Modularization

Readings: CP:AMA 19.1

Modularization

In previous courses we designed programs with a small number of definitions in a single Racket (`.rkt`) file.

For large programs, keeping all of the code in one file is unwieldy. Teamwork on a single file is awkward, and it is difficult to share or 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 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 need to understand **what** functionality it provides, but we do not need to understand **how** it is implemented. In other words, we only need an *"abstraction"* of how it works. This allows us to write large programs without having to understand how every piece works.

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

example: tax module

Consider a program that allows Waterloo students to order their school supplies online and have them delivered.

Such a large program might be broken into modules, such as:

- looking up course requirements,
- keeping track of inventories,
- creating a virtual shopping cart,
- processing payments such as credit cards, etc...

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

Consider an “Ontario tax module” that can identify the proper tax rate for each item (*e.g.,* textbooks are 5%, but empty workbooks are 13%). The advantages of creating this module are:

- **re-usability:** This module could be easily reused for other Ontario-based shopping applications.
- **maintainability:** When the tax rates and rules change, we only have to modify one module.
- **abstraction:** To use this module, you do not need to understand how the taxes are calculated. *It does not matter* because the details have been *abstracted* away.

example: fun numbers

Consider that some integers are more “fun” than others, and we want to create a Racket module that “provides” a `fun?` function.

```
;; fun.rkt

(provide fun?)  ; <---- this is new!

(define lofn '(-3 7 42 136 1337 4010 8675309))

;; (fun? n) determines if n is a fun integer
;; fun?: Int -> Bool
(define (fun? n)
  (not (false? (member n lofn))))
```

The `provide` special form above makes the `fun?` function available to other programs.

The `require` special form allows us to use the `fun?` function in a different program.

```
;; different-program.rkt

(require "fun.rkt")  ; <---- this is new!

(fun? 7)     ; => #t
(fun? -7)    ; => #f
```

This programmer only requires an *"abstract"* understanding of fun integers, and does not need to understand what integers are fun, or how they are determined.

When new numbers become fun (or become less fun), this program does not need to change. Only the fun module is changed (this increases maintainability).

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 specify the identifiers or "bindings" (*e.g.*, function names) that are available in the module.

`require` is used to identify a module that the current module (program) depends upon.

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

Creating a module

In full Racket, each `.rkt` file we create is automatically a module. However, none of the functions are accessible outside of the module unless they are `provide`d.

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

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

example: provide

```
(provide function-a function-b)

(define (function-a p) ...)

(define (function-b p1 p2) ...)

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

In this example, the function `hidden-helper` is private and not visible outside of the module.

Scope

`provide` extends the **scope** of a global identifier beyond the module, giving us three "levels" of scope.

- **local:** only visible inside of the "block" or `local` region
- **module:** only visible inside of the module it is defined in
- **program:** visible outside of the module (*i.e.*, `provide`d)

We continue to use the term **"global"** to describe top-level identifiers that have either **program *or* module** scope.

> Because `local`s can be "nested" there can be multiple "levels" of local scope, but in this slide we are generalizing.

example: scope

```
(provide p)

(define p 1)            ; program scope (provided)

(define m 2)            ; module scope (not provided)

(define (f x)
  (define b 3)          ; local scope (within a function)
  ...)
```

In addition, the function `f` has module scope, and its parameter `x` has local scope.

Other courses may use different scope terminologies.

The require special form

When the `require` special form is evaluated, it "runs" all of the code in the required module and makes the `provide`d identifiers or "bindings" available.

`require` also "outputs" the final value of any of the top-level expressions in the module.

In a module you are providing to clients, you should only have definitions, not any top-level expressions.

Module interface

The module ***interface*** is the list of the functions that the module provides, including the contract and purpose for each function. Interfaces may also include structure definitions and variables.

The interface is separate from the module ***implementation***, which is the code of the module (*i.e.,* function **definitions**).

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

> The interface is everything that a client would need to use the module. The client does not need to see the implementation.

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 includes:

- 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.

Racket module interfaces

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

In the implementation, it is not necessary to duplicate the interface documentation for public (program scope) functions that are `provide`d.

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

sum.rkt

```
;; A module for summing numbers [description of module]

(provide sum-first sum-squares)  ;; [list of functions]

;; (sum-first n) sums the integers 1..n
;; sum-first: Int -> Int
;; requires: n >= 1

;; (sum-squares n) ...

;;;;;;;;;; IMPLEMENTATION ;;;;;;;;;;

;; see interface above  [no further info required]
(define (sum-first n) ...)

;; see interface above  [no further info required]
(define (sum-squares n) ...)

;; [purpose & contract for private helper function]
(define (private-helper p) ...)
```

Testing modules

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

For each module you design, you should also create a **testing module** that ensures the `provide`d functions are correct.

test-sum.rkt

```
;; this is a simple testing module for sum.rkt

(require "sum.rkt")

; Each of the following should produce #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)
```

In Section 07 we discuss more advanced testing strategies.

There may be “white box” tests that cannot be tested from outside of your module. These may include implementation-specific tests or tests of private functions.

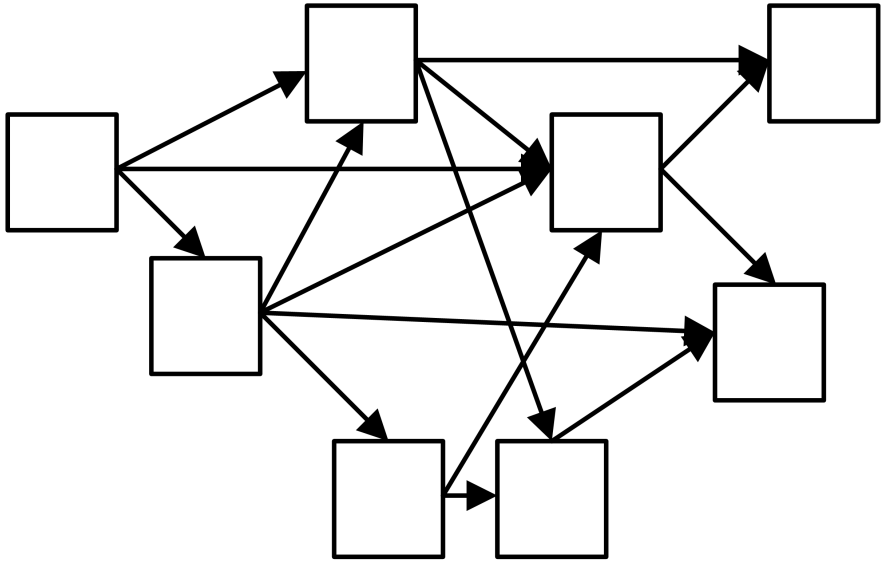
In these circumstances, you can `provide` a `test-module-name` predicate function that produces `#t` if the tests are successful.

Cohesion and coupling

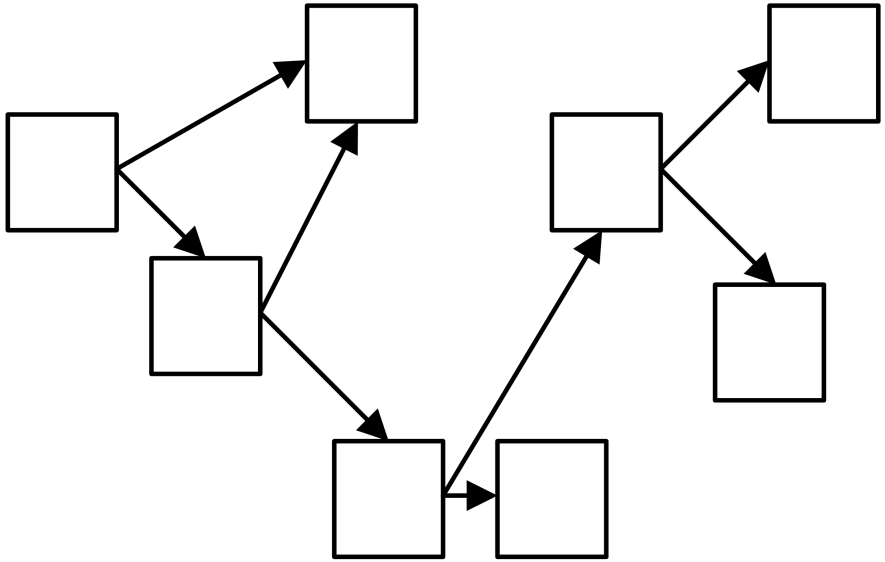
When designing module interfaces, we want to achieve ***high cohesion*** and ***low coupling***.

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.



High coupling



Low coupling

Interface design is especially important when working with large projects. Identifying the appropriate modules and interfaces can be quite challenging.

In this course there is very little interface design.

example: tax module

When designing our Ontario tax module, we should consider:

- **high cohesion:** All of the interface functions should be related to determining tax rates. Including a function for calculating the area of a triangle would be *low* cohesion.
- **low coupling:** The interface of a module should not depend on the "parent" program. Having the tax module call functions from the inventory maintenance module would be *high* coupling.

Interface vs. implementation

Earlier, we made the distinction between the module **interface** and the module **implementation**.

Another important aspect of interface design is ***information hiding***, where the interface is designed to hide any implementation details from the client.

In Racket, the module interface and implementation appear in the same file, so the distinction between interface and implementation may not seem important, but in C (and in many other languages) only the interface is provided to the client.

Information hiding

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.

By hiding the implementation details from the client, we gain the **flexibility** to change the implementation in the future.

example: information hiding (account)

Consider the following module for storing a username and a password in an "account". The only interface functions are for creating the account and verifying the password.

```
(provide create-account correct-password?)

(struct account (username password))

;create-account: Str Str -> Account
(define (create-account u p)
  (account u p))

;correct-password?: Account Str -> Bool
(define (correct-password? a p)
  (equal? (account-password a) p))
```

The definition (`struct account (username password)`) automatically generates the functions `account`, `account?`, `account-username` and `account-password`.

However, the account module **did not** `provide` any of those functions. As a result, the client is unable to "peek" inside of an account to view the password. The client is also unable to "forge" an account and can only create one with the provided `create-account` interface function.

This is an example of achieving **security** through information hiding.

To improve security, full Racket structures are ***opaque*** by default.

Adding `#:transparent` to a `struct` definition makes the structure "not opaque" or *transparent*.

```
(struct account (username password) #:transparent)
```

The only significant difference is that the contents of a transparent structure can be viewed in output (*e.g.,* with `printf`). This is useful while debugging and testing, but not very secure.

Both opaque and transparent structures require field selector functions (*e.g.,* `account-username`) to obtain field values.

To make the username of an account visible to the client, we could `provide` the `account-username` selector function.

However, this exposes the field name "`username`" to the client, which is an implementation detail. Including that detail in the interface is poor information hiding and restricts **flexibility**.

Instead, a "wrapper" function can be provided.

```
;get-username: Account -> String
(define (get-username a)
  (account-username a))
```

For this small Racket example, the change is very subtle. In other languages this difference is more significant.

To illustrate flexibility, we can change the **implementation** without changing the **interface**. There is no difference to the client.

```
(struct account (lst)) ; account now only has 1 field (a list):
                       ;   (list username password)

;; create-account: Str Str -> Account
(define (create-account u p)
  (account (list u p)))

;; correct-password?: Account Str -> Bool
(define (correct-password? a p)
  (equal? (second (account-lst a)) p))

;; get-username: Account -> Str
(define (get-username a)
  (first (account-lst a)))
```

If `account` was `provide`d instead of the `create-account` wrapper, this change would not have been possible.

Data structures & abstract data types

In the previous `account` example, we demonstrated two **implementations** with two different ***data structures***:

- a `struct` with two fields
- a *list* with two elements (wrapped within another `struct`)

When working with a *data structure* we know precisely how the data is "structured" or "arranged".

However, the client doesn't need to know how the data is structured. The client only requires an *abstract* understanding that an `account` stores data (a username and a password).

Formally, an ***Abstract Data Type (ADT)*** is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

However, 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*).

The `account` module is an implementation of an “account ADT” because it stores data that can only be accessed through the module interface functions (operations).

Account is not a “typical” ADT because it stores a fixed amount of data and it has limited use.

Collection ADTs

A ***Collection ADT*** is an ADT designed to store an arbitrary number of items. Collection ADTs have well-defined operations and are useful in many applications.

In CS 135 we were introduced to our first *collection ADT*: a **dictionary**.

In most contexts, when someone refers to an ADT they *implicitly* mean a "collection ADT".

By some definitions, collection ADTs are the *only* type of ADT.

Dictionary (revisited)

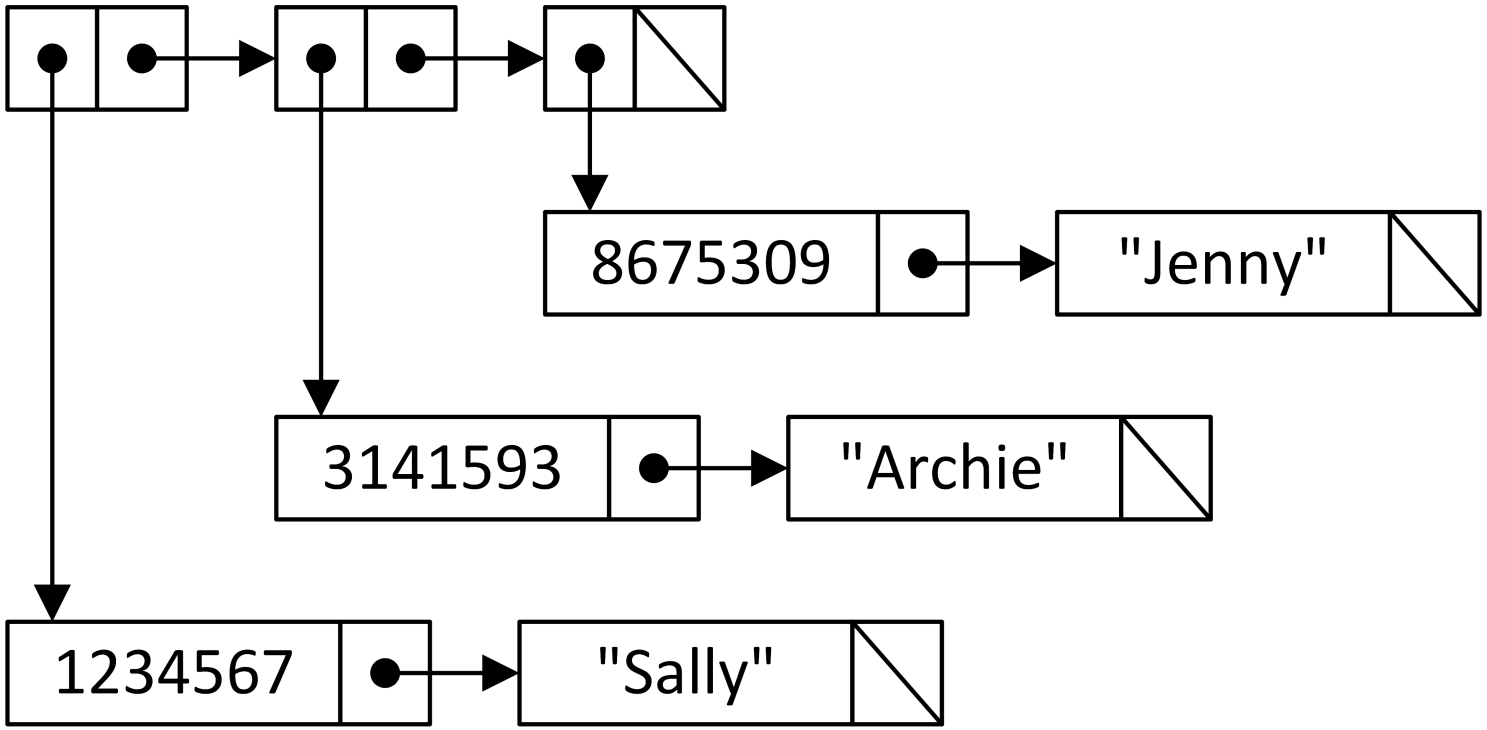
Recall that a ***Dictionary*** stores ***key*** and ***value*** pairs. For a given key, we can `lookup` the corresponding value in the dictionary (lookup is an *operation*).

example: student numbers

key (student number)	value (student name)
`1234567`	`"Sally"`
`3141593`	`"Archie"`
`8675309`	`"Jenny"`

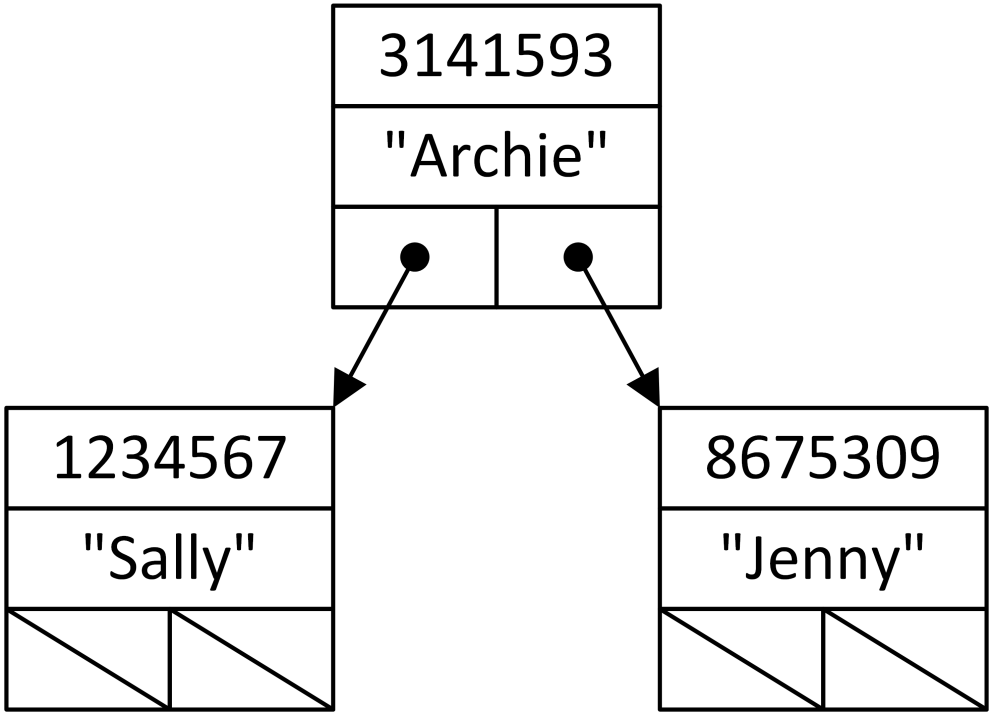
We can implement a dictionary with an ***association list*** *data structure* (a list of key/value pairs with each pair stored as a list of two elements).

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



Alternatively, we can implement a dictionary with a ***Binary Search Tree (BST)*** *data structure*.

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



BSTs are briefly reviewed in Section A.

To *implement* a dictionary, we have a choice:

use an association list, a BST or perhaps something else?

This is a **design decision** that requires us to know the advantages and disadvantages of each choice.

You likely have an intuition that BSTs are “more efficient” than association lists.

In Section 09 we explore what it means to be “more efficient”, and introduce a formal notation to describe the efficiency of a design decision.

We can **use** a dictionary ADT as a client without knowing *how* it is implemented. We are only interested in performing the dictionary operations: `lookup`, `insert` and `remove`.

Recall that hiding the data structure from the client provides us:

- **flexibility:** we can change data structures without the client knowing (perhaps with a more efficient implementation)
- **security:** the client cannot "accidentally" corrupt the data (for example, violating the ordering property of a BST)

Not all modules are designed to store data, and not all data storage modules are ADTs.

While we have demonstrated the advantages of information hiding, it is not always necessary. Transparency may be more important than security or flexibility.

An important aspect of interface design is deciding upon the level of information hiding, and determining if it is appropriate to expose any implementation details in the interface.

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, re-usability and maintainability
- identify two characteristics of a good modular interface: high cohesion and low coupling
- explain and demonstrate information hiding and how it supports both security and flexibility
- explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.

- use `provide` and `require` to implement modules in Racket
- design an ADT module in Racket
- produce good interface documentation, including the new documentation changes introduced