# From the Hut to the Cathedral

## Small hints on modular software development

D. Renault

**ENSEIRB-Matmeca**

March 13, 2019, v. 1.0.1

Software design promotes a set of programming **qualities** (cf. [Booch]) :

- **Abstraction** : the ability to represent parts of a program with a summary of their essential characteristics ;
- **Encapsulation** : the possibility to expose public parts and hide private parts of an abstraction ;
- **Modularity** : the ability to decompose a program into a set of cohesive and loosely coupled components ;

Some goals :

- **Maintenability** : is it easy to modify and extend a program ?
- **Reusability** : is it possible to avoid code duplication ?

Basic example : a library based on sets, developed in **Racket**.

# Types of specifications (1/2)

Code specification may take different forms :

- **Functional** specification :
  - It should be possible to define sets containing any kinds of objects of the same type.
  - It should be possible to add and remove elements from a set, and to tell if an element belongs to a set or not.
- **Non-functional** specification :
  - The implementation should run efficiently on current computers.
  - The implementation should handle correctly sets of arbitrary sizes.

Here comes a problem of validation : how can specifications be checked ?

# Types of specifications (2/2)

- **Interface** / **Type** specification :

```
;; T is the type of the
;; elements inside the set.
type set[T];
```

```
set_empty    : set[T]
set_is_empty : set[T] → boolean
set_add      : set[T]*T → set[T]
set_remove   : set[T]*T → set[T]
set_find     : set[T]*T → boolean
```

- **Formal** specification :

```
set_is_empty(set_empty)          = true
set_is_empty(set_add(s,i))       = false

set_find(set_empty, i)           = false
set_find(set_add(s, i), i)       = true
set_find(set_add(s, i), i')      = set_find(s, i')

set_add(s, i)                    = s [if set_find(s,i)=true]
                                     [unconstrained otherwise]
set_remove(set_empty, i)         = set_empty
set_remove(set_add(s, i), i)     = s
set_remove(set_add(s, i), i')    = set_add(set_remove(s, i'), i)
```

# Modules

## Definition : Module

A module is a construct representing a unit of code (set of types, values, functions and any expression allowed by a language) and satisfying :

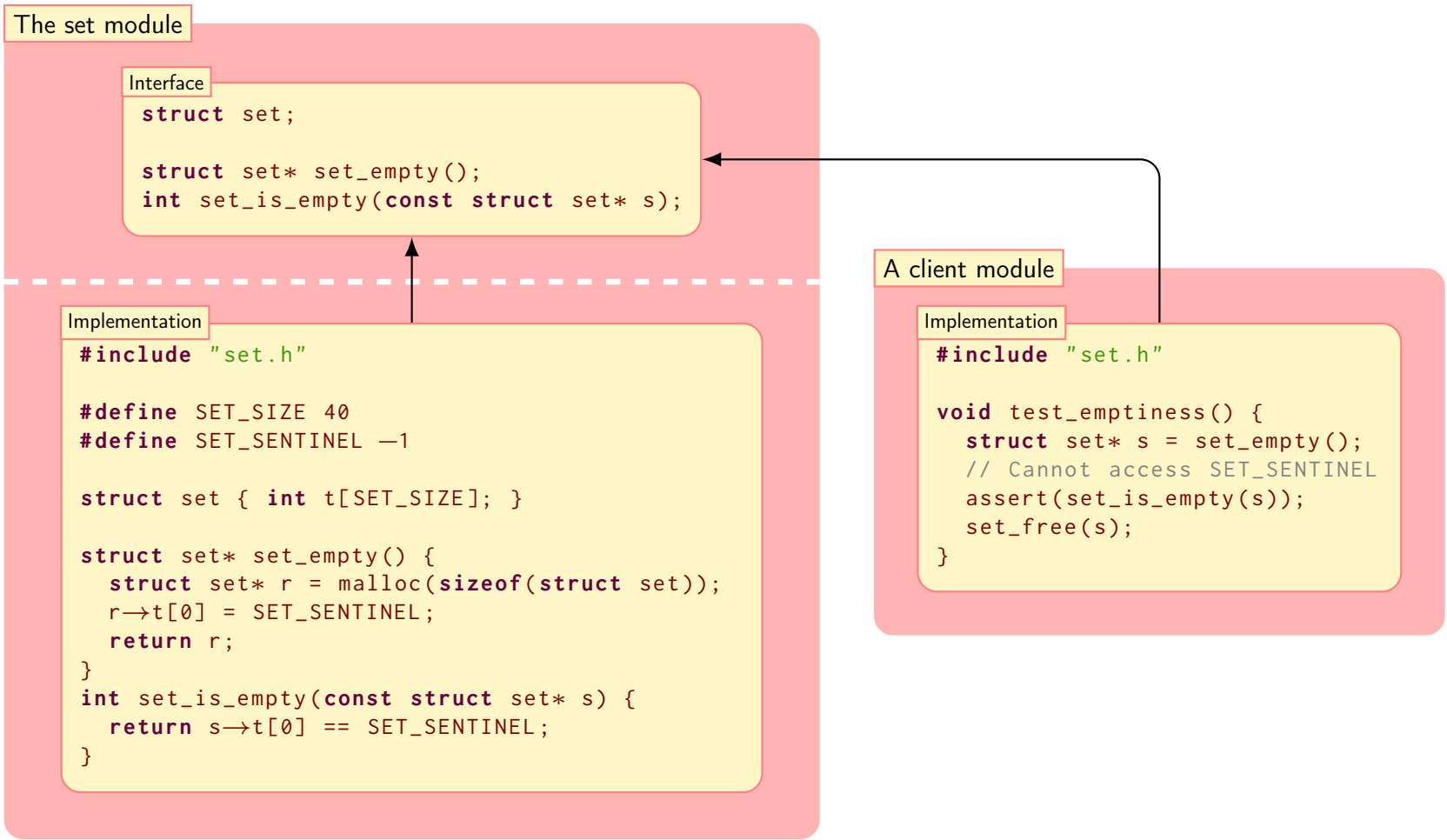
- **Interface** : a module may publicly provide and require a set of components ;
- **Encapsulation** : a module may hide or make abstract some of its components ;

Sets of modules are meant to be connected according to the dependencies induced by their interfaces.

- **Independence** : a module should depend only on the interfaces of its dependencies.

Examples : `.c` and `.h` files in **C**, `module`s in **OCaml**.

# Example : modules in C



# Example : modules in C

The set module

Interface

```
struct set;

struct set* set_empty();
int set_is_empty(const struct set* s);
```

Implementation

```
#include "set.h"

#define SET_SIZE 40
#define SET_SENTINEL -1

struct set { int t[SET_SIZE]; }

struct set* set_empty() {
  struct set* r = malloc(sizeof(struct set));
  r->t[0] = SET_SENTINEL;
  return r;
}
int set_is_empty(const struct set* s) {
  return s->t[0] == SET_SENTINEL;
}
```

A client module

Implementation

```
#include "set.h"

void test_emptiness() {
  struct set* s = set_empty();
  // Cannot access SET_SENTINEL
  assert(set_is_empty(s));
  set_free(s);
}
```

Interface ✓ Encapsulation ✓ Independence ✓

# Modules and modules . . .

Many programming languages provide modules complying only to a subset of these properties :

- Interfaces may provide different levels of verification : names only (**Racket**, **Python**), types (**C**, **OCaml**)
- Encapsulation may not exist (cf. **Python** module privates) ;
- Interfaces must sometimes be shipped with the module (**Racket**, **Haskell**) or live as independent citizens (**C**, **OCaml**).

In the following, we investigate the capabilities of **Racket** in regard to modules.

# “No module” development

## Definition

All the code is written in a single unit, with little specifications.

```
(define (set-empty)       '())
(define set-is-empty?     null?)
(define (set-add l x)     (cons x l))
(define (set-remove l x) (remove x l))
(define (set-find l x)    (member x l))

(set-is-empty? (set-empty))           ;; → #t
(set-find (set-add 1 (set-empty)) 1 ;; → #t
```

Favorable features :

- REPL (simplifies incremental development)
- Dynamic type system (delays verification)
- Extensible language (simplifies writing ad-hoc code)

# Problems

Induces a **monolithic** development :

- parts of the code with **different purposes** are all mixed up in a single file (ex : implementation and tests) ;
- hinders code **reuse** / **modification** of a subset (e.g. modification of the set representation) ;
- complexifies the **verification** of the code (all or nothing) ;
- not adapted to separate compilation, separate testing, team development . . .

# Modular programming

## Modular programming

Break the code into a set of cohesive and loosely coupled modules, that shall be composed depending on the specification.

```
(define set? (...))
(define set_add (...))
(define set_empty (...))
(define set_find (...))
(define test_add (...))
(define test_empty (...))
(define test_find (...))
(define set_recipes (...))
(define find_recipe (...))
(define bartender (...))
```

# Modular programming

## Modular programming

Break the code into a set of cohesive and loosely coupled modules, that shall be composed depending on the specification.

```
(define set? (...))
(define set_add (...))
(define set_empty (...))
(define set_find (...))
(define test_add (...))
(define test_empty (...))
(define test_find (...))
(define set_recipes (...))
(define find_recipe (...))
(define bartender (...))
```

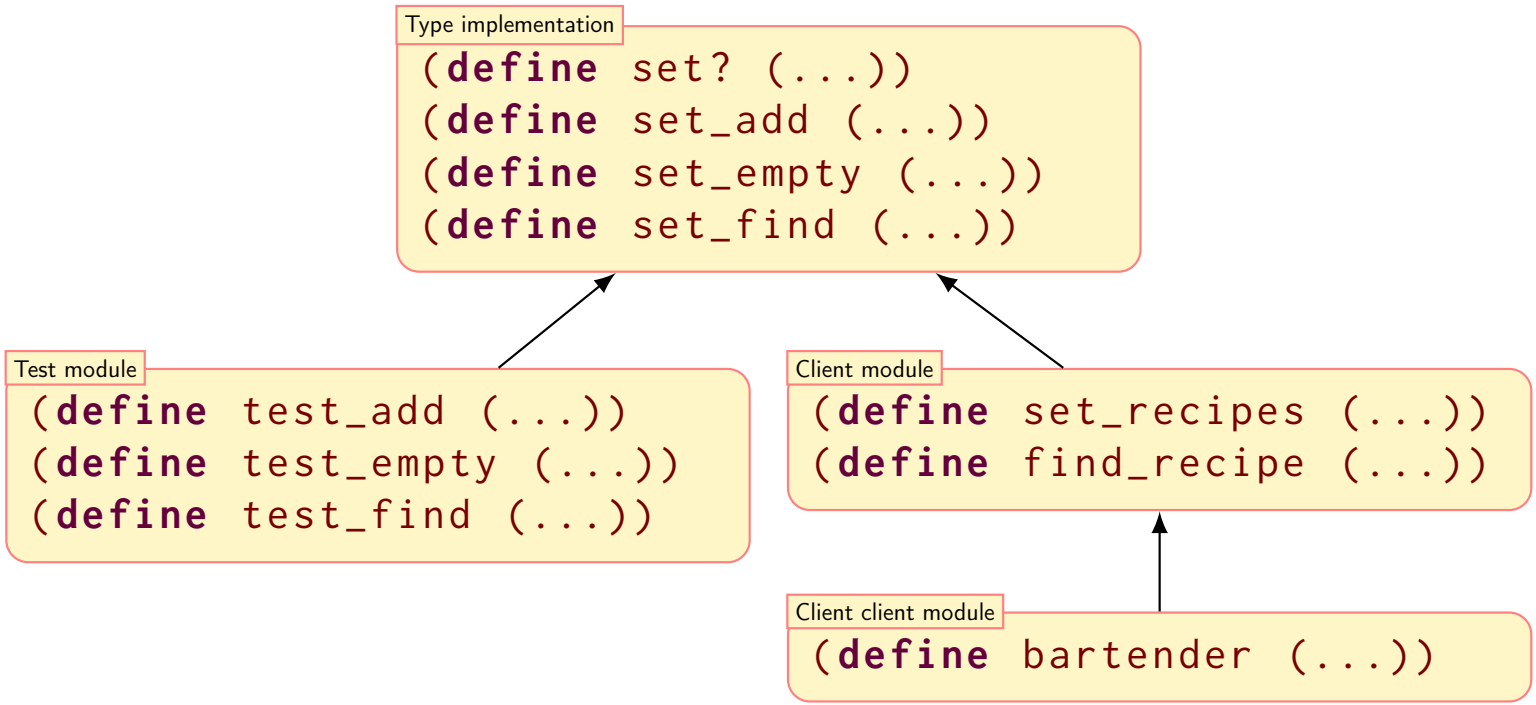
Implementation

Tests

Client code

## Modular programming

Break the code into modules with different responsibilities, that shall be composed depending on the specification.



# Modules in Racket

In Racket, files are modules, and can `require` or `provide` implementations :

set-impl.rkt

```
(require)                          ;; Required module (none)
(provide set? set_empty set_add) ;; Exported functions

(define set? list?))
(define (set_add l x) (cons x l))
(define (set_empty) '())
```

set-test.rkt

```
(require "set-impl.rkt")           ;; Required module

(define (test_add) (eq?  (set_add (set_empty) 1) '(1)))
```

set-client.rkt

```
(require "set-impl.rkt")           ;; Required module
(provide set_recipes add_recipe) ;; Exported functions

(define set_recipes '((1 . goulash) (3 . spatzle)))
(define (add_recipe key sym)
    (set! set_recipes (set_add (cons key sym) set_recipes)))
```

# Summary : modular programming in Racket

Advantages :

- Facilitates **code reuse** : both tests and clients can make use of the same implementation module ;
- Allows **multiple implementations** : the client can select an appropriate implementation module without modification of its code ;
- Ensures **verification** that required functions are effectively provided.

Limitations in **Racket** :

- Verification limited to checking the presence/absence of names ;
- Dependencies of the modules embedded inside the code.

# Predicates and Types

## Definition for the children : Type

A type is a subset of the values of the language.

Example : the boolean type is the set $\{$`true`, `false`$\}$.

## Definition : Predicate

A predicate is a function taking any possible value and returning a boolean.

- A predicate is the characteristic function of a type (e.g. `boolean?`).
- Predicates are usually the sign of a language with dynamical type checking (**Lisp**, **Racket**).

# Racket contracts

## Definition : Contracts

In **Racket**, contracts are predicates that constrain the behavior (precondition, postcondition, invariant) of the functions inside a module.

As for assertions, contracts are checked dynamically.

## Example

Documentation of `list-set` :

```
(list-set lst pos val) → list?
  lst : list?
  pos : (and/c (>=/c 0) (</c (length lst)))
  val : any/c
```

- `pos` must be a valid index in `lst`.

Erroneous usage :

```
(list-set '(1) 2 'one)
```

```
<=: contract violation
  expected: (and (>= 0) (< 1))
  given: 2
```

**Typical languages** : **Eiffel**, **Racket**, **C#** with Code Contracts

# Contracts in Racket

**Racket** contracts are merely particular `provide` forms :

set-impl.rkt

```
(require)
(provide (contract-out
   [set?          (→ any/c boolean?)]
   [set_add       (→i ([s set?] [x any/c])
                      [result (x) (lambda (r) (set_find r x))])]
   [set_find      (→ set? number? boolean?)]
   [set_size      (→i ([s set?])
                      [result (s) (and/c number? positive?)])
                  )]))

(define set? list?)
(define (set_add l x)  (cons x l))
(define (set_find s x) (member x s)) ;; ...
```

set-test.rkt and set-client.rkt remain unchanged.

# Summary : contracts programming in Racket

Advantages :

- Very precise definition of the interactions between modules (contracts may be expressed as Racket functions) ;
- Blends naturally with the Racket standard library.

Limitations :

- The contracts are checked dynamically (undecidability problems).
- Requires tests to verify that the contracts are enforced.

# Typed programming

## Definition : Type system [modified from Pierce]

A type system is a formal method applied to a program which aims at **classifying** elements of the program with types so as to guarantee some correctness of its behavior.

## Example in OCaml

Documentation of `Array.set` :

```
'a array → int → 'a → unit
```

- takes an array, an index and a value ;
- the index is **any** integer ;
- the type of the value must be the same as those inside the array.

Erroneous (static) usage :

```
Array.set [||] "zero" 1;;
```

```
Characters 15–21:
  Array.set [||] "zero" 1;;
                ^^^^^^
Error: This expression has
type string but an expr
was expected of type int
```

# Typed programming

## Definition : Type system [modified from Pierce]

A type system is a formal method applied to a program which aims at **classifying** elements of the program with types so as to guarantee some correctness of its behavior.

## Example in OCaml

Documentation of `Array.set` :

```
'a array → int → 'a → unit
```

- takes an array, an index and a value ;
- the index is **any** integer ;
- the type of the value must be the same as those inside the array.

Erroneous (dynamic) usage :

```
Array.set [||] 1 1;;
```

```
Exception: Invalid_argument
  "index out of bounds".
```

# Sorts of type systems

Depending on the language and its compiler, type systems may come in different flavours :

- **Dynamic** type checking :
  - all verifications are done at runtime,
  - no type annotations necessary.

  Typical languages : **Racket**, **Python**, **Ruby**
- **Static** type checking :
  - all verifications are done at compile time,
  - type annotations are either required (**C**, **Java**) or inferred (**OCaml**, **Haskell**).

# Typed Racket

Vanilla **Racket** possesses a dynamical type system, but the language allows different dialects to be written, one of them being called Typed Racket.

- **Gradual** type checking : type annotations may be added incrementally inside the code, thus mixing annotated and non-annotated expressions.

Example :

```
#lang typed/racket                ;; Choice of the language
(: num-fun (Number → Number));; Type annotation of num-fun (optional)
(define (num-fun n) (add1 n)) ;; Definition with type (checked)
(define (oth-fun n) (sub1 n)) ;; Definition without type (unchecked)
```

# Types in Typed Racket

set-impl.rkt

```
#lang typed/racket
(provide set? set_add set_empty set_is_empty? set_size)

(define-type (Set a) (Listof a))
(: set? (Any → Boolean))
(define set? list?)

(: set_add : (All (a) ((Set a) a → (Set a))))
(define (set_add l x) (cons x l))

(: set_empty : (All (a) (→ (Set a))))
(define (set_empty) '())

(: set_is_empty? : (All (a) ((Set a) → Boolean)))
(define (set_is_empty? l) (null? l))

(: set_size : (All (a) ((Set a) → Number)))
(define set_size length)
```

set-test.rkt and set-client.rkt remain unchanged.

# Summary : typed modular programming in Racket

Advantages :

- The types are checked **statically** ;
- The Typed/Racket type system is pretty expressive.

Limitations in **Racket** :

- Types (as set of values) are **not as expressive** as contracts ;
- Types and contracts libraries in **Racket** do not mix well (for technical reasons).

# Back to interfaces . . .

Separating an interface from its implementation is done via **signatures**.

The set module

Interface

```
(require)
(provide Set set-adt^)

(define-type (Set a) (Listof a))

(define-signature set-adt^
  ([set_add       : (All (a) ((Set a) a → (Set a)))]
   [set_remove    : (All (a) ((Set a) a → (Set a)))]
   [set_empty     : (All (a) (→ (Set a)))]
   [set_is_empty? : (All (a) ((Set a) → Boolean))]
   [set_find      : (All (a) ((Set a) a → Boolean))]
   [set_size      : (All (a) ((Set a) → Number))]
   ))
```

Implementation

```
(require "set-sig.rkt")
(provide set-adt^ set-impl@)

(: ext_set_add    : (All (a) ((Set a) a → (Set a))))
(define (ext_set_add l x)    (sort (cons x l) <))
(: ext_set_remove : (All (a) ((Set a) a → (Set a))))
(define (ext_set_remove l x) (remove x l)) ;; ...

(define-unit set-impl@
  (import) (export set-adt^)
    (define set_add          ext_set_add)
    (define set_remove       ext_set_remove) ;; ...
)
```

A client module

```
(require "set-impl.rkt")
(define-values/invoke-unit set-impl@
    (import) (export set-adt^))
(set_add (set_empty) 1)
(set? (set_empty))
```

# . . .and to abstraction

## Abstract data types

- an abstract and independent representation of a component ;
- possibly multiple different implementations for the same abstraction ;
- possibly multiple clients for the same abstraction.

Complete independence is still missing in Racket :

- the client depends on the implementation and not on the interface ;
- `require` statements instantiate their modules.

Racket lacks truly separate compilation.

# Signatures in other languages

- **Interfaces** : in **Java** (here in a functional manner)

```
interface Set<T> {
    Set<T>  set_add(Set<T> set, T el);
    Set<T>  set_remove(Set<T> set, T el);
    Set<T>  set_empty();
    boolean set_is_empty(Set<T> set);
    boolean set_find(Set<T> set, T el);
    int     set_length(Set<T> set);
}
```

- **Signatures** : in **OCaml**

```
module type SET = sig
  type 'a set
  val set_add      : 'a set → 'a → 'a set
  val set_remove   : 'a set → 'a → 'a set
  val set_empty    : unit   → 'a set
  val set_is_empty : 'a set → bool
  val set_find     : 'a set → 'a → bool
  val set_length   : 'a set → int
end
```

# Some pointers


- G. Booch, *Object-oriented analysis and design*, Addison-Wesley, 1991.
- B. Meyer, *Applying « Design by Contract »*, *Computer*, vol. 25, 1992.
- B. J. Pierce, *Types and Programming Languages*, MIT Press, 2002.