Modularity

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 4

1

Goals of this Lecture

- Help you learn how to:
  - Create high quality modules in C
- Why?
  - Abstraction is a powerful (only?) technique available for understanding large, complex systems
  - A power programmer knows how to find the abstractions in a large program
  - A power programmer knows how to convey a large program's abstractions via its modularity

2

Module Design Heuristics

- A well-designed module:
  - (1) Separates interface and implementation
  - (2) Encapsulates data
  - (3) Manages resources consistently
  - (4) Is consistent
  - (5) Has a minimal interface
  - (6) Reports errors to clients
  - (7) Establishes contracts
  - (8) Has strong cohesion
  - (9) Has weak coupling
- Let's consider one at a time…

Interfaces

(1) A well-designed module separates interface and implementation

- Why?
  - Hides implementation details from clients
    - Thus facilitating abstraction
  - Also allows separate compilation of each implementation
    - Thus facilitating partial builds

Interface Example 1

- Stack: A stack whose items are strings
  - Data structure
    - Linked list
  - Algorithms
    - new: Create a new Stack object and return it
    - free: Free the given Stack object
    - push: Push the given string onto the given Stack object
    - top: Return the top item of the given Stack object
    - pop: Pop a string from the given Stack object and discard it
    - isEmpty: Return 1 (TRUE) iff the given Stack object is empty

5

Interfaces Example 1

- Stack (version 1)

```
/* stack.c */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
void  Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

```
/* client.c */

#include "stack.c"

/* Use the functions
defined in stack.c. */
```

- Stack module consists of one file (stack.c); no interface
- Problem: Change stack.c => must rebuild stack.c ***and client***
- Problem: Client "sees" Stack function definitions; poor abstraction

6

Interfaces Example 1

- Stack (version 2)

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

- Stack module consists of two files:
  (1) stack.h (the interface) declares functions and defines data structures

Interfaces Example 1

- Stack (version 2)

```
/* stack.c */

#include "stack.h"

struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
void  Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

(2) stack.c (the implementation) defines functions
- #includes stack.h so
  - Compiler can check consistency of function declarations and definitions
  - Functions have access to data structures

Interfaces Example 1

- Stack (version 2)

```
/* client.c */

#include "stack.h"

/* Use the functions declared in stack.h. */
```

- Client #includes only the interface
- Change stack.c => must rebuild stack.c, ***but not the client***
- Client does not “see” Stack function definitions; better abstraction

Interface Example 2

- string (also recall Str from Assignment 2)

```
/* string.h */

size_t strlen(const char *s);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strcat(char *dest, const char *src);
char  *strncat(char *dest, const char *src, size_t n);
char  *strcmp(const char *s, const char *t);
char  *strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
…
```

Interface Example 3

- stdio (from C90, vastly simplified)

```
/* stdio.h */

struct FILE {
   int cnt;    /* characters left */
   char *ptr;  /* next character position */
   char *base; /* location of buffer */
   int flag;   /* mode of file access */
   int fd;     /* file descriptor */
};

#define OPEN_MAX 20
FILE _iob[OPEN_MAX];

#define stdin  (&_iob[0]);
#define stdout (&_iob[1]);
#define stderr (&_iob[2]);
…
```

Don't be concerned with details

11

Interface Example 3

- stdio (cont.)

```
…
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);

int   fgetc(FILE *f);
int   getc(FILE *f);
int   getchar(void);

int   putc(int c, FILE *f);
int   putchar(int c);

int   fscanf(FILE *f, const char *format, …);
int   scanf(const char *format, …);

int   fprintf(FILE *f, const char *format, …);
int   printf(const char *format, …);
…
```

12

Encapsulation

(2) A well-designed module encapsulates data

- An interface should hide implementation details
- A module should use its functions to encapsulate its data
- A module should not allow clients to manipulate the data directly

- Why?
  - Clarity: Encourages abstraction
  - Security: Clients cannot corrupt object by changing its data in unintended ways
  - Flexibility: Allows implementation to change – even the data structure – without affecting clients

13

Encapsulation Example 1

- Stack (version 1)

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Structure type definitions in .h file

- That's bad
- Interface reveals how Stack object is implemented (e.g., as a linked list)
- Client can access/change data directly; could corrupt object

14

Encapsulation Example 1

- Stack (version 2)

```
/* stack.h */

struct Stack;

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Move definition of struct Node to implementation; clients need not know about it

Place ***declaration*** of struct Stack in interface; move ***definition*** to implementation

- That's better
- Interface does not reveal how Stack object is implemented
- Client cannot access data directly

15

Encapsulation Example 1

- Stack (version 3)

```
/* stack.h */

typedef struct Stack * Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

Opaque pointer

- That's better still
- Interface provides "Stack_T" abbreviation for client
- Interface encourages client to view a Stack as an object, not as a (pointer to a) structure
- Client still cannot access data directly; data is "opaque" to the client

16

Encapsulation Example 2

- string
  - "Stateless" module
  - Has no state to encapsulate!

Encapsulation Example 3

- stdio

```
/* stdio.h */

struct FILE {
   int cnt;     /* characters left */
   char *ptr;   /* next character position */
   char *base;  /* location of buffer */
   int flag;    /* mode of file access */
   int fd;      /* file descriptor */
};
…
```

Structure type definition in .h file

  - Violates the heuristic
  - Programmers can access data directly
    - Can corrupt the FILE object
    - Can write non-portable code
  - But the functions are well documented, so
    - Few programmers examine stdio.h
    - Few programmers are tempted to access the data directly

Resources

(3) A well-designed module manages resources consistently

- A module should free a resource if and only if the module has allocated that resource
- Examples
  - Object allocates memory <=> object frees memory
  - Object opens file <=> object closes file

- Why?
  - Allocating and freeing resources at different levels is error-prone
    - Forget to free memory => memory leak
    - Forget to allocate memory => dangling pointer, seg fault
    - Forget to close file => inefficient use of a limited resource
    - Forget to open file => dangling pointer, seg fault

19

Resources Example 1

- Stack: Who allocates and frees the strings?
  - Reasonable options:
    (1) Client allocates and frees strings
      - **`Stack_push()`** does not create copy of given string
      - **`Stack_pop()`** does not free the popped string
      - **`Stack_free()`** does not free remaining strings

    (2) Stack object allocates and frees strings
      - **`Stack_push()`** creates copy of given string
      - **`Stack_pop()`** frees the popped string
      - **`Stack_free()`** frees all remaining strings
  - Our choice: (1), but debatable
  - Unreasonable options:
    - Client allocates strings, Stack object frees strings
    - Stack object allocates strings, client frees strings

20

Resources Examples 2, 3

- string
  - Stateless module
  - Has no resources to manage!
- stdio
  - **`fopen()`** allocates memory, uses file descriptor
  - **`fclose()`** frees memory, releases file descriptor

21

SymTable Aside

- Consider SymTable (from Assignment 3)…
- Who allocates and frees the key strings?
- Reasonable options:
  - (1) Client allocates and frees strings
    - **`SymTable_put()`** does not create copy of given string
    - **`SymTable_remove()`** does not free the string
    - **`SymTable_free()`** does not free remaining strings
  - (2) SymTable object allocates and frees strings
    - **`SymTable_put()`** creates copy of given string
    - **`SymTable_remove()`** frees the string
    - **`SymTable_free()`** frees all remaining strings
- Our choice: (2)
  - With option (1) client could corrupt the SymTable object (as described last lecture)

22

Passing Resource Ownership

- Passing resource ownership
  - Violations of the heuristic should be noted explicitly in function comments

```
somefile.h

…

void *f(void);
/* …
   This function allocates memory for
   the returned object.  You (the caller)
   own that memory, and so are responsible
   for freeing it when you no longer
   need it. */

…
```


Consistency

(4) A well-designed module is consistent
- A function’s name should indicate its module
  - Facilitates maintenance programming; programmer can find functions more quickly
  - Reduces likelihood of name collisions (from different programmers, different software vendors, etc.)
- A module’s functions should use a consistent parameter order
  - Facilitates writing client code

Consistency Examples

- Stack
  - (+) Each function name begins with "Stack_"
  - (+) First parameter identifies Stack object
- string
  - (+) Each function name begins with "str"
  - (+) Destination string parameter comes before source string parameter; mimics assignment
- stdio
  - (-) Some functions begin with "f"; others do not
  - (-) Some functions use first parameter to identify FILE object; others (e.g. **`putc()`**) use a different parameter

25

Minimization

(5) A well-designed module has a minimal interface

- Function declaration should be in a module's interface if and only if:
  - The function is necessary to make objects complete, or
  - The function is very convenient for many clients

- Why?
  - More functions => higher learning costs, higher maintenance costs

26

Minimization Example 1

- Stack

```
/* stack.h */

typedef struct Stack *Stack_T ;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

Necessary

Convenient

- All functions are necessary

27

Minimization Example 1

- Another Stack function?

  `void Stack_clear(Stack_T s);`
  - Pops all items from the Stack object
  - Unnecessary; client can call **`pop()`** repeatedly
  - But could be convenient

- Our decision: No, but debatable

28

Minimization Example 2

- string

```
/* string.h */

size_t strlen(const char *s);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strcat(char *dest, const char *src);
char  *strncat(char *dest, const char *src, size_t n);
char  *strcmp(const char *s, const char *t);
char  *strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);
…
```

Necessary

Convenient

- "n" functions are necessary
- Corresponding "non-n" functions are more convenient; they're also more efficient

Minimization Example 3

- stdio

```
…
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);

int   fgetc(FILE *f);
int   getc(FILE *f);
int   getchar(void);

int   putc(int c, FILE *f);
int   putchar(int c);

int   fscanf(FILE *f, const char *format, …);
int   scanf(const char *format, …);

int   fprintf(FILE *f, const char *format, …);
int   printf(const char *format, …);
…
```

Necessary

Convenient

- Are "convenient" functions convenient enough?

SymTable Aside

- Consider SymTable (from Assignment 3)

  Q: Given **`SymTable_get()`**, is **`SymTable_contains()`** necessary?

  A: Yes. If **`SymTable_get()`** returns NULL, it could be the case that:
  - A binding with the specified key does not exist
  - A binding exists, and its value is NULL
- Other SymTable functions

  ```
  int SymTable_hash(const char *key, int bucketCount);
  ```
  - Used internally
  - Clients should not know about it
    - Would make no sense for linked list implementation
  - Our decision: **Not in interface!!! Should be static**
- C Rule: Any function should be either:
  - Non-static, and declared in the interface
  - Static, and not declared in the interface

31

Reporting Errors

(6) A well-designed module reports errors to clients

- A module should detect errors, but allow clients to handle them
- (But it should do so with some moderation)

- Why?
  - Handling errors in the client provides the most flexibility
    - Module should not assume what error-handling action clients prefer

32

Reporting Errors in C

- C options for detecting errors
  - `if` statement
  - `assert` macro
- C has no Java-style exceptions, try-catch, or throw; so…
- C options for reporting errors to client
  - Set global variable?
    - Easy for client to forget to check
    - Bad for multi-threaded programming
  - Use function return value?
    - Awkward if return value has some other natural purpose
  - Use extra call-by-reference parameter?
    - Awkward for client; must pass additional parameter
  - Call `assert` macro?
    - Kills the client!
- No option is ideal

33

Reporting Errors in C (cont.)

- Our recommendation: Distinguish between…
- User errors
  - Errors made by human user
    - Example: Bad data in stdin
    - Example: Bad command-line argument
  - Errors that "easily could happen"
  - To detect: Use `if` statement
  - To report: Use return value or call-by-reference parameter
- Programmer errors
  - Errors made by a programmer
  - Errors that "never should happen"
    - Example: Call `Stack_pop()` with NULL stack, empty stack
  - To detect and report: Use `assert`
- The distinction sometimes is unclear
  - Example: Write to file fails because disk is full

34

Reporting Errors Example 1

- Stack

```
/* stack.c */

…

void Stack_push(Stack_T s, const char *item) {
   struct Node *p;
   assert(s != NULL);
   p = (struct Node*)malloc(sizeof(struct Node));
   assert(p != NULL);
   p->item = item;
   p->next = s->first;
   s->first = p;
}
```

- Stack functions:
    - Consider invalid parameter to be **programmer** error
    - Consider **`malloc()`** failure to be **programmer** error
    - Detect/report no **user** errors

35

Reporting Errors Examples 2, 3

- string
    - No error detection or reporting
        - Example: NULL parameter to **`strlen()`** => probable seg fault
- stdlib
    - Uses return values to indicate failure
        - Note awkwardness of **`scanf()`**
    - Sets global variable "errno" to indicate cause of failure

36

Establishing Contracts

(7) A well-designed module establishes contracts

- A module should establish contracts with its clients
- Contracts should describe what each function does, esp:
  - Meanings of parameters
  - Valid/invalid parameter values
  - Meaning of return value
  - Side effects

- Why?
  - Establishing contracts facilitates cooperation between multiple programmers on a team
  - Establishing contracts assigns blame to violators
    - Catch errors at the door!
    - Better that the boss yells at the programmer who is your client rather than at you!!!

37

Establishing Contracts in C

- Our recommendation…

- In C, establish contracts via comments in module interface
  - A module's implementation then should enforce the contracts

38

Establishing Contracts Example

- Stack

```
/* stack.h */
…
char *Stack_top(Stack_T s);
/* Return the top item of stack s.
   It is a checked runtime error for s
   to be NULL or empty. */
…
```

- Comment defines contract:
  - Meanings of function's parameters
    - s is the pertinent stack
  - Valid/invalid parameter values
    - s cannot be NULL or empty
  - Meaning of return value
    - The return value is the top item
  - Side effects
    - (None, by default)

39

Strong Cohesion

(8) A well-designed module has strong cohesion
- A module's functions should be strongly related to each other

- Why?
  - Strong cohesion facilitates abstraction

40

Strong Cohesion Examples

- Stack
  - (+) All functions are related to the encapsulated data
- string
  - (+) Most functions are related to string handling
  - (-) Some functions are not related to string handling
    `memcpy()`, **`memmove()`**, **`memcmp()`**, **`memchr()`**, **`memset()`**
  - (+) But those functions are similar to string-handling functions
- stdio
  - (+) Most functions are related to I/O
  - (-) Some functions don't do I/O
    `sprintf()`, **`sscanf()`**
  - (+) But those functions are similar to I/O functions

41

Weak Coupling

(9) A well-designed module has weak coupling
- Module should be weakly connected to other modules in program
- Interaction within modules should be more intense than interaction among modules

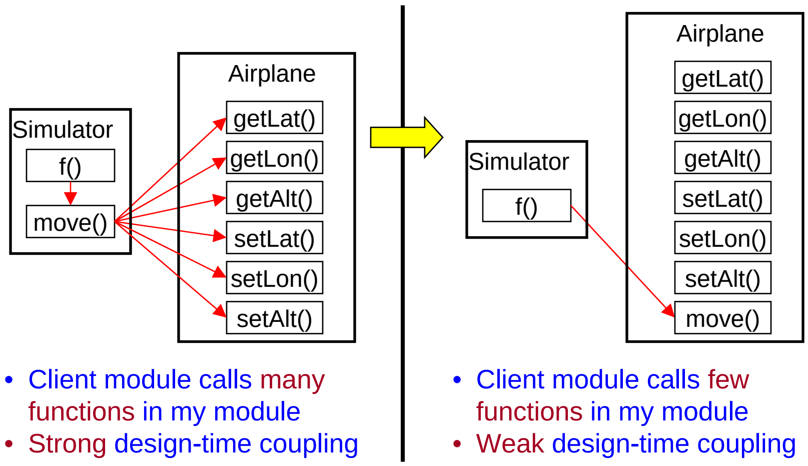
- Why? Theoretical observations
  - Maintenance: Weak coupling makes program easier to modify
  - Reuse: Weak coupling facilitates reuse of modules
- Why? Empirical evidence
  - Empirically, modules that are weakly coupled have fewer bugs

42

Weak Coupling Examples

- Design-time coupling

Function call



- Client module calls many functions in my module
- Strong design-time coupling

- Client module calls few functions in my module
- Weak design-time coupling

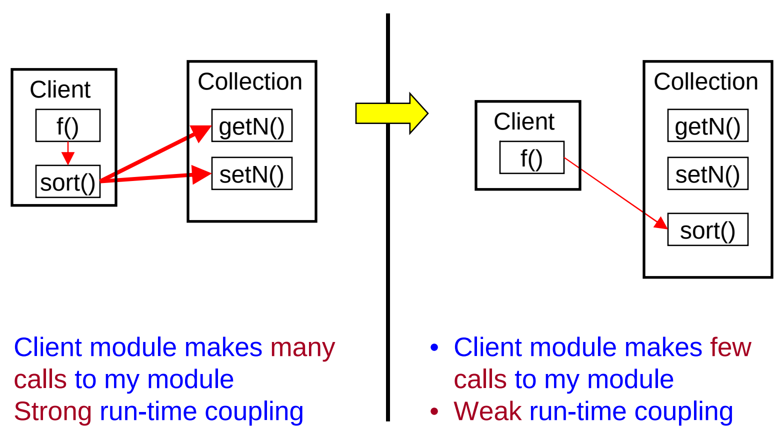
43

Weak Coupling Examples (cont.)

- Run-time coupling

Many function calls

One function call



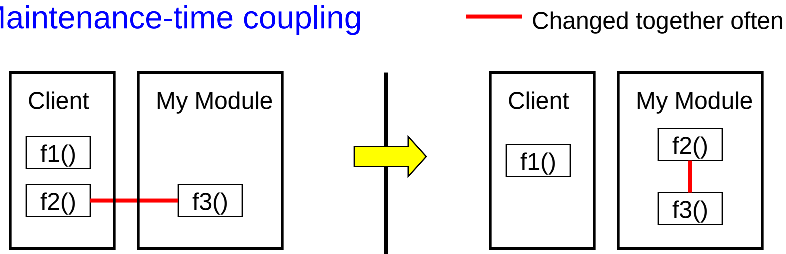
- Client module makes many calls to my module
- Strong run-time coupling

- Client module makes few calls to my module
- Weak run-time coupling

44

Weak Coupling Examples (cont.)

- Maintenance-time coupling



- Maintenance programmer changes client and my module together frequently
- Strong maintenance-time coupling

- Maintenance programmer changes client and my module together infrequently
- Weak maintenance-time coupling

45

Achieving Weak Coupling

- Achieving weak coupling could involve:
  - Moving code from clients to my module (shown)
  - Moving code from my module to clients (not shown)
  - Moving code from clients and my module to a new module (not shown)

46

Summary

- A well-designed module:
  - (1) Separates interface and implementation
  - (2) Encapsulates data
  - (3) Manages resources consistently
  - (4) Is consistent
  - (5) Has a minimal interface
  - (6) Reports errors to clients
  - (7) Establishes contracts
  - (8) Has strong cohesion
  - (9) Has weak coupling

47

The Rest of This Week

- Reading
  - Recommended: K&P chapter 4
- Assignment #3
  - Due Sunday at 9pm
- Midterm exam
  - Class time on Wednesday March 11 (i.e., 10:00-10:50am)
  - In CS large lecture hall (room CS 104), *not* the normal room
  - Open book, open notes, open lecture slides, but no computers
- Midterm exam preparation
  - Practice exams online
    - http://www.cs.princeton.edu/courses/archive/spring09/cos217/exam1prep/
    - See especially Spring 2008 and Fall 2005 (i.e., my exams!)
  - Review session in lecture on Monday March 9

48