Module 6:
Part I: Modularization

Readings: CP:AMA 19.1, 10.2 – 10.5

The primary goal of this section is to be able to write a module.

Detour: mutual recursion in C

You may have noticed that we can't yet do *mutual recursion* in C.

We write two **definitions**:

```
bool is_even(int n) {
  if (0 == n) {
    return true;
  } else {
    return is_odd(n - 1);
  }
}

bool is_odd(int n) {
  if (0 == n) {
    return false;
  } else {
    return is_even(n - 1);
  }
}
```

When we try to compile this, we see:

```
warning: implicit declaration of function 'is_odd'
error: conflicting types for 'is_odd'; have '_Bool(int)'
```

When it tries to compile `is_even`, the compiler sees that we need a function `is_odd`.

It "guesses" that maybe `is_odd` should take and return an `int`. (The "implicit declaration".)

Then we define `is_odd`, and this "guess" is wrong: `is_odd` actually returns a `bool`. We get an error.

Function **declarations**

We need to tell the compiler,
"I promise that later I will define a function `is_odd` that will take a `int` and return a `bool`".

We do this by writing a function **declaration**:

```
bool is_odd(int n);
```

A function **declaration** looks like a function **definition**, but has no code block `{...}`.

We may omit the `names` of the parameters:

```
bool is_odd(int);
```

...but it's bad style. Include the names of the parameters when you write a declaration.

Declaration vs definition

- A **declaration** only specifies the *type* of an identifier.
- A **definition** instructs C to *"create"* the identifier.

However, a definition *also* specifies the type of the identifier, so

> **!** **a definition also includes a declaration**.

An identifier can be declared multiple times, but only defined once.

Function ordering

By adding a declaration, we can now place `main` above a function that it calls:

```c
int my_sqr(int n);       // DECLARATION

int main(void) {
  trace_int(1 + 1);
  trace_int(my_sqr(7));  // OK
}

int my_sqr(int n) {      // DEFINITION
  return n * n;
}
```

! A declaration is necessary to implement **mutual recursion**.

Function ordering

Adding only the two **declarations**, it now works:

```
bool is_even(int n);
bool is_odd(int n);

bool is_even(int n) {
  if (0 == n) {
    return true;
  } else {
    return is_odd(n - 1);
  }
}
```

```
bool is_odd(int n) {
  if (0 == n) {
    return false;
  } else {
    return is_even(n - 1);
  }
}

int main(void) {
  assert(is_even(42));
  assert(is_odd(17));
}
```

Exercise

The Hofstadter-FM sequences are defined as:

$$F(0) = 1; M(0) = 0$$
$$F(n) = n - M(F(n-1)), \quad n > 0$$
$$M(n) = n - F(M(n-1)), \quad n > 0$$

Write a pair of functions `F` and `M` that compile without warnings and work correctly.

```
printf("F:");
for (int i = 0; i < 24; ++i) { printf(" %d,", F(i)); }
printf("\nM:");
for (int i = 0; i < 24; ++i) { printf(" %d,", M(i)); }
```

Should print:

```
F: 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 8, 8, 9, 9, 10, 11, 11, 12, 13, 13, 14, 14,
M: 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 9, 10, 11, 11, 12, 12, 13, 14, 14,
```

Modularization

In Racket, we used the `local` keyword. How did it help?

From the CS135 course notes:

~~Clarity~~	~~Naming subexpressions~~ *local variables in C.*
~~Efficiency~~	~~Avoid recomputation~~ *local variables in C.*
Encapsulation	Hiding stuff – this will become part of *abstraction*.
Scope	~~Reusing parameters~~ *No* `local` *functions in C*, though scope does matter.

Modularization

We want to be able to build larger projects; we have bigger goals:

Reusability	using the same module in many projects
Maintainability	programmers can work on separate modules simultaneously
Abstraction	a module can change how it implements its task, as long as the **interface** is maintained.

In C we can use **modules** to break a program into multiple files.

Example: the Racket-8.11.1 source code includes:

- 93 675 lines of Racket code in 668 files,
- 343 726 lines of C code in 449 files.

Modularization

The concept of modularization extends far beyond computer science. You can see examples of modularization in construction, automobiles, furniture, nature, etc.
A practical example of a good modular design is an “AA battery”. You can replace any AA battery with any other AA battery, even with a different chemistry.

We have already seen an elementary type of modularization in the form of *helper functions* that can “help” many other functions.

When designing larger programs, we move from writing “helper functions” to writing “helper modules”.

A **module** provides a collection of functions that share a common aspect or purpose.

There are three key advantages to modularization: reusability, maintainability and abstraction.

reusability: A good module can be reused by many clients. Once we have a “repository” of reusable modules, we can construct large programs more easily.

Maintainability: It is much easier to test and debug a single module instead of a large program. If a bug is found, only the module that contains the bug needs to be fixed. We can even replace an entire module with a more efficient or more robust implementation.

Abstraction: To use a module, the client needs to understand **what** functionality it provides, but it does not need to understand **how** it is implemented. In other words, the client only needs an *“abstraction”* of how it works. This allows us to write large programs without having to understand how every piece works.

Modules vs files

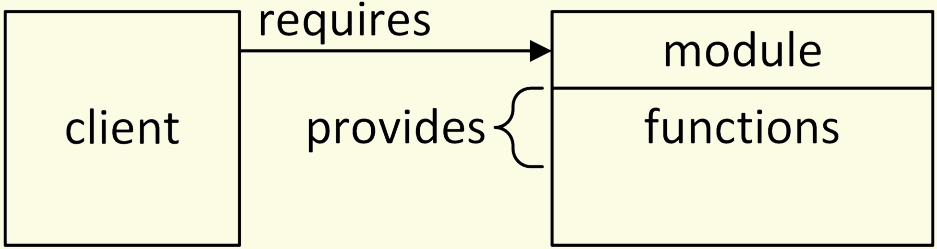
In this course, and in much of the “real world”, it is considered **good style** to store **modules in separate files**.

While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

> Some computer languages enforce this relationship (one file per module), while in others it is only a popular *convention*.
> There are advanced situations (beyond the scope of this course) where it may be more appropriate to store multiple modules in one file, or to split a module across multiple files.

Modules vs files

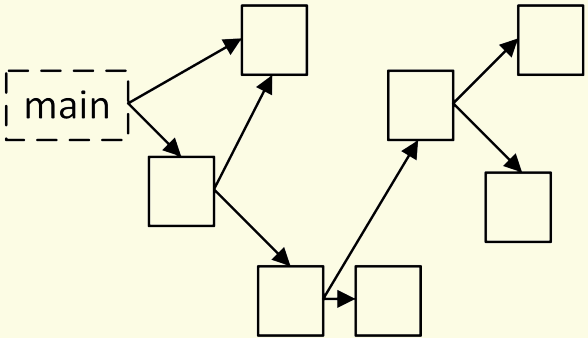
It is helpful to think of a **“client”** that **requires** the functions that a module **provides**.



Modules vs files

Large programs can be built from many modules.

A module can be a client itself and *require* functions from other modules.



There must be a “root” (or *main file*) that acts only as a client.
This is the program file that defines `main` and is “run”.

Modules vs files

We want to create a prime module that provides an `is_prime` function:

```
// prime.c
// is_prime(n) determine if n is prime.
// (FIXME: this is buggy, why?!)
bool is_prime(int n) {
  return (n >= 2 && (n == 2 || n == 3 || n == 5
                     || (n % 2 != 0 && n % 3 != 0 && n % 5 != 0)));
}
```

This illustrates three key advantages of modularization:

Reusability multiple programs can use the module.

Maintainability To fix the bug or when we come up with a better algorithm, we only to update the module.

Abstraction The **client** does not need to understand how it does its calculation. (Code that relies on a module is called a “client”.)

Modules vs files

We try to use this from `main`:

```c
// main.c
#include <stdio.h>
// 14
int main(void) {
  printf("Here are some primes,
    apparently:\n");
  for (int i=0; i<100; ++i) {
    if (is_prime(i)) { // we can
    use functions from the module.
      printf(" %d", i);
    }
```

When we try to compile this, we see:

```
warning: implicit declaration of function 'is_prime'
```

A familiar warning!

When we saw this warning before, we used a function **declaration** to fix it.

We are going to do the same thing here. But we're not going to declare it in `main.c`. Instead, we create a file `prime.h`, and put the declaration there.

Then in `main.c`, we add `#include "prime.h"`

`#include` is a **preprocessor directive**: it "inserts" the contents of the named file in its

Modules vs files

A module consists of two files:

- the **implementation file** (`.c`) that contains **definitions**, and
- the **interface file** (`.h`) that contains only **declarations**.

> The interface is everything that a client needs to use the module: the documentation and declarations.
> The client does not need to see the implementation file.

Example: prime number module

implementation

```c
// prime.c
#include "prime.h"
// See prime.h for details.
// (FIXME: this is buggy, why?!)
bool is_prime(int n) {
  return (n >= 2 && (n == 2 || n == 3 || n == 5
                     || (n % 2 != 0 && n % 3 != 0 && n % 5 != 0)));
}
```

interface

```c
// prime.h
#include "cs136.h"

// is_prime(n) determine if n is prime.
bool is_prime(int q);
```

Exercise

Get this to work. Then change `prime.c` to make `is_prime` work for larger `n`.
(Try primes 8191, 999331, etc.)

Global variables and constants in modules

Sometimes it's useful to have global **constants** (or variables) in a module.

For example, we have already seen global constants `INT_MIN` and `INT_MAX`.
The `cs136.h` module gets these from the standard `limits.h` module.

Recall: to make a function available in a module, we:

1. Wrote a **definition** in our `.c` file,
```
bool is_prime(int n) {
  // ... code ...
}
```
2. Wrote a **declaration** in our `.h` file.
```
bool is_prime(int q);
```

Global variables and constants in modules

Imagine we want to make available two constants: an array containing the Mersenne primes that can be stored in an `int`, $\{3, 7, 31, 127, 8191, 131071, 524287, 2147483647\}$, and the length of this array, 8.

For each constant, we will:

1. Write a **definition** in our `.c` file, and

```
const int mersenne[8] = {3, 7, 31, 127, 8191, 131071, 524287, 2147483647};
const int mersenne_count = 8;
```

2. Write a **declaration** in our `.h` file.

```
extern const int mersenne[];
extern const int mersenne_count;
```

Note: the declaration does not initialize the constants or specify the array size. "`extern`" means it is defined "somewhere else": not in the `.h` file, but in the `.c` file.

Structures in modules

Later we will see how we can make **opaque** structures in modules.

For now, we create a **transparent** structure simply by putting the structure declaration in the interface file. For example, in a module to deal with rational numbers, we might use the following structure:

```
struct rational {
  int numerator;
  int denominator;
};
```

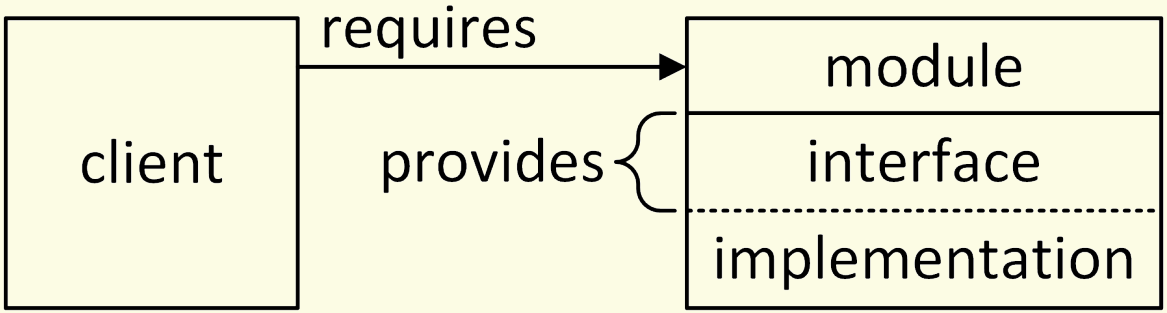
Exercise

Create a module `rational`. It should have:

- a function `void print_rational(const struct rational *f)` that prints `f` as a fraction.
- a constant `R_PI` that stores a rational approximation of π: $\frac{22}{7}$, or $\frac{5419351}{1725033}$, or similar.

Exercise

Expand the `rational` module so it can do various reasonable things, like add, multiply, or print as a decimal.



The **interface** is what is provided to the client (in the `.h` header file).

The **implementation** is hidden from the client (in the `.c` source file).

interface file contents

The contents of the interface file include:

- an **overall description** of the module
- a **declaration** for each provided function or global constant
- **documentation** (e.g. a **purpose**) for each provided function

The interface should also provide *examples* to illustrate how the module is used and how the interface functions interact.

```
// This is the prime module.
// It helps us work with prime numbers.

#include "cs136.h"

// is_prime(n) determine if n is prime.
// examples: is_prime(0) => false
//           is_prime(1) => false
//           is_prime(2) => true
//           is_prime(3) => true
//           is_prime(15) => false
//           is_prime(127) => true
bool is_prime(int n);

extern const int mersenne[];
extern const int mersenne_count;
```

Implementation notes

In the previous example, the implementation `prime.c` included its own interface `prime.h`.

This is good style as it helps catch certain discrepancies between the two files.

Implementation notes

A file should only `#include` a header if it has a reason to do so.

`prime.h` uses the `bool` type.
In this course we get it from `cs136.h`, so `prime.h` needs to `#include "cs136.h"`.

Alternatively, it could be more specific and only `#include <stdbool.h>`.

If the module only used `int` values, it might not need to `#include` anything in the its header file.

But if it uses library functions such as `printf` in the source file, it would need to `#include` the appropriate library in the source file: either `#include "cs136.h"`, or more specifically `#include <stdio.h>`.

Implementation notes

In addition to the `#include`, The CP:AMA textbook frequently uses the `#define` directive. In its simplest form it performs a *search & replace*.

```
// replace every occurrence of MY_NUMBER with 42
#define MY_NUMBER 42

int my_add(int n) {
  return n + MY_NUMBER;
}
```

In C99, it is usually better style to define a variable (constant), but you will still see `#define` in the "real world".

cs136.h

Since the beginning of this course, we have started each program with:

```
#include "cs136.h"
```

We now understand that we have been using support tools and functions (e.g. `trace_int` and `read_int`) provided by a `cs136` module.

The `cs136.h` interface file requires additional modules:

```
#include <assert.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

C standard modules

Unlike Racket, there are no “built-in” functions in C.

Instead, C provides **standard modules** (also known as **libraries**).

For example, the `stdio` module provides `printf` and `scanf`.

When using `#include` we use `<angle brackets>` for standard modules, and `"quotes"` for custom modules.

```
#include <stdio.h>
#include "mymodule.h"
```

Here are some of the other standard modules that we have been using:

- `assert.h` provides the function `assert`
- `limits.h` provides the constants `INT_MAX` and `INT_MIN`
- `stdbool.h` provides the `bool` data type and the constants `true` and `false`
- `stdlib.h` provides the constant `NULL`

Test Client

For each module you design, it is good practice to create a **test client** that ensures the provided functions are correct.

We have been doing this in the file called `main.c`, but there is nothing special about that file.

Whichever file has the function called `main` is the "main" file of the program. You cannot have more than one function called `main`.

```c
// test-prime.c: testing client for the prime module
#include <assert.h>
#include "prime.h"
int main(void) {
  assert(!is_prime(42));
  assert(is_prime(13));
  //...
}
```

> **!** Your module must not contain a function called `main`.

Test Client

There may be “white box” tests that cannot be tested by a client. These may include implementation-specific tests and tests for module-scope functions.
In these circumstances, you can provide a `test_module_name` function that asserts your tests are successful.

Interface Design

The ability to break a big programming project into smaller modules, and to define the interfaces between modules, is an important skill that will be explored in later courses.

Unfortunately, due to the nature of the assignments in this course, there are very few opportunities for you to **design** any module interfaces.

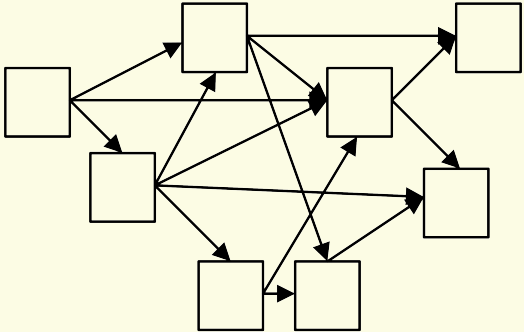
For now, we will have a brief discussion on what constitutes a **good** interface design.

Cohesion and coupling

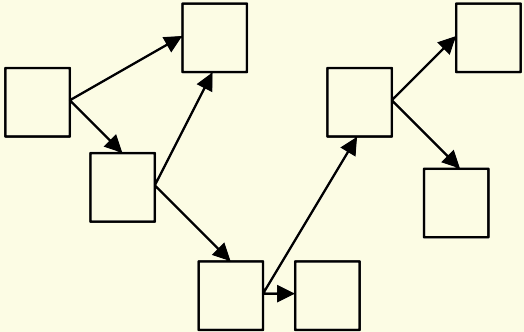
When designing module interfaces, we want to achieve *high cohesion* and *low coupling*.

High cohesion means that all of the interface functions are related and working toward a "common goal". A module with many unrelated interface functions is poorly designed.

Low coupling means that there is little interaction *between* modules. It is impossible to completely eliminate module interaction, but it should be minimized.



High coupling



Low coupling

Module 6:
Part II: Abstract Data Types

An ADT is a **mathematical model** for storing & accessing data through **operations**.

As mathematical models they transcend any specific computer language or implementation.

In practice (and in this course) ADTs are **implemented** as data storage **modules** that only allow access to the data through interface functions (ADT *operations*). The underlying data structure and implementation of an ADT is **hidden** from the client (which provides *flexibility* and *security*).

Dictionary (revisited)

In CS 135 we saw our first collection ADT: a **dictionary**.

The dictionary ADT is a collection of **pairs** of *keys* and *values*. Each *key* is unique and has a corresponding value, but more than one key may have the same value.

Dictionary ADT operations:

lookup	for a given key, retrieve the corresponding value or "not found"
insert	add a new key/value pair or replace the value of an existing key
remove	delete a key and its value

The "dictionary ADT" is this interface that includes **lookup**, **insert**, and **remove**.

We **implemented** this ADT in Racket, at least twice. We will implement it in C.

These implementations may be quite different. But provided they satisfy this interface, they "are" the Dictionary ADT.

Data structures vs ADTs

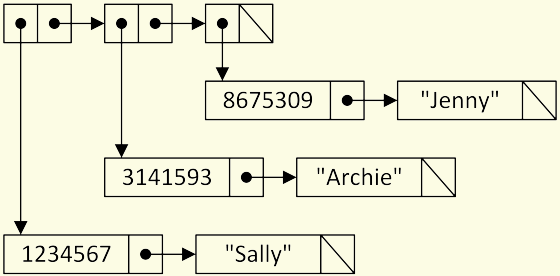
As the **client**, if you have a **data structure**, you know how the data is "structured" and you can access the data directly in any manner you desire.

With an **ADT**, the client does not know how the data is structured and can only access the data through the interface functions (operations) provided by the ADT.

Data structures vs ADTs

In CS 135 we implemented a dictionary with an association list **data structure**.

```
(define al-dict '((1234567 "Sally") (3141593 "Archie") (8675309 "Jenny")))
```

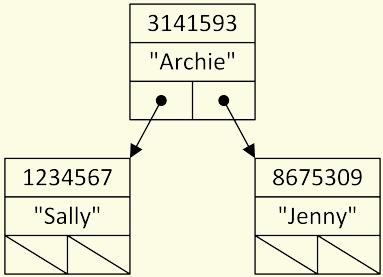



```
;; (lookup k alst) produces the value
;;     associated with k if k is in alst;
;;     false otherwise.
;; lookup-al: Num AL -> (anyof Str false)
(define (lookup k alst)
  (cond [(empty? alst) false]
        [(= k (first (first alst)))
         (second (first alst))]
        [else
         (lookup k (rest alst))]))
```

```
(check-expect (lookup 3141593 al-dict) "Archie")
(check-expect (lookup 1122334 al-dict) false)
```

We also implemented a dictionary with a Binary Search Tree **data structure**.

```
(define bst-dict (make-node 3141593 "Archie"
                            (make-node 1234567 "Sally" empty empty)
                            (make-node 8675309 "Jenny" empty empty)))
```




```
;; (lookup k t) produces the value
;;     associated with k if k is in t;
;;     false otherwise.
;; lookup-bst: Nat BST -> (anyof Str false)
(define (lookup k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (lookup k (
    node-left t))]
       [(> k (node-key t)) (lookup k (
    node-right t))]))
```

```
(check-expect (lookup 3141593 bst-dict) "Archie")
(check-expect (lookup 1122334 bst-dict) false)
```

Data structures vs ADTs

In Racket we had two implementations of `lookup`.
Unless we look at what a `AL` is or what a `BST` is, to a client they are *identical*.

To implement a dictionary, we have a choice:
use an association list, a BST, or perhaps something else?

There will be advantages and disadvantages of each choice.

But the client would not know which implementation was being used.

More collection ADTs

Three additional collection ADTs that are explored in this course:

- stack
- queue
- sequence

Stack ADT

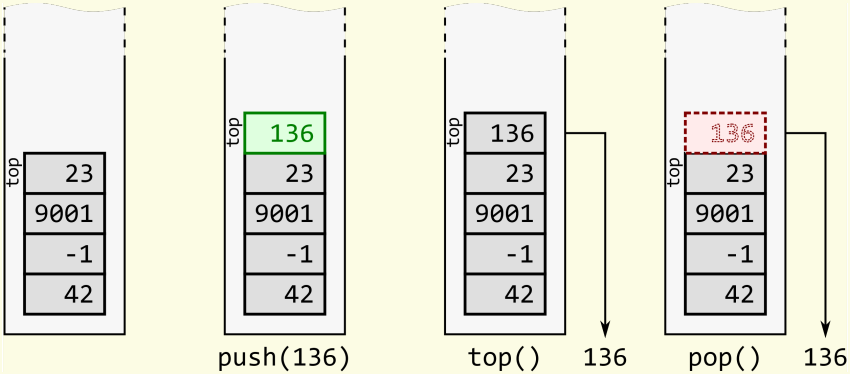
A **stack** ADT is like a stack of books.

In a stack ADT, items are **pushed** onto the top of stack and **popped** off the top of stack. A stack exhibits LIFO behaviour (last in, first out).

We have already been exposed to the idea of a stack when we learned about the **call stack**.

Stack ADT

- **push:** adds an item to the top stack
- **pop:** removes the top item from the stack
- **top:** returns the top item on the stack
- **is_empty:** determines if the stack is empty



Stack ADT

We can't create a stack "properly" without knowing how big it will be.
For now, we will make an implementation that can hold at most 100 items.

Exercise

Create a module `stack` that has:

- A **transparent** structure definition, that stores:
  - `maxlen`: the max number of items that could be in the stack; this should be 100
  - `len`: how many items are in the stack currently
  - `data`: an array of 100 items

  Then in `main`, `struct stack mystack = {100, 0, {0}}` will create an empty stack.
- `void print_stack(const struct stack *s)`
- `void stack_push(int v, struct stack *s)`
- `int stack_pop(struct stack *s)`
- `int stack_top(const struct stack *s)`
- `bool stack_is_empty(const struct stack *s)`

Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- **add-back:** adds an item to the end of the queue
- **remove-front:** removes the item at the front of the queue
- **front:** returns the item at the front
- **is-empty:** determines if the queue is empty

Sequence ADT

The sequence ADT is useful when you want to be able to retrieve, insert or delete an item at any position in a sequence of items.

Typical sequence ADT operations:

- **item-at:** returns the item at a given position
- **insert-at:** inserts a new item at a given position
- **remove-at:** removes an item at a given position
- **length:** return the number of items in the sequence

The **insert-at** and **remove-at** operations change the position of items after the insertion/removal point.

Goals of this Section

At the end of this section, you should be able to:

- explain and demonstrate the three core advantages of modular design: abstraction, reusability and maintainability
- explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.
- explain the difference between a declaration and a definition
- write modules in C with implementation and interface files
- write good interface documentation
- as a client, use a module; write a test client