04: Design Recipe

Programs as communication

Every program is an act of communication:

- Between you and the computer
- Between you and your future self
- Between you and others

By writing code, we communicate with the computer to say exactly what it shall do.

For other programmers we want to communicate other things, such as:

- What are we trying to do?
- Why are we doing it this way?

We communicate with other programmers (and ourselves) using comments.

The design recipe

The **design recipe** is an *process* for developing a function.

The design recipe has three main purposes:

1. Writing it helps you **understand the problem.** If you cannot write the design recipe, you probably don’t actually understand the problem.
2. It helps you write **understandable code**.
3. It helps you write **tested code** so you have some confidence it does what is should.

You should use the design recipe for every function you write.

Reading the Design Recipe

See the definition of `(e10 n)`. Carefully read the design recipe.

design recipe

```
;; (e10 n) produce 1 followed by n zeros.
;; Examples:
(check-expect (e10 2) 100)
(check-expect (e10 5) 100000)
(check-expect (e10 0) 1)
;; e10: Nat -> Nat
```

definition

```
(define (e10 n)
```

(The function definition is correct, but will not work in the *Beginning Student* language. It is intentionally hard to read. **Read the design recipe only!**)

> The five design recipe components

Purpose: Describes what the function is to compute.

Examples: Illustrating the typical use of the function.

Contract: Describes what type of arguments the function consumes and what type of value it produces.

Definition: The Racket definition of the function (**header** & **body**).

Tests: A representative set of function applications and their expected values. Examples also serve as Tests.

> Order of execution

The order in which you carry out the steps of the design recipe is very important. Use the following order:

1. Write a draft of the Purpose
2. Write Examples (by hand, then using `check-expect`)
3. Write Definition Header & Contract
4. Finalize the purpose with parameter names
5. Write Definition Body
6. Write additional tests, if required

Applying the design recipe

Purpose (first draft):

```
;; produce the sum of the squares of two numbers
```

Examples:

$$3^2 + 4^2 = 9 + 16 = 25$$

```
;; Examples:
(check-expect (sum-of-squares 3 4) 25)
```

> Applying the design recipe (cont)

Header & Contract:

```
;; sum-of-squares: Num Num -> Num
(define (sum-of-squares n1 n2)
```

Purpose (final draft):

```
;; (sum-of-squares n1 n2) produces the sum of squares of n1 and n2.
```

> Applying the design recipe (cont)

Write Function Body:

```
(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))
```

Write Tests:

```
;; Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

> Applying the design recipe (final result)

```
;; (sum-of-squares n1 n2) produces the sum of squares of n1 and n2.
;; Examples:
(check-expect (sum-of-squares 3 4) 25)

;; sum-of-squares: Num Num -> Num
(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))

;; Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

Tests

- Tests should be written later than the code body.
- Tests can then handle complexities encountered while writing the body.
- Tests don’t need to be “big”.
  In fact, they should be small and directed.
- The number of tests needed is a matter of judgement.

! **Do not** attempt to figure out the expected answers to your tests by running your program! Always work them out **independently**.

> Testing functions

In addition to `check-expect`, the teaching languages offer two other testing tools:

```
(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 .001)
(check-error (/ 1 0) "/: division by zero")
```

Tests written using `check-expect`, `check-within`, and `check-error` are saved and evaluated at the very end of your program.

The vast majority of your examples and tests will be written with `check-expect`. `check-within` will usually be used only for inexact values.

Examples vs. Tests

Examples	Tests
Computes a result and compares it to an expected value.	Computes a result and compares it to an expected value.
Uses `check-expect`, `check-within`, or `check-error`.	Uses `check-expect`, `check-within`, or `check-error`.
Tests the correctness of the code.	Tests the correctness of the code.
Are placed between purpose and contract.	Are placed after the function definition.
Written before the code is written.	Written after the code is written.
Derived from the function’s purpose only.	Takes into account the actual code and more knowledge of what can go wrong.
Shows typical uses of the function.	Focuses on more unusual, complex, or error-prone cases.
Also called “closed-box tests”	Also called “open-box tests”

Contracts

Contracts list the types of data consumed by a function (in the same order as the parameters), an arrow, and the type of data produced by the function.

Example:

```
;; sum-of-squares: Num Num -> Num
(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))
```

Types in Contracts

In contracts, we'll use the following types:

Type	Description and Examples
`Num`	Numbers: `3`, `-22/7`, π, `#i3.141592653589793`
`Int`	Integers: `-3`, `0`, `3`
`Nat`	Natural numbers: `0`, `1`, `2`
`Bool`	Booleans: `true`, `false`
`Char`	Characters: `#\A`, `#\a`
`Str`	Strings: `"Hello, world!"`
`Sym`	Symbols: `'earth`, `'female`
`Any`	Values of any type

We will be adding to this list throughout the term.

Ex. 1 Write purpose, contract, examples, and tests for the absolute value function `abs`.

> Additional contract requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section to “extend” the contract.

```
;; (my-function a b c) ...
;; Examples: ...

;; my-function: Num Num Num -> Num
;; Requires: 0 < a < b
;;           c must be non-zero
(define (my-function a b c) ...)

;; Tests: ...
```

There is no formal notation for the **requires** section. Aim for clarity and brevity. Mathematical notation is nice where it makes sense but is not required.

Ex. 2

Consider the function:

```
;; (sqrt-shift x c) produce the square root of (x - c).
;; Examples:
(check-expect (sqrt-shift 7 3) 2)
(check-expect (sqrt-shift 125 4) 11)

;; sqrt-shift: Num Num -> Num


(define (sqrt-shift x c)
  (sqrt (- x c)))
```

What inputs are invalid?

Write a `requires` section for this function.

> Contract enforcement

Contracts are important in keeping us unconfused. However, in Racket they are only human-readable comments and are not enforced by the computer.

In many programming languages, contracts are a part of the language and certain classes of errors are always caught by the computer before the program is allowed to run. This has many advantages but also eliminates some programming techniques that we will use.

We can also create functions that check their arguments to catch type errors more gracefully.

Unless stated otherwise, **you may assume that all arguments provided to a function will obey the contract** (including in our automated testing).

Design recipe style guide

CS135 has a style guide that you are expected to adhere to. It includes specifics about the Design Recipe.

Note that in these slides, sections of the design recipe are often omitted or condensed because of space considerations.

Consult the course style guide before completing your assignments.

Style guides are required in industry!

Students sometimes consider the design recipe as an afterthought, as “something annoying they make you do in school”. It’s not.

For reference, take a look at the *Google C++ Style Guide*.

Design Recipe Summary

The Design Recipe:

- provides a starting point for solving the problem.
- helps you understand the problem better.
- helps you write correct, reliable code.
- improves readability of your code.
- prevents you from losing marks on assignments!

Goals of this module

- You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.
- You should start to use the design recipe and appropriate coding style for all Racket programs you write.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`*` `+` `-` `/` `<` `<=` `=` `>` `>=` `abs` **`and`** `boolean?` `ceiling` `check-error` `check-expect` `check-within` **`cond`** `cos` **`define`** `e` **`else`** `even?` `exp` `expt` `floor` `integer?` `log` `max` `min` `modulo` `negative?` `not` `number->string` `number?` `odd?` **`or`** `pi` `positive?` `quotient` `remainder` `round` `sgn` `sin` `sqr` `sqrt` `string-append` `string-downcase` `string-length` `string-lower-case?` `string-numeric?` `string-upcase` `string-upper-case?` `string<=?` `string<?` `string=?` `string>=?` `string>?` `string?` `substring` `symbol=?` `symbol?` `tan` `zero?`