpecGen

[What if instead you had a module that stored the calculation output and you could view (return/output) the data in whatever form you needed? I think I like your design better, but it depends on how complex the data is that you'll be passing around as inputs and outputs to your different modules. It looks like none of the modules will have state, except maybe M3. It will be easier to judge once your MIS is done. Great initial design. I like the external interface idea. —SS]

5.3 Software Decision

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: –

5.3.1 Non-linear Solver (M8)

Secrets: The non-linear solver algorithm.

Services: Calculates a solution for a non-linear system of equations.

Implemented By: Python

5.3.2 Plotting (M9)

Secrets: The plotting algorithm.

Services: Produces plots (as image files).

Implemented By: Python

6 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	M2, M3, M5, M4
R2	M3, M5, M4
R3	M3
R4	M2, M7, M6, M8
R5	M1, M2, M9

Table 3: Trace Between Requirements and Modules

AC	Modules
AC1	M1
AC2	M2
AC3	M3
AC5	M4
AC4	M5
AC6	M7
AC7	M6
AC8	M8
AC9	M9

Table 4: Trace Between Anticipated Changes and Modules

7 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.

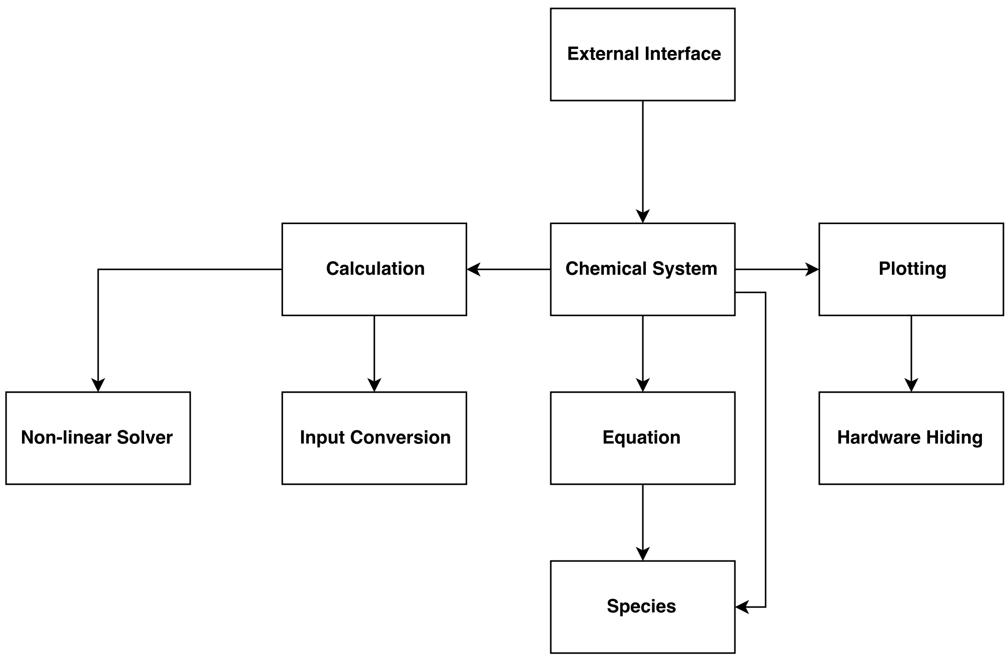

Figure 1: Use hierarchy among modules

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.