The design recipe

Readings:

- HtDP, section 2.5
- Thrival and Style Guides

Topics:

- Programs as communication
- The design recipe
- Using the design recipe
- Tests
- Contracts

Programs as communication

Every program is an act of communication:

- Between you and the computer
- Between you and yourself in the future
- Between you and others

Human-only comments in Racket programs:
from a semicolon (**;**) to the end of the line.

> Some goals for software design

Programs should be:

- compatible
- composable
- correct
- durable
- efficient
- extensible
- flexible
- maintainable
- portable
- readable
- reliable
- reusable
- scalable
- testable
- usable
- useful

The design recipe

The **design recipe** is an approach to developing a function.

- Use it for every function you write in CS 135.
- It leaves behind a written explanation of the development
- It results in a trusted (tested) function which future readers (you or others) can understand

> The five design recipe components

Purpose: Describes what the function is to compute.

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

Examples: Illustrating the typical use of the function.

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 code)
3. Write Definition Header & Contract
4. Finalize the purpose with parameter names
5. Write Definition Body
6. Write Tests

Using 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)
```

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.
```

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)
```

> Using 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)
```

Exercise 1

Carefully read the design recipe for `(e10 n)`.

What should this produce? `(+ (* 4 (e10 3)) (* 2 (e10 2))) ; ⇒ ?`

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

(define (e10 n)
  ((lambda (+ -)      (- + -)      ; <-- Tie fighter!
   ) n (lambda (+ /) (cond [(= + 0) 1][else (* 10 (/ (- + 1) /))]))))
```

(The code is correct, but will not work until we get to *Intermediate Student with lambda*. It is intentionally hard to read. **Read the design recipe only!**)

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 and examples needed is a matter of judgement.
- **Do not** figure out the expected answers to your tests by running your program! Always work them out **independently**.

The teaching languages offer convenient testing methods:

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

`check-within` should only be used for inexact values.

Tests written using these functions are saved and evaluated at the very end of your program.

This means that examples can be written as code.

> Examples vs. tests

Examples and tests are the same:

- Both compute a result and compare it to an expected value.
- Both use `check-expect`, `check-within`, or `check-error`.
- Both help test the correctness of the function.

Examples and tests are different:

- Examples show typical uses of the function; tests may focus on more unusual, complex, or error-prone cases.
- Examples are derived from the function's purpose only; tests take into account the actual code and more knowledge about what can go wrong.

Contracts

- We will be more careful than HtDP and use abbreviations.
  - `Num`: any Racket numeric value (e.g. 2, 3.25, $\frac{22}{7}$)
  - `Int`: restriction to integers (e.g. -5, 0, 3)
  - `Nat`: restriction to natural numbers (e.g. 0, 1, 2, 3, ...)
  - `Any`: any Racket value
- We will see more types soon.

Use the most specific type available.

Exercise 2

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) ...
;; my-function: Num Num Num → Num
;; requires: 0 < a < b
;;           c must be non-zero
```

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.

Exercise 3

Consider the function:

```
;; (sqrt-shift x c) produce the square root of (x - c).
;; sqrt-shift: Num Num → Num


;; Examples:
(check-expect (sqrt-shift 7 3) 2)
(check-expect (sqrt-shift 125 4) 11)

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

What inputs are invalid?

Write a **requires** section for this function.

Racket does not enforce contracts, which are just comments, and ignored by the machine.

Each value created during the running of a program has a type (integer, Boolean, etc.).

Types are associated with values, not with constants or parameters.

```
(define p 5)
(define q (mystery-fn 5))
```

This is known as **dynamic typing**. When a function is applied to a value with an inappropriate type, the error is found when that code is executed (which may be years in the future).

Many other mainstream languages use **static typing** in which constants, parameters and values all have specified types. Constants and parameters of one type may not hold a value of an incompatible type.

With static typing, the header of a function might look like this:

```
foo(c:Num, g:Nat):Int
```

Here the contract is part of the language.

A program containing the function application `foo(65, 0.333)` would be illegal because 0.333 is not a `Nat`.

Dynamically typed languages used for developing large programs often develop a static type system or tools that mimic one.

> Contract enforcement

Dynamic typing is a potential source of both flexibility (as we will see) and confusion.

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

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

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

Design recipe style guide

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.

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.