Modularity design principles

Module Design Principles

We propose 7 module design principles

And illustrate them with 4 examples

- List, string, stdio, SymTable

Stack Module

List module (wrong)

list.h

```
struct List {int len; int *data};

struct List * new(void);

void insert (struct List *p,
             int key);

void concat (struct List *p,
             struct List *q);

int nth_key (struct List *p,
             int n);

void free (List *p);
```

List module (abstract)

list.h

```
struct List;

struct List * new(void);

void insert (struct List *p,
             int key);

void concat (struct List *p,
             struct List *q);

int nth_key (struct List *p,
             int n);

void free (List *p);
```

String Module

string module (from C90)

```
/* 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);
int    strcmp(const char *s1, const char *s2);
int    strncmp(const char *s1, const char *s2, size_t n);
char  *strstr(const char *haystack, const char *needle);
void  *memcpy(void *dest, const void *src, size_t n);
int    memcmp(const void *s1, const void *s2, size_t n);
…
```

Stdio Module

stdio module (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 1024
FILE _iob[OPEN_MAX];

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

Don’t be concerned with details

Stdio Module

stdio (cont.)

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

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

int   fputc(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, …);

int   sscanf(const char *str, const char *format, ...);
int   sprintf(char *str, const char *format, ...);
…
```

SymTable Module

SymTable module (from Assignment 3)

```
/* symtable.h */

typedef struct SymTable *SymTable_T;

SymTable_T SymTable_new(void);
void        SymTable_free(SymTable_T t);
size_t      SymTable_getLength(SymTable_T t);
int         SymTable_put(SymTable_T t, const char *key,
               const void *value);
void       *SymTable_replace(SymTable_T t, const char *key,
               const void *value);
int         SymTable_contains(SymTable_T t, const char *key);
void       *SymTable_get(SymTable_T t, const char *key);
void       *SymTable_remove(SymTable_T t, const char *key);
void        SymTable_map(SymTable_T t,
               void (*pfApply)(const char *key,
                  void *value, void *extra),
               const void *extra);
```

Agenda

A good module:

- **Encapsulates data**
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Encapsulation

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

Encapsulation Example 1

List (nonabstract)

list.h

```
struct List {int len; int *data};

struct List * new(void);

void insert (struct List *p,
             int key);

void concat (struct List *p,
             struct List *q);

int nth_key (struct List *p,
             int n);

void free_list (List *p);
```

Structure type definition in .h file

- Interface reveals how List object is implemented
  - That is, as an array
- Client can access/change data directly; could corrupt object

Encapsulation Example 1

List (abstract)

list.h

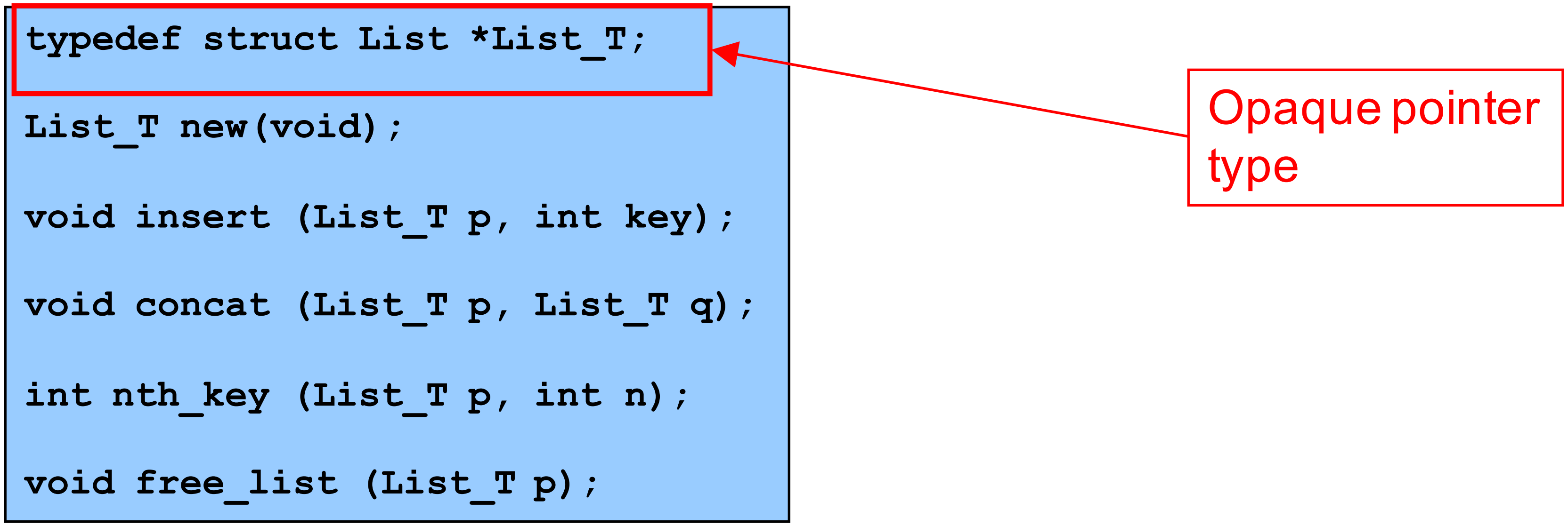
```
struct List;

struct List * new(void);

void insert (struct List *p,
             int key);

void concat (struct List *p,
             struct List *q);

int nth_key (struct List *p,
             int n);

void free_list (List *p);
```

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

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

Encapsulation Example 1

List (abstract, with typedef)



```
typedef struct List *List_T;

List_T new(void);

void insert (List_T p, int key);

void concat (List_T p, List_T q);

int nth_key (List_T p, int n);

void free_list (List_T p);
```

Opaque pointer type

- Interface provides **`List_T`** abbreviation for client
  - Interface encourages client to think of **objects** (not structures) and **object references** (not pointers to structures)
- Client still cannot access data directly; data is “opaque” to the client
- That’ s better still

Encapsulation Examples 2, 4

string

- Doesn’t encapsulate the string data: user can access the representation directly.
- This is *not* an ADT, it is just a (nonabstract) “data type.”

SymTable

- Uses the opaque pointer-to-type pattern
- Encapsulates state properly

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 abstraction principle
- 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 boneheaded enough to access the data directly

If your neighbors leave their door unlocked, does that make it OK to steal their stuff?

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

- **Why** did its designers violate the abstraction principle?

Two reasons:

1. In 1974 when stdio.h was first written, the abstraction principle was not widely understood (Barbara Liskov at MIT was just then inventing it)

2. Because function calls were expensive, getchar() and getc() were implemented as macros that accessed the FILE struct directly. But in the 21st century, function calls are not expensive anymore; getchar() isn’t a macro anymore in most implementations of stdio.h.

When’s the right time . . .

to make an opaque structure definition in stdio.h?

```
/* stdio.h */

struct FILE;
…
```

Well, suppose there are 100 million C modules that import stdio.h.

Let’s suppose 99% use FILE as if it were opaque, and would not be broken by changing to an opaque structure definition.

In which year is it OK to break 1 million C modules?

Summary: stdio.h is “practically” an ADT, and you should treat it as if “struct FILE” were opaque.

Agenda

A good module:

- Encapsulates data
- **Is consistent**
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Consistency

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 1

List

(-) Each function name begins with "List_"

(+) First parameter identifies List object

Oops, let's fix that!

```
typedef struct List *List_T;

List_T List_new(void);

void List_insert (List_T p, int key);

void List_concat (List_T p, List_T q);

int List_nth_key (List_T p, int n);

void List_free (List_T p);
```

List (revised)

(+) Each function name begins with "List_"

(+) First parameter identifies List object

Consistency Examples 1, 4

List

- (+) Each function name begins with "List_"
- (+) First parameter identifies List object

SymTable

- (+) Each function name begins with "SymTable_"
- (+) First parameter identifies SymTable object

Consistency 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);
int    strcmp(const char *s1, const char *s2);
int    strncmp(const char *s1, const char *s2, size_t n);
char  *strstr(const char *haystack, const char *needle);
void  *memcpy(void *dest, const void *src, size_t n);
int    memcmp(const void *s1, const void *s2, size_t n);
…
```

Are function names consistent?

Is parameter order consistent?

Consistency Example 3

stdio

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

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

int   fputc(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, …);

int   sscanf(const char *str, const char *format, ...);
int   sprintf(char *str, const char *format, ...);
…
```

Are function names consistent?

Is parameter order consistent?

Agenda

A good module:

- Encapsulates data
- Is consistent
- **Has a minimal interface**
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Minimization

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 **convenient** for many clients

Why?

- More functions ⇒ higher learning costs, higher maintenance costs

Minimization Example 2

string

Should any functions be eliminated?

```
/* 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);
int    strcmp(const char *s1, const char *s2);
int    strncmp(const char *s1, const char *s2, size_t n);
char  *strstr(const char *haystack, const char *needle);
void  *memcpy(void *dest, const void *src, size_t n);
int    memcmp(const void *s1, const void *s2, size_t n);
…
```

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   getchar(void);

int   fputc(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, …);

int   sscanf(const char *str, const char *format, ...);
int   sprintf(char *str, const char *format, ...);
…
```

Should any functions be eliminated?

Minimization Example 4

SymTable

- Declares **`SymTable_get()`** in interface
- Declares **`SymTable_contains()`** in interface

Should **`SymTable_contains()`** be eliminated?

Minimization Example 4

SymTable

- Defines `SymTable_hash()` in implementation

Should `SymTable_hash()` be declared in interface?

Incidentally: In C any function should be either:

- **Declared** in the interface and defined as **non-static**, or
- **Not declared** in the interface and defined as **static**

Agenda

A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- **Detects and handles/reports errors**
- Establishes contracts
- Has strong cohesion (if time)
- Has weak coupling (if time)

Error Handling

A well-designed module detects and handles/reports errors

A module should:

- **Detect** errors
- **Handle** errors if it can; otherwise…
- **Report** errors to its clients
  - A module often cannot assume what error-handling action its clients prefer

Handling Errors in C

C options for **detecting** errors

- `if` statement
- `assert` macro

C options for **handling** errors

- Write message to `stderr`
  - Impossible in many embedded applications
- Recover and proceed
  - Sometimes impossible
- Abort process
  - Often undesirable

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Set **global variable**?

```
int successful;
…
int div(int dividend, int divisor)
{   if (divisor == 0)
    {   successful = 0;
        return 0;
    }
    successful = 1;
    return dividend / divisor;
}
…
quo = div(5, 3);
if (! successful)
    /* Handle the error */
```

- Easy for client to forget to check
- Bad for multi-threaded programming

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Use **function return value**?

```
int div(int dividend, int divisor, int *quotient)
{  if (divisor == 0)
      return 0;
   …
   *quotient = dividend / divisor;
   return 1;
}
…
successful = div(5, 3, &quo);
if (! successful)
   /* Handle the error */
```

- Awkward if return value has some other natural purpose

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Use **call-by-reference parameter**?

```
int div(int dividend, int divisor, int *successful)
{  if (divisor == 0)
   {  *successful = 0;
      return 0;
   }
   *successful = 1;
   return dividend / divisor;
}
…
quo = div(5, 3, &successful);
if (! successful)
   /* Handle the error */
```

- Awkward for client; must pass additional argument

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- Call `assert` **macro**?

```
int div(int dividend, int divisor)
{   assert(divisor != 0);
    return dividend / divisor;
}
…
quo = div(5, 3);
```

- Asserts could be disabled
- Error terminates the process!

Reporting Errors in C

C options for **reporting** errors to client (calling function)

- No option is ideal

What option does Java provide?

User Errors

Our recommendation: Distinguish between…

(1) **User** errors

- Errors made by human user
- Errors that "could happen"

- Example: Bad data in **`stdin`**
- Example: Too much data in **`stdin`**
- Example: Bad value of command-line argument

- Use **`if`** statement to detect
- Handle immediately if possible, or…
- Report to client via return value or call-by-reference parameter
  - Don't use global variable

Programmer Errors

(2) **Programmer** errors

- Errors made by a programmer
- Errors that "should never happen"
- Example: pointer parameter should not be `NULL`, but is
- For now, use `assert` to detect and handle
  - More info later in the course

The distinction sometimes is unclear

- Example: Write to file fails because disk is full
- Example: Divisor argument to `div()` is 0

Default: user error

Error Handling Example 1

List

```
typedef struct List *List_T;

List_T List_new(void);

void List_insert (List_T p, int key);

void List_concat (List_T p, List_T q);

int List_nth_key (List_T p, int n);

void List_free (List_T p);
```

add assert(p) in each of the functions.... try to protect against bad clients

```
void List_insert (List_T p, int key) {
    assert(p);
    . . .
}
```

Error Handling Example 1

List

```
typedef struct List *List_T;

List_T List_new(void);

void List_insert (List_T p, int key);

void List_concat (List_T p, List_T q);

int List_nth_key (List_T p, int n);

void List_free (List_T p);
```

Operation `nth_key`(*p*,*n*), if *p* represents $\sigma_1 \cdot i \cdot \sigma_2$ where the length of σ_1 is *n*, returns *i* ; **otherwise (if the length of the string represented by *p* is ≤*n*), it returns an arbitrary integer.**

- This error-handling in List_nth_key is *a bit lame.*
- How to fix it? Some choices:
  - `int List_nth_key (List_T p, int n, int *error);`
  - Or, perhaps better: add an interface function,

`int List_length (List_T p);` and then,

Operation `nth_key`(*p*,*n*), if *p* represents $\sigma_1 \cdot i \cdot \sigma_2$ where the length of σ_1 is *n*, returns *i* ; **otherwise (if the length of the string represented by *p* is ≤*n*), it fails with an assertion failure or abort( ).**

Error Handling Examples 2, 3, 4

string

- No error detection or handling/reporting
- Example: **`strlen()`** parameter is NULL ⇒ seg fault (if you're lucky*)

stdio

- Detects bad input
- Uses function return values to report failure
  - Note awkwardness of **`scanf()`**
- Sets global variable **`errno`** to indicate reason for failure

SymTable

- (See assignment specification for proper errors that should be detected, and how to handle them)

Agenda

A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- **Establishes contracts**
- Has strong cohesion (if time)
- Has weak coupling (if time)

Establishing Contracts

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
  - Work performed
  - Meaning of return value
  - Side effects

Why?

- Facilitates cooperation between multiple programmers
- Assigns blame to contract violators!!!
  - If your functions have precise contracts and implement them correctly, then the bug must be in someone else’s code!!!

How?

- Comments in module interface

Contracts Example 1

List

```
/* list.h */

/* Return the n'th element of the list p,
if it exists.  Otherwise (if n is
negative or >= the length of the list),
abort the program. */

int List_nth_key (List_T p, int n);
```

Comment defines contract:

- Meaning of function's parameters
  - **`p`** is the list to be operated on; **`n`** is the index of an element
- Obligations of caller
  - make sure n is in range; (implicit) make sure p is a valid list
- Work performed
  - Return the n'th element.
- Meaning of return value
- Side effects
  - (None, by default)

Contracts Example 1b

List

```
/* list.h */

/* If 0 <= n < length(p), return the n’th element of
the list p and set success to 1.  Otherwise (if n is
out of range) return 0 and set success to 0. */

int List_nth_key (List_T p, int n, int *success);
```

Comment defines contract:

- Meaning of function’s parameters
  - **p** is the list to be queried; **n** is the index of an element; **success** is an error flag
- Obligations of caller
  - (implicit) make sure p is a valid List
- Work performed
  - Return the n’th element; set **success** appropriately
- Meaning of return value
- Side effects
  - Set **success**

Contracts Examples 2, 3, 4

string

- See descriptions in man pages

stdio

- See descriptions in man pages

SymTable

- See descriptions in assignment specification

Agenda

A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- **Has strong cohesion (if time)**
- Has weak coupling (if time)

Strong Cohesion

A well-designed module has **strong cohesion**

- A module’s functions should be strongly related to each other

Why?

- Strong cohesion facilitates abstraction

Strong Cohesion Examples

List

- `(+)` 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()`**, **`memcmp()`**, …
- `(+)` 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

SymTable

- `(+)` All functions are related to the encapsulated data

Agenda

A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion (if time)
- **Has weak coupling (if time)**

Weak Coupling

A well-designed module has **weak coupling** with other modules

- 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

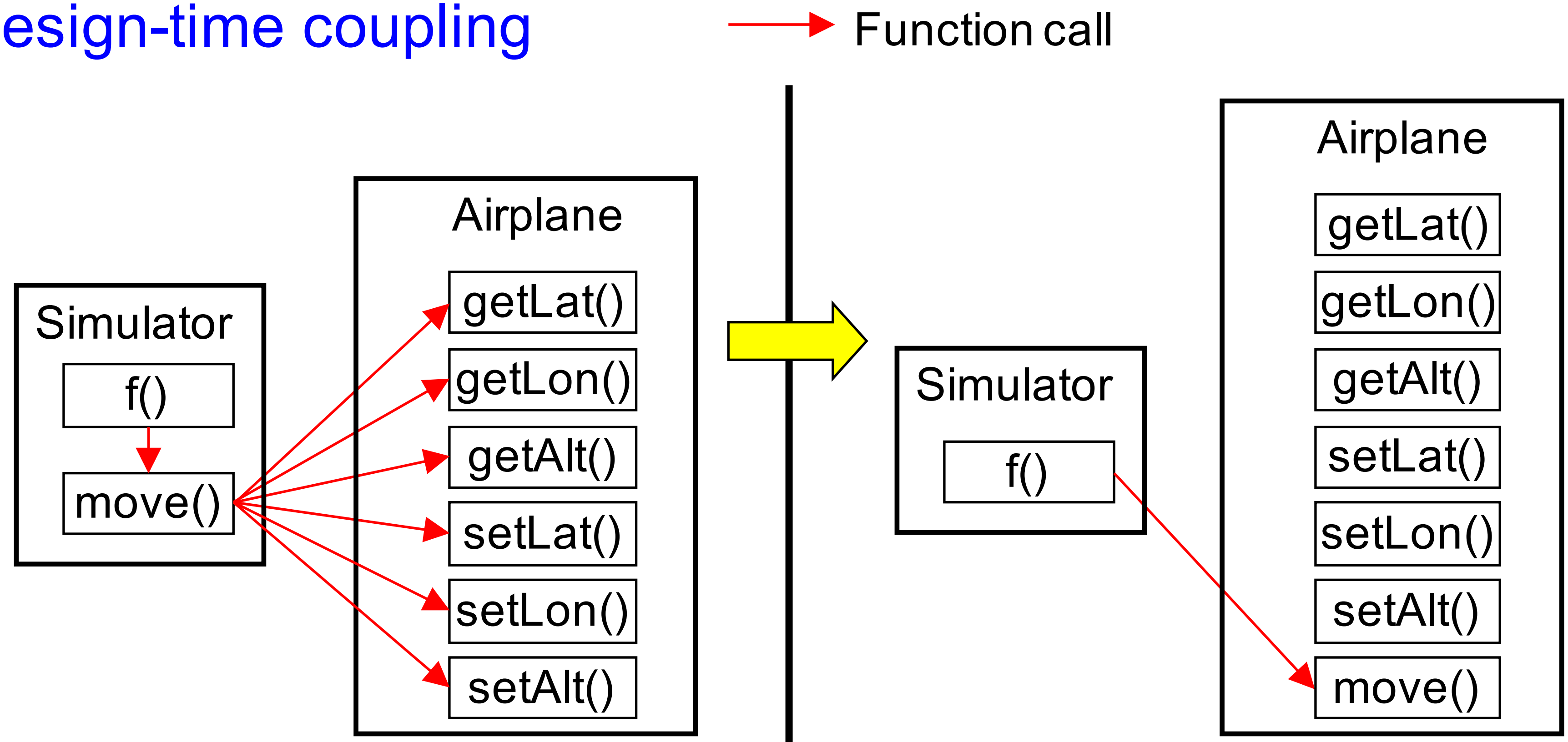
Why? Empirical evidence

- Empirically, modules that are weakly coupled have fewer bugs

Examples (different from previous)…

Weak Coupling Example 1

Design-time coupling



- Simulator module calls many functions in Airplane
- Strong design-time coupling

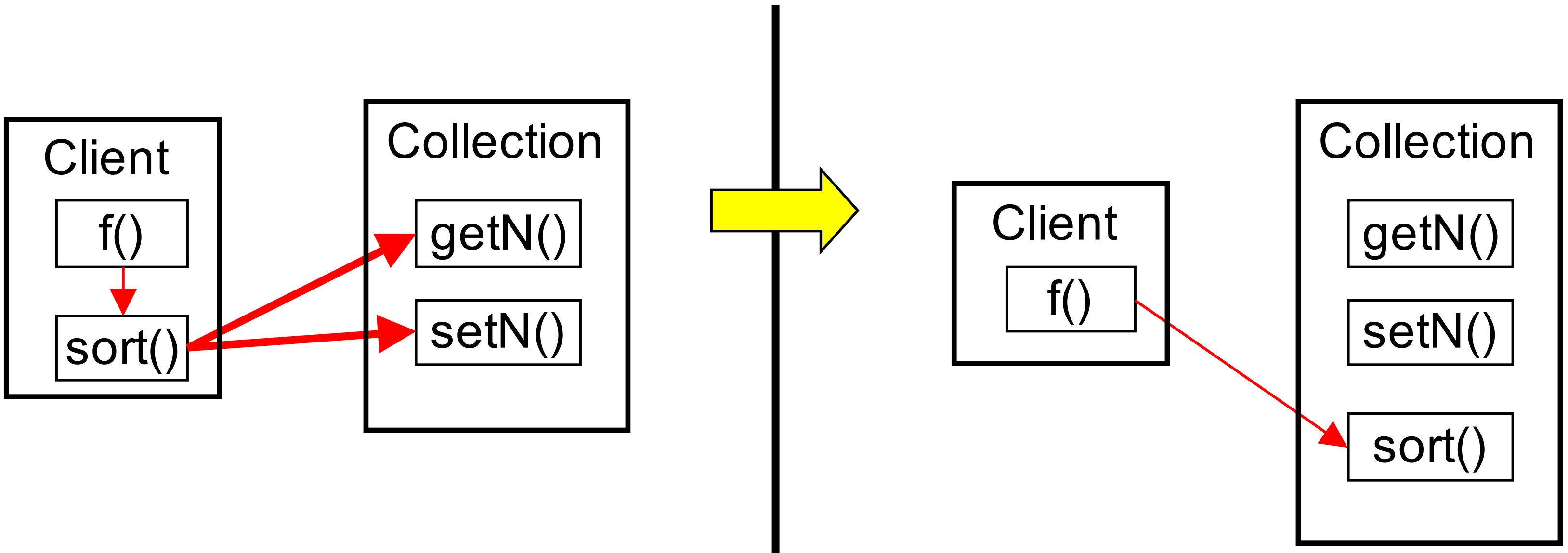
- Simulator module calls few functions in Airplane
- Weak design-time coupling

Weak Coupling Example 2

Run-time coupling

→ Many function calls

→ One function call



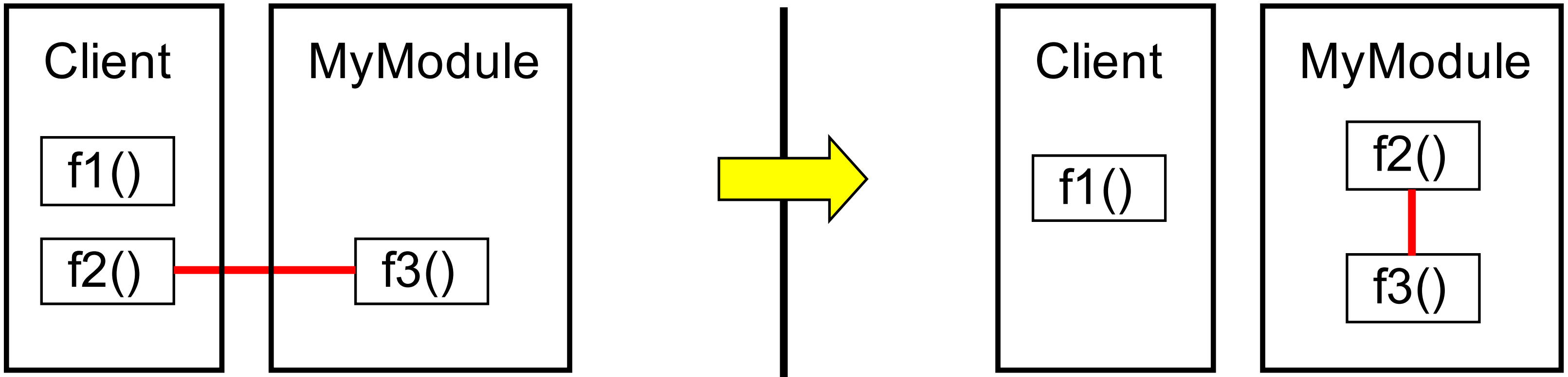
- Client module makes many calls to Collection module
- Strong run-time coupling

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

Weak Coupling Example 3

Maintenance-time coupling

🟥 Changed together often



- Maintenance programmer changes Client and MyModule together frequently
- Strong maintenance-time coupling

- Maintenance programmer changes Client and MyModule together infrequently
- Weak maintenance-time coupling

Achieving Weak Coupling

Achieving weak coupling could involve **refactoring** code:

- Move code from client to module (shown)
- Move code from module to client (not shown)
- Move code from client and module to a new module (not shown)

Summary

A good module:

- Encapsulates data
- Is consistent
- Has a minimal interface
- Detects and handles/reports errors
- Establishes contracts
- Has strong cohesion
- Has weak coupling