Module Guide: Project Title

Author Name

November 2, 2017

1 Revision History

Date	Version	Notes
November 3	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 laid 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 **??** 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 algorithm used to solve the given IVP.

AC2: The implementation of the algorithms used to solve the IVP.

...

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: Output format (generated code).

UC2: The way the driver invokes the software.

...

4 Binding Times

This section will demonstrate the difference between binding times via the oft-used power example.

```
let square x = x * x
let rec power n x =
  if n = 0 then 1
  else if n mod 2 = 0 then square (power (n/2) x)
  else x * (power (n-1) x)
(* val power : int -> int -> int = <fun> *)
```

(The comment below the code shows the reply of the MetaOCaml toplevel.)

Suppose our program has to compute x^7 many times. We may define

```
let power7 x = power 7 x
```

to name and share the partial application.

In MetaOCaml, we may also specialize the power function to a particular value n, obtaining the code which will later receive x and compute x^n. We re-write power n x annotating expressions as computed 'now' (when n is known) or 'later' (when x is given).

```
let rec spower n x =
  if n = 0 then .<1>.
  else if n mod 2 = 0 then .<square .~(spower (n/2) x)>.
  else .<.~x * .~(spower (n-1) x)>.;;
(* val spower : int -> int code -> int code = <fun> *)
```

The two annotations, or staging constructs, are brackets .< e >. and escape .~e . Brackets .< e >. 'quasi-quote' the expression e, annotating it as computed later. Escape .~e, which must be used within brackets, tells that e is computed now but produces the result for later. That result, the code, is spliced-in into the containing bracket. The inferred type of spower is different. The result is no longer an int, but int code – the code of expressions that compute an int. The type of spower spells out which argument is received now, and which later: the future-stage argument has the code type. To specialize spower to 7, we define

```
let spower7_code = .<fun x -> .~(spower 7 .<x>.)>.;;
(*
val spower7_code : (int -> int) code = .<
  fun x_1 ->
    (x_1 *
      (((* cross-stage persistent value (id: square) *))
        (x_1 * (((* cross-stage persistent value (id: square) *)) (x_1 * 1))))
    )
  >.
*)
```

and obtain the code of a function that will receive x and return x^7. Code, even of functions, can be printed, which is what MetaOCaml toplevel just did. The print-out contains so-called cross-stage persistent values, or CSP, which 'quote' present-stage values such as square to be used later. One may think of CSP as references to 'external libraries' – only in our case the program acts as a library for the code it generates. If square were defined in a separate file, say, sq.ml, then its occurrence in spower7_code would have been printed simply as Sq.square.

To use thus specialized x^7 now, in our code, we should 'run' spower7_code, applying the function Runcode.run : 'a code -> 'a. The function compiles the code and links it back to our program.

```
open Runcode
let spower7 = !. spower7_code
(* val spower7 : int -> int = <fun> *)
```

The specialized spower7 has the same type as the partially applied power7 above. They behave identically. However, power7 x will do the recursion on n, checking n's parity, etc. In contrast, the specialized spower7 has no recursion (as can be seen from spower7_code). All operations on n have been done when the spower7_code was computed, producing the straight-line code spower7 that only multiplies x.

MetaOCaml supports an arbitrary number of later stages, letting us write not only code generators but also generators of code generators, etc.

4.1 Compile-Time

N/A. Nothing will be bound while the generator is compiled along with the driver program.

4.2 Generation Time

The ODE, range, step size, and algorithm are known at generation time. Since these will not change, the IVP can be solved at this time.

4.3 Run-Time

The function that will return values for any input within the specified interval will execute at run-time.

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 1.

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 (M??)

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 Input Format Module (M??)

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: [Your Program Name Here]

6.2.2 Etc.

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 Etc.

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

Table 1: Trace Between Requirements and Modules

AC	**Modules**
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 2: 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

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.