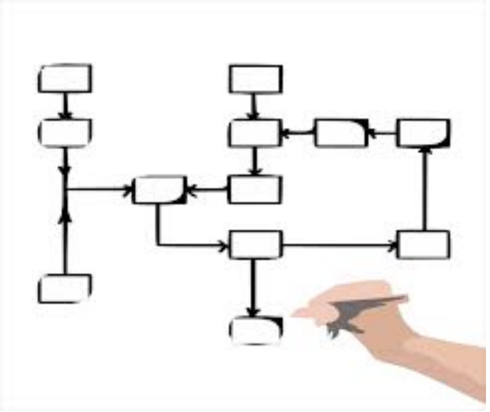
Modularity Heuristics



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

“Programming in the Large” Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms (done)
- Modularity <-- we still are here
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools

Goals of this Lecture

Help you learn:

- How to create high quality modules in C

Why?

- Abstraction is a powerful (the 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

Module Design Heuristics

We propose 7 module design heuristics

And illustrate them with 4 examples

- Stack, string, stdio, SymTable

Stack Module

Stack module (from last lecture)

```
/* stack.h */

enum {MAX_STACK_ITEMS = 100};

struct Stack
{   double items[MAX_STACK_ITEMS];
    int top;
};

struct Stack *Stack_new(void);
void          Stack_free(struct Stack *s);
int           Stack_push(struct Stack *s, double d);
double        Stack_pop(struct Stack *s);
int           Stack_isEmpty(struct Stack *s);
```

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);
int         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

Stack (version 1)

```
/* stack.h */

enum {MAX_STACK_ITEMS = 100};

struct Stack
{  double items[MAX_STACK_ITEMS];
   int top;
};

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

Structure type definition in .h file

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

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, double item);
double        Stack_pop(struct Stack *s);
int           Stack_isEmpty(struct Stack *s);
```

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

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

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, double item);
double  Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Opaque pointer type

- Interface provides `Stack_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

- “Stateless” module
- Has no state to encapsulate!

SymTable

- Uses the opaque pointer 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 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

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, 4

Stack

- (+) Each function name begins with "Stack_"
- (+) First parameter identifies Stack 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 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, double item);
double  Stack_pop(Stack_T s);
int     Stack_isEmpty(Stack_T s);
```

Should any functions be eliminated?

While we're on the subject, should any functions be added?

Minimization Example 1

Another Stack function?

```
void Stack_clear(Stack_T s);
```

- Pops all items from the Stack object

Should the Stack ADT
define Stack_clear()?

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);
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);
…
```

Should any functions be eliminated?

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

Stack

```
/* stack.c */
…
int Stack_push(Stack_T s, double d)
{   assert(s!= NULL);
    if (s->top >= MAX_STACK_ITEMS)
       return 0;
    s->items[s->top] = d;
    (s->top)++;
    return 1;
}
```

- Invalid parameter is **programmer** error
  - Should never happen
  - Detect and handle via `assert`
- Exceeding stack capacity is **user** error
  - Could happen (too much data in `stdin`)
  - Detect via `if`; report to client via return value

Error Handling Examples 2, 3, 4

string

- No error detection or handling/reporting
- Example: **`strlen()`** parameter is NULL => seg fault

stdlib

- 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

Stack

```
/* stack.h */
…
/* Push item onto s.  Return 1 (TRUE)
   if successful, or 0 (FALSE) if
   insufficient memory is available. */

int Stack_push(Stack_T s, double item);
…
```

Comment defines contract:

- Meaning of function's parameters
  - `s` is the stack to be affected; `item` is the item to be pushed
- Work performed
  - Push `item` onto `s`
- Meaning of return value
  - Indicates success/failure
- Side effects
  - (None, by default)

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

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()`, **`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**

- 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

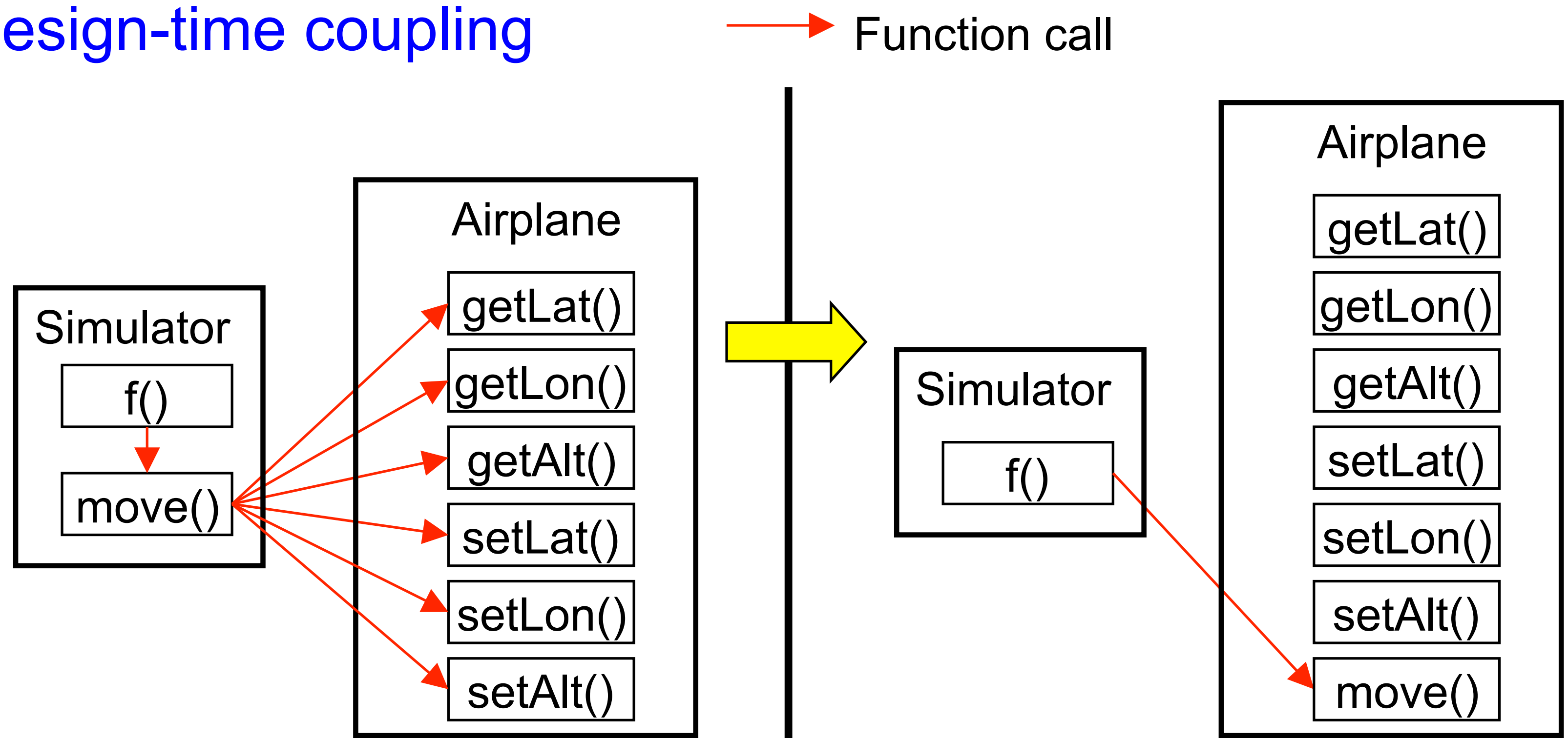
Why? Empirical evidence

- Empirically, modules that are weakly coupled have fewer bugs

Examples (different from previous)…

Weak Coupling Example 1

Design-time coupling

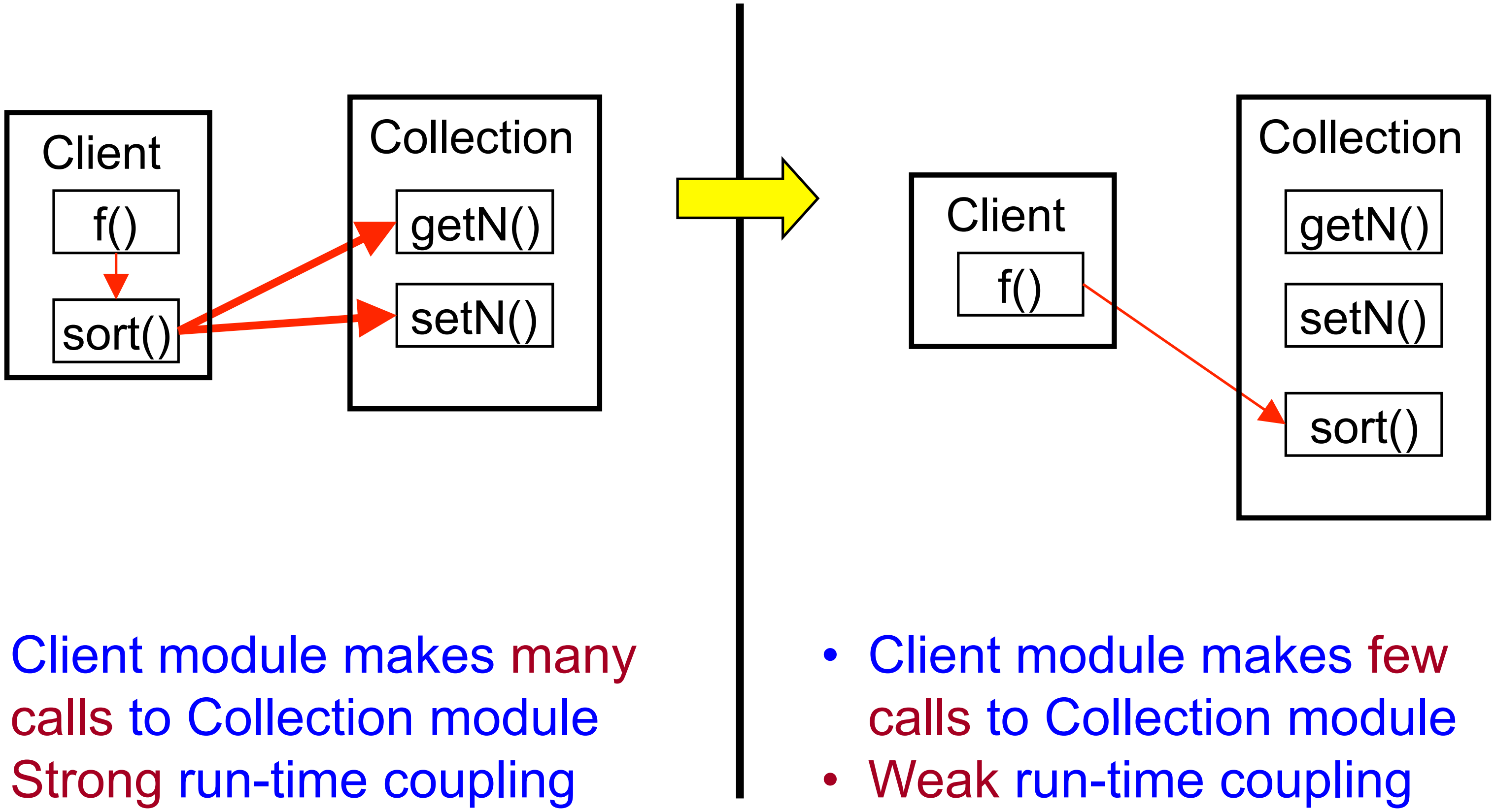


- Simulator module calls many functions in Airplane
- Strong design-time coupling

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

Weak Coupling Example 2

Run-time coupling

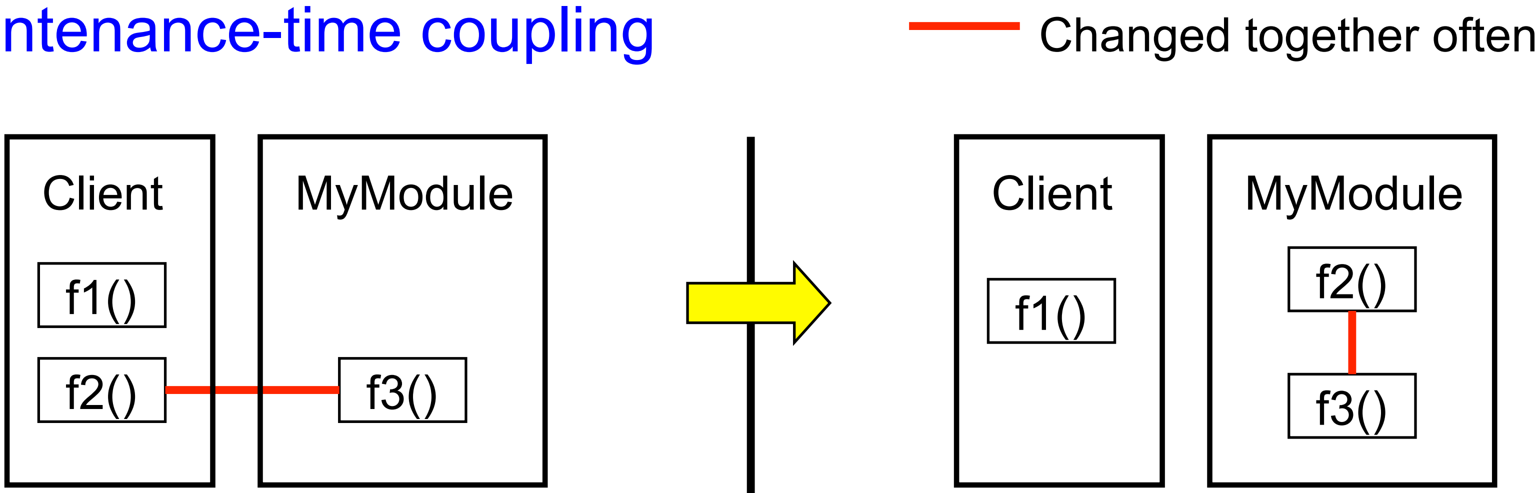


- Client module makes many calls to Collection module
- Strong run-time coupling

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

Weak Coupling Example 3

Maintenance-time coupling



- 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