SE 2AA4 Winter 2007

04 Software Modules

William M. Farmer

Department of Computing and Software
McMaster University

13 February 2007

What is a Software Module?

- Modules are relatively self-contained systems that can be combined to make large systems (Parnas).
- Design is often the assembly of many previously designed modules (Parnas).
  - Modules are interconnectable and interchangeable parts.
  - Modules can be designed, implemented, tested, and changed independently.
- A software module is a cohesive collection of data and procedures that provides a set of services to other client modules.
  - Programs and procedures are usually not modules.
  - Modules usually have state.

Components of a Module

A software module has two components:

1. An interface that enables the module's clients to use the services the module provides.
2. An implementation of the interface that provides the services offered by the module.

The Module Interface

- A module's interface can be viewed in various ways:
  - As a set of services.
  - As a contract between the module and its clients.
  - As a language for using the module's services.
- The interface is exported by the module and imported by the module's clients.
- An interface describes the data and procedures that provide access to the services of the module.

The Module Implementation

- A module's implementation is an implementation of the module's interface.
- The implementation is hidden from other modules.
- The interface data and procedures are implemented together and may share data structures.
- The implementation may utilize the services offered by other modules.

Examples of Modules

- Record.
  - Consists of only data.
  - Has state but no behavior.
- Collection of related procedures
  - Has behavior but no state.
- Object.
  - Consists of data (fields) and procedures (methods).
  - Has state and behavior.
- Abstract data structure.
  - Consists of a collection of constructors, selectors, and mutators.
- Abstract data type (ADT).
  - Consists of a collection of abstract data structures and a collection of procedures that can be applied to them.

An Example Interface in C for a Stack

```
int stack_top();                    /* selector          */
int stack_height();                 /* selector          */
void stack_push(int element);       /* mutator           */
void stack_pop();                   /* mutator           */
void stack_print();                 /* print procedure */
```

The Principles of Modular Design (1)

1. Separation of Concerns.
   - Different parts of the problem are handled by different modules (horizontal decomposition)
   - What (i.e., interface) is separated from how (i.e., implementation) (vertical decomposition)
2. Abstraction.
   - The services that other modules need are expressed in the interface.
   - The implementation details that other modules do not need are left out of the interface.
3. Anticipation of Change: Information Hiding Method
   - Implementation details and design decisions likely to change are hidden from other modules (design for change).
   - Each module's implementation is a treated as a "secret" (Parnas).

The Principles of Modular Design (2)

4. Generality.
   - Modules are designed to solve general rather than specific problems.
   - Modules are intended to be reused and to be useful to as wide a range of clients as possible.
5. Generality: Little Languages Method
   - The interface is designed as a language that can solve a family of problems instead of just a single problem.
   - More abstract languages are defined in terms of more concrete languages.

Hallmarks of a Good Module

- The module has high cohesion (i.e., its components are closely related).
- The module has low coupling (i.e., it does not strongly depend on other modules).
- The interface is small and orthogonal.
- The interface language is highly expressive.
- What is likely to change in the module is hidden from other modules.
- The data structures of the implementation are accessible only via the interface procedures.

Module Extensions

- Let M_i be a module with interface I_i for $i = 1, 2$.
- M_2 is an extension of M_1 (and M_1 is a submodule of M_2) if:
  1. I_2 is an extension of I_1 (i.e., I_2 includes all the components of I_1).
  2. The implementation of M_2 is an extension of the implementation of M_1.
- If M_2 is an extension of M_1, then M_2 provides all the services that M_1 provides.

Conservative Extensions

- Let M_i be a module with interface I_i for $i = 1, 2$.
- M_2 is a conservative extension of M_1 if:
  1. M_2 is an extension of M_1.
  2. M_2 has no access to the implementation of M_1 outside of I_1.
- If M_2 is a conservative extension of M_1, then:
  - The language of I_2 will be an enrichment of the language of I_1.
  - I_2 need not be small and orthogonal.
  - If the implementation of M_1 is changed without violating the specification of I_1, then the behavior of M_2 will not violate the specification of I_2.

Definitional Extensions

- Let M_i be a module with interface I_i for $i = 1, 2$.
- M_2 is a definitional extension of M_1 if:
  1. M_2 is a conservative extension of M_1.
  2. The state of M_2 is always equal to the state of M_1.
- If M_2 is a definitional extension of M_1, then:
  - The components in I_2 but not in I_1 will be defined in terms of the components of I_1.
  - The components in I_2 but not in I_1 can be "eliminated" by replacing them with code written using only the components in I_1.

Modules in the C Programming Language

- C does not directly support modules, but modules can be implemented in C using C's independent compilation mechanism.
- A module interface is expressed as a C header file (e.g., `StackPlus.h`).
  - The header file contains declarations of the types, variables, constants, and procedure signatures (prototypes) that are to be exported.
  - Note: Header files are often used to contain macros, constants, and types that are used by several code files.
- A module implementation is expressed as a C code file (e.g., `StackPlus.c`).
- When compiling with `gcc`, use the following options:
  - `-ansi -pedantic` to conform to ANSI C.
  - `-Wall` to list all warnings.

Header Files as Module Interfaces

- Required module interfaces are imported with `#include`.
  - Example: `#include "Stack.h"`
- Interface types are declared with C type declarations.
  - Example: `typedef int Weight;`
- Interface constants are declared with `#define` or with C constant variable declarations.
  - Example: `#define TRUE 1`
  - Example: `const double PI = 3.14159265358979;`
- The signatures of interface procedures are declared with C function prototypes.
  - Example: `void stack_push(int element);`
- External variables can be declared, but they are not recommended.
- C function definitions should never be in header files.

Code Files as Module Implementations

- Required module interfaces are imported with `#include`.
  - Example: `#include "Stack.h"`
  - All external objects should be components of module interfaces (and thus there should be no use of `extern`).
- Interface procedures are implemented by C function definitions in the code file.
- All local types, constants, variables and procedures are declared in the code file.
  - Local variables and procedures should be marked with the `static` attribute to restrict their scope to the file they are declared in.
  - Example: `static int x1;`

Modules in the Java Programming Language

- A Java object is a module:
  - The interface is the object's set of public fields and methods.
  - The implementation is the object's entire set of fields and methods.
- A Java class is a module:
  - The interface is the object's set of public static fields and methods.
  - The implementation is the object's entire set of static fields and methods.
- A Java class C is also a module specification:
  - Each instance of C is an object (module) of the type of C.

Module Extensions in Java

- Let C_2 be a subclass of C_1 that does not override any of the methods of C_1.
- C_2 specifies a subset of the set of modules specified by C_1.
- An instance O_2 of C_2 is a module extension of an instance O_1 of C_1 (and O_1 a submodule of O_2).

Java Abstract Classes as Module Interfaces

- An abstract method is a method without an implementation.
- An abstract class is a class marked as `abstract` that cannot be instantiated but can be subclassed.
  - A class with abstract methods must be an abstract class.
- An abstract class can be used as a class specification.
- An abstract class C with only abstract methods can be used as a module interface.
  - An implementation of C as a module interface is any instance of a subclass of C.

Java Interfaces as Module Interfaces

- A Java interface consists of a set of public constant and method declarations.
  - The interface name becomes a new reference type.
  - An interface has no implementation at all.
  - An interface cannot be instantiated.
- A class implements an interface by defining the interfaces's methods.
  - The constants of the interface are inherited.
  - A class can simultaneously implement several interfaces.
- Interfaces are good for recording similarities between unrelated classes.
- Key benefit of interfaces: A variable of an interface type *I* may be bound to any object whose class implements *I*.
- Viewed as a module interface, an implementation of a Java interface *I* is any instance of any class that implements *I*.