Modularization

Readings: CP:AMA 19.1

Modularization

In CS 135 (and CS 115) we designed solutions with a small number of functions in a single Racket (`.rkt`) file.

For large programs with many functions, this approach becomes unwieldy. Working with a single file makes teamwork more difficult, and it is difficult to share and re-use code between programs.

A better strategy is to use ***modularization*** to divide programs into well defined ***modules***.

In computer science, a **module** is collection of functions that share a common aspect or purpose.

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 a “AA battery”.

Motivation

There are three key advantages to modularization: re-usability, maintainability and abstraction.

Re-usability: A good module can be re-used by many programs. Once we have a "repository" of re-usable modules, we can construct large programs much 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 changed. We can even replace an entire module with a more efficient or more robust implementation.

Abstraction: To use a module, we do not need to understand how it is implemented. We only need an *"abstraction"* of how it works. This allows us to write large programs without having to understand how every piece works. Abstraction also improves teamwork, because modules may be written by different team members.

> *Modularization* is also known in computer science as the *Separation of Concerns (SoC)*.

A lucky example

To help illustrate modularization, consider a program to help Waterloo students find suitable housing.

To write this program, we would identify the modules we would need. For example, we might want modules for collecting housing data, searching, using maps, processing credit cards, etc.

What if we want this program to give *superstitious* students the ability to find housing with “lucky” addresses?

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

We should create a lucky *module* for working with lucky numbers.

We identified 3 modularization advantages:

- **Re-usability:** Other programs can now easily add features for working with lucky numbers (e.g.: a wedding planner program).
- **Maintainability:** A separate module will help us test our implementation. If we find a bug, or we develop a more effective method for finding lucky numbers, we can change our code in one place and update several programs.
- **Abstraction:** To use this module, you do not need to understand how lucky numbers are determined (it could be straightforward or very complicated). *It does not matter* because the details have been *abstracted* away.

Module interface

One of the most important aspects of modular design is developing the ***interfaces*** of the modules.

The interface of a module is:

- a collection of functions (and constants) that are accessible outside of the module, and
- the documentation for the module.

It is helpful to think of a *"client"* who is using a module.

The interface is everything a client needs to use the module.

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. If module A depends on the interface for module B and B also depends on the interface for A, that is high coupling and poor design.

Information hiding

Another important aspect of interface design is ***information hiding***, where information about the module *implementation* is intentionally hidden from the client. The two key advantages of information hiding are ***security*** and ***flexibility***.

Security is important because we may want to prevent the client from tampering with data used by the module. Even if the tampering is not malicious, we may want to ensure that the only way the client can interact with the module is through the interface. We may need to protect the client from themselves.

If we design the interface properly, the interface should not reveal any details about how the module has been implemented.

By hiding the implementation details from the client, we gain the **flexibility** of changing the implementation in the future.

In CS 135, we used the term *behaviour encapsulation* to describe a form of information hiding: using `local` helper functions.

Modules in Racket

There are two Racket special forms that allow us to work with modules: `provide` and `require`.

`provide` is used in a module to identify the interface functions that are available to clients.

`require` is used by clients to identify a module that the client depends upon.

There is also a `module` special form in Racket, and many other module support functions that we will not introduce in this course.

Creating a module

In full Racket, each `.rkt` file we create is automatically considered a module. However, none of the functions will be accessible to clients unless you `provide` them.

Conceptually, the `provide` special form can be seen as the "opposite" of the `local` special form: `local` makes definitions "invisible", whereas `provide` makes definitions "visible".

Any private functions you wish to hide from the client should not be provided.

For our lucky module, we will have two functions: `lucky?` will determine if a number is lucky or not, and `luckiest` will determine the luckiest number from a list of numbers.

```
#lang racket ;; lucky.rkt

(provide lucky? luckiest)

(define (lucky? n)
  ...)

(define (luckiest lon)
  ...)

(define (hidden-helper n) ; not provided
  ...)
```

In this example, the function `hidden-helper` is private and not visible to the clients.

Interface documentation

The module interface includes documentation **for the client**: it should provide the client with all of the information necessary to use your module. The client does not need to know how your module is implemented.

Interface documentation must include:

- an overall description of the module,
- a list of functions it provides, and
- the contract and purpose for each provided function.

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

```
#lang racket ;; lucky.rkt

;; this module provides lucky number functions

(provide lucky? luckiest)

;; lucky?: Num -> Boolean
;;   PRE:  true
;;   POST: produces #t if n is lucky, #f otherwise
;; (lucky? n) determines if n is a lucky number

;; luckiest: (listof Num) -> Num
;;   PRE:  lon is not empty
;;   POST: produces an element of lon
;; (luckiest lon) finds the luckiest number in lon

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; see interface above
(define (lucky? n)
  ...)
```

For Racket modules, the interface should appear at the top of the file above any function definitions.

For public (provided) functions, it is not necessary to duplicate the interface documentation.

For private functions, the proper documentation (contract and purpose) should accompany the function definition.

Example: cohesion and coupling

Our lucky interface should have *high cohesion* and *low coupling*.

To achieve **high cohesion**, all of the interface functions should be related to lucky numbers. Providing a function for calculating the area of a triangle would be low cohesion.

To achieve **low coupling**, the interface should not depend on the "parent" program. If we provide an interface function `lucky-apartment?`, it would depend on the data definition for an apartment and our module would not be very re-usable.

A better strategy would be to create a "lucky apartment" module that depends on our lucky module and an apartment module.

Example: information hiding

The following module implementation has only one lucky number, which is visible in the interface:

```
(provide lucky? lucky-number)

(define lucky-number 42) ; this is provided

(define (lucky? n)
  (equal? n lucky-number))
```

This could be a **security** problem if the client uses this information directly or in unintended ways.

Later, we will have modules that manage data, and see better examples of how information hiding provides security.

The previous module implementation also reduced **flexibility**, which we can regain by hiding information:

```
(provide lucky?)

(define lucky-number 42) ; not provided

(define (lucky? n)
  (equal? n lucky-number))
```

This interface gives us the flexibility to change the implementation **without changing the interface**:

```
(provide lucky?)

(define lucky-numbers (list 7 42 88))

(define (lucky? n)
  (if (member n lucky-numbers) #t #f))
```

Using Racket modules

To use a module in a client, you use the `require` special form.

In this course, you should always put the required modules in the same directory (folder) as the client module.

```
#lang racket ; client.rkt

(require "lucky.rkt") ; provides lucky? and luckiest

(lucky? 13) ; => #f

(luckiest '(4 8 15 16 23 42)) ; => 42
```

The require special form

When the `require` special form is evaluated, it will "run" all of the code in the required module and make the `provide`d functions available.

`require` will also *display* the final value of any of the top-level expressions in the module. You should avoid any top-level expressions and only put *definitions* in your modules.

```
#lang racket ; annoying.rkt

(provide f)

(define (f x) (sqr x))

"This string is displayed every time you require this module"
```

Testing modules

Without `check-expect` or top-level expressions, how will we **test** our module?

For each module you design, you should also create a **testing module** that ensures the interface functions work properly.

> If you want to perform "white box" tests, including any implementation-specific tests or tests on private functions, you can `provide` a `test-module-name` function.

Example: testing module

On the next slide, we will provide a corresponding test module.

```
#lang racket ;; sum.rkt

;; this module provides functions for summing integers

(provide sum-first)

;; sum-first: Int -> Int
;;   PRE:  k >= 1
;;   POST: produces an integer >= 1
;; (sum-first k) finds the sum of all integers 1..k

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sum-first k)
  (if (<= k 1) 1 (+ k (sum-first (sub1 k)))))
```

```
#lang racket ;; test-sum.rkt

;; this is a testing module for sum.rkt

(require "sum.rkt")

; Each of the following should be #t

(equal? (sum-first 1) 1)
(equal? (sum-first 2) 3)
(equal? (sum-first 3) 6)
(equal? (sum-first 10) 55)
(equal? (sum-first 99) 4950)
```

Later we will discuss more advanced testing strategies.

Goals of this module

You should understand the three core advantages of modular design: abstraction, re-usability and maintainability.

You should understand and be able to identify two characteristics of a good modular interface: high cohesion and low coupling.

You should understand information hiding, and why it supports both security and flexibility.

You should understand what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.

You should be comfortable using `provide` and `require` to implement modules in Racket

You should understand the importance of good interface documentation and the changes to our documentation requirements