Module Guide: Project Title

Author Name

November 23, 2017

1 Revision History

Date	Version	Notes
November 3	1.0	Initial Draft
November 8	1.1	Processed feedback, continued writing

Contents

2 Introduction

[Reusing this introduction does not make any sense for your project. Please replace with an introduction suitable for your design document. —SS]

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 **??** [missing this section —SS] 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.

[These ideas still apply, but rather than separate sections, maybe you could just summarize this information in your revised introduction. —SS]

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. Rather than dividing the program into modules, a decomposition by binding times is given. Such a decomposition is more appropriate when multi-stage programming (MSP) is employed, something Parnas (1972) was unaware of at that time, due to Multi-stage programming being much newer.

Consider the following example for calculating any x^n:

```
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 interactive prompt, called the "toplevel", which contains the types of the power function.)

Assuming x^7 is a common operation, it would be useful to define

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

to give the *partial application* a name and use it throughout the code. Partial application means that when giving a function some but not all the arguments it expects, one creates a new function that only expects the remaining arguments. In the case of power7, the resulting function type will be int -> int.

In MetaOCaml, one can also create a specialized instance of the power function for a particular value of n. However, in the case of MetaOCaml, one can generate code which accepts any x and computes x^n. To do this, the power function must be annotated such that computations which can be performed at generation time are distinguished from those that must be performed at run time. Because information such as n is available at right away and certain not to change in the future, any computation that only depends on n may be performed when the code is being generated. In contrast, information such as x is not known at generation time and subject to change, so any computation that needs it will have to be postponed until the generated code is actually run (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> *)
```

Two annotations, or staging constructs, specific to MetaOCaml appear in the code above. These 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 inserted (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, one can define

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

and obtain the code of a function that expects any x and returns x^7. Code, even of functions, can be printed, which is what MetaOCaml toplevel (interactive prompt) just did. The printout 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 such as `sq.ml`, then its occurrence in spower7_code would have been printed as Sq.square.

To use the 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. Runcode.run is aliased to the prefix operation (!.):

```
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 one write not only code generators but also generators of code generators, etc.)

[I like the above summary. I would like you to use the same systematic steps for presenting your code. —SS]

4.1 Compile-Time

As the name implies, this is when the code is compiled. Code generation could occur at this time, but this will not happen for the RK generator.

The generator and its RK methods will be completely defined at this time. There are no inputs to it at this time, because it is effectively a library.

4.2 Generation Time

This is part of the run-time of the driver program, but from the point of view of this library, separating the moment the code for a particular IVP is generated and the moment(s) it is

executed is useful.

The ODE, interval, step size, desired RK method and initial value are known at generation time. Since these will not change when looking for solutions given a particular point on the interval for which the algorithm is solving, the IVP can be solved at this time.

The type of the ODE is `float -> float array -> float array`. The types for the range, step size and initial value are `(float * float)`, `float`, and `float array` respectively. The type of the function that is returned should be (close to) `(float -> float) code`.

4.3 Run-Time

This is also part of the driver program's run-time, but from the point of view of the RK generator, this is (one of) the moment(s) when the generated code is run.

The function that will return values for any input within the specified interval will execute at run-time. As shown in the previous subsection, the input will be a `float`, and the return type is another `float`.

5 Connection Between Requirements and Design

N/A. While the commonality analysis has clear implications, the requirements are not explicitly recorded anywhere.

6 Module Decomposition

N/A. The program is broken down by binding times, rather than modules, due to the use of multi-stage programming via MetaOCaml.

7 Traceability Matrix

N/A.

8 Use Hierarchy Between Modules

N/A.

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

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.