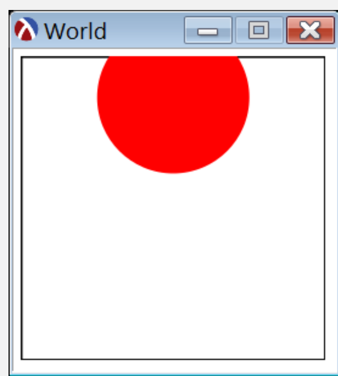
CS450

“Big Bang”, Testing, Contracts

UMass Boston Computer Science

~~Thursday, February 6, 2025~~

Tuesday, February 11, 2025



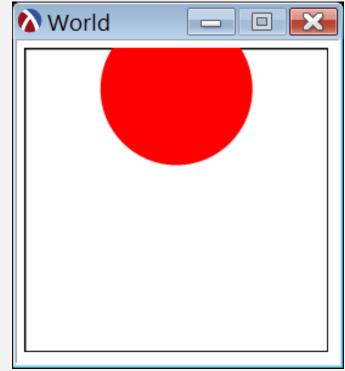
Logistics

- HW 1 in, HW 0 grades out
  - Questions / complaints must use gradescope re-grade request
- HW 2 out
  - due: Tue 2/18 11am EST

- No HW questions by email! (I may not see it)
  - Post to piazza (use **private** or **anonymous** if unsure) (I may change)
  - Make it easier for staff to check one place

- **"Autograder error???"** (not allowed)
  - This class is about learning to communicate (e.g., ask questions) effectively!
  - See forum rules

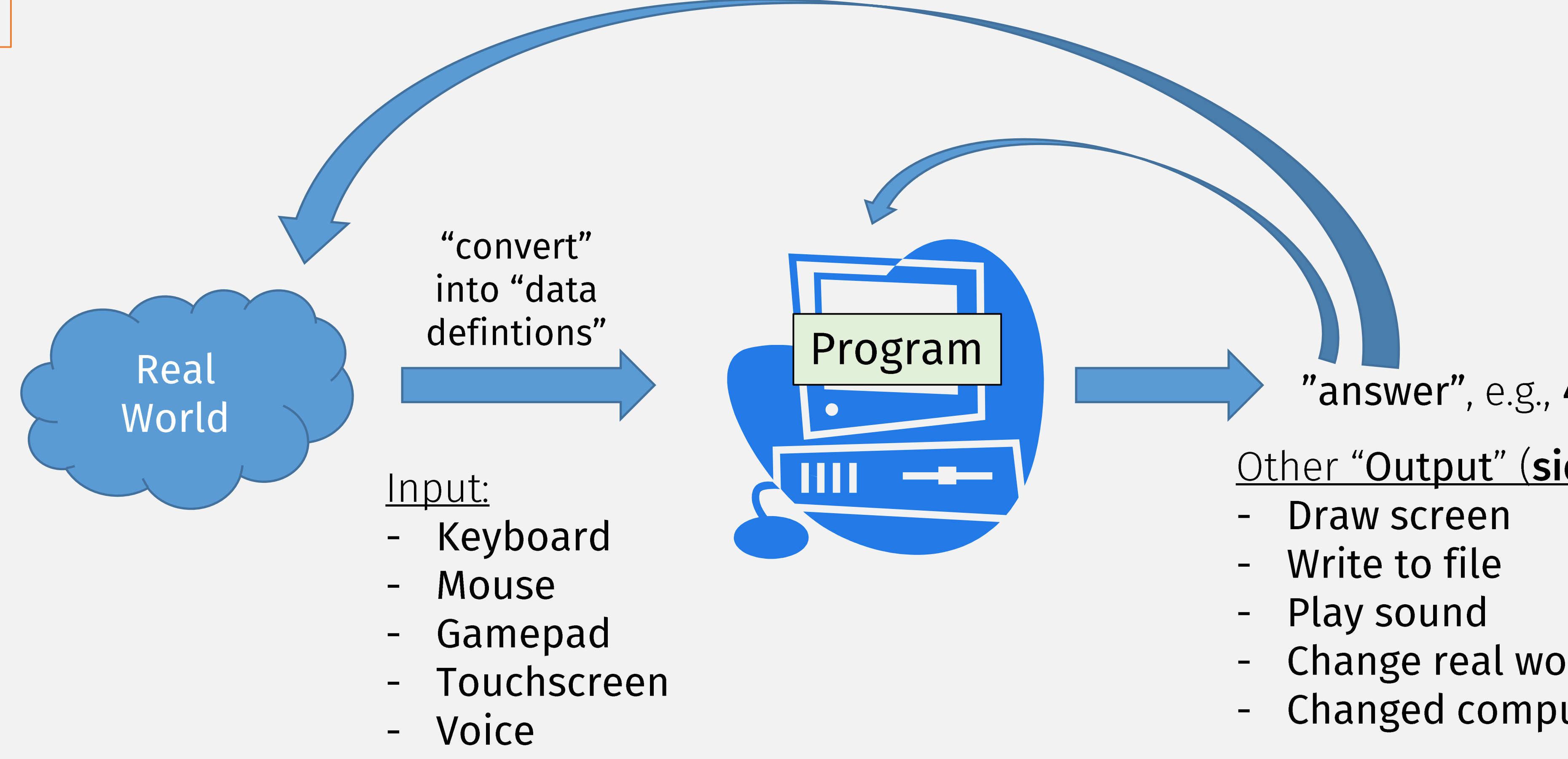
- Course web site:
  - Added Design Recipe section
  - Lecture code (see lecture04.rkt) may occasionally be posted



Last Time

Programs can be Interactive

More fun to write and use!



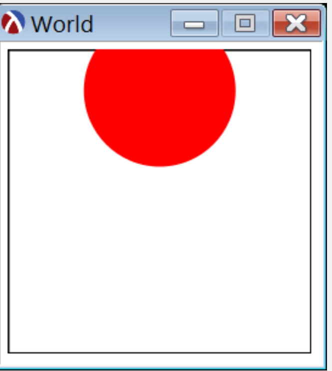
Input:
- Keyboard
- Mouse
- Gamepad
- Touchscreen
- Voice
- File

Other “**Output**” (**side effect**):
- Draw screen
- Write to file
- Play sound
- Change real world (unlock door)
- Changed computer state (game)

Interactive Programs (with **`big-bang`**)

`(require 2htdp/universe)`

- DEMO

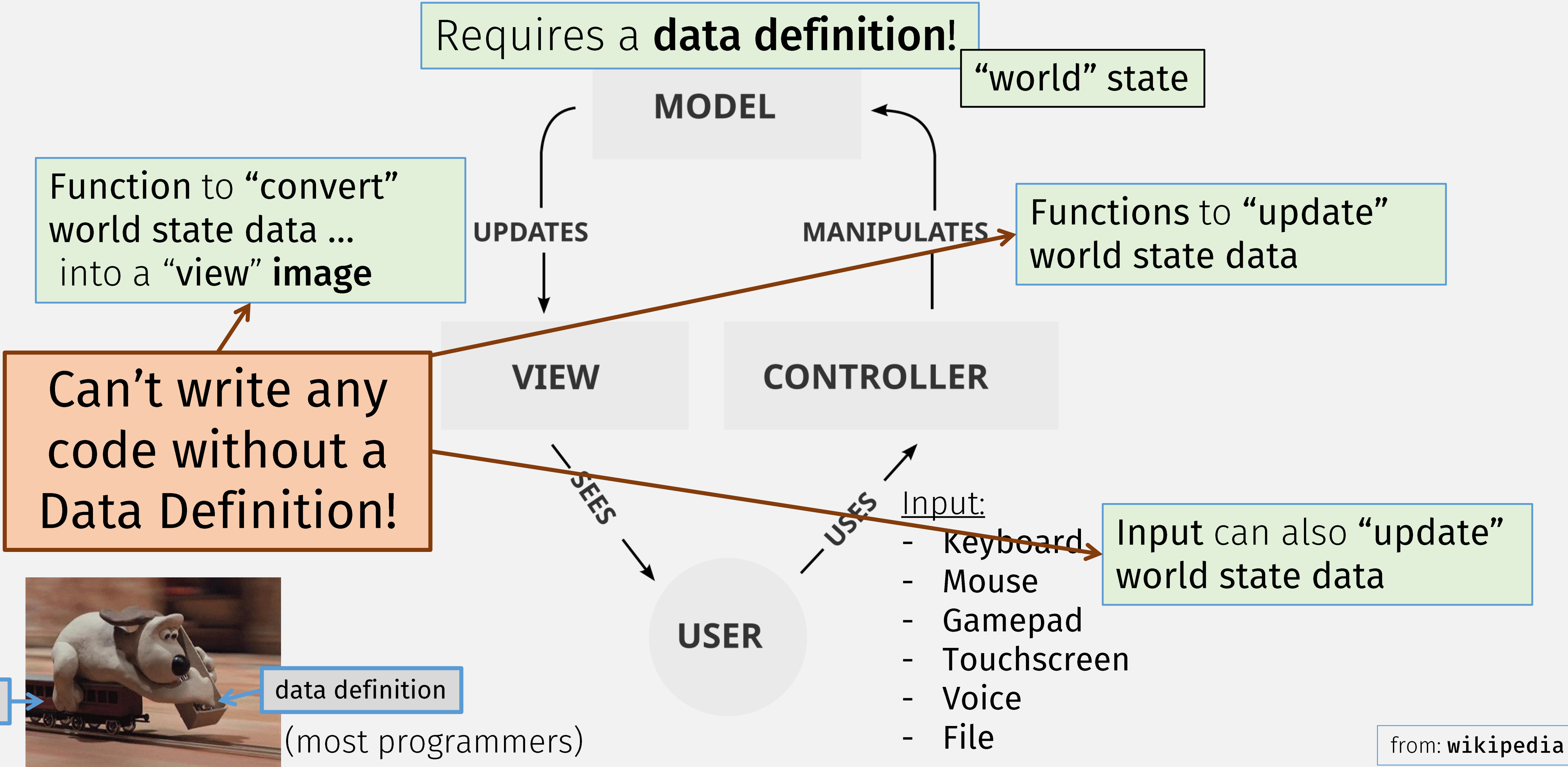


Interactive Programs (with `big-bang`)

```
(require 2htdp/universe)
```

- **`big-bang`** **starts an** (MVC-like) **interactive loop**

Model-View-Controller (MVC) Pattern



Interactive Programs (with `big-bang`)

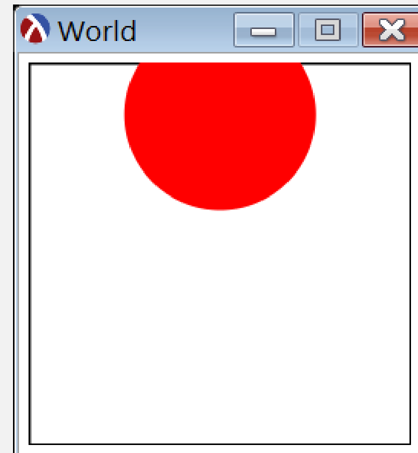
`(require 2htdp/universe)`

- **`big-bang`** starts an (MVC-like) interactive loop
  - repeatedly updates a "world state"
  - Programmer must define what the "World" is …
  - … with a **Data Definition!**

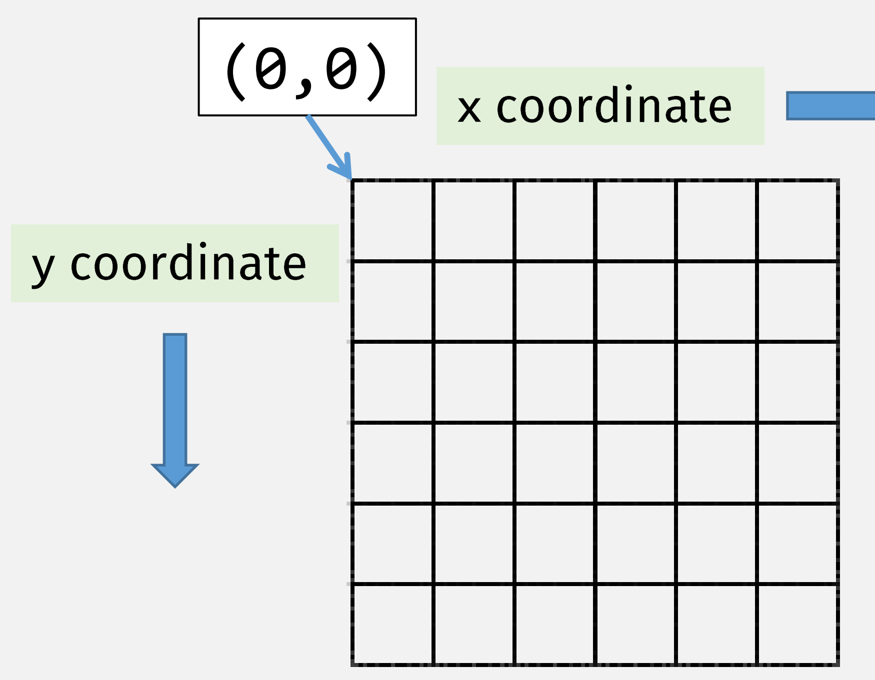
```
;; A WorldState is a Non-negative Integer
;; Interp: y-coordinate of a circle center, in a big-bang animation
```

Data Definitions should represent values that change

(Values that don't change should be defined as constants)



Interlude: htdp universe coordinates



(**place-image** *image x y scene*) → image?　　procedure
- *image* : image?
- *x* : real?
- *y* : real?
- *scene* : image?

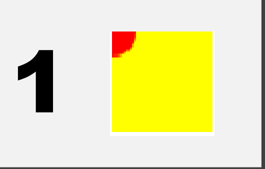
Places *image* onto *scene* with its center at the coordinates (*x*,*y*) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

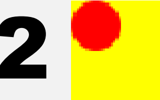
(**circle** *radius mode color*) → image?
- *radius* : (and/c real? (not/c negative?))
- *mode* : mode?
- *color* : image-color?

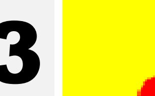
(**square** *side-len mode color*) → image?
- *side-len* : (and/c real? (not/c negative?))
- *mode* : mode?
- *color* : image-color?

```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???







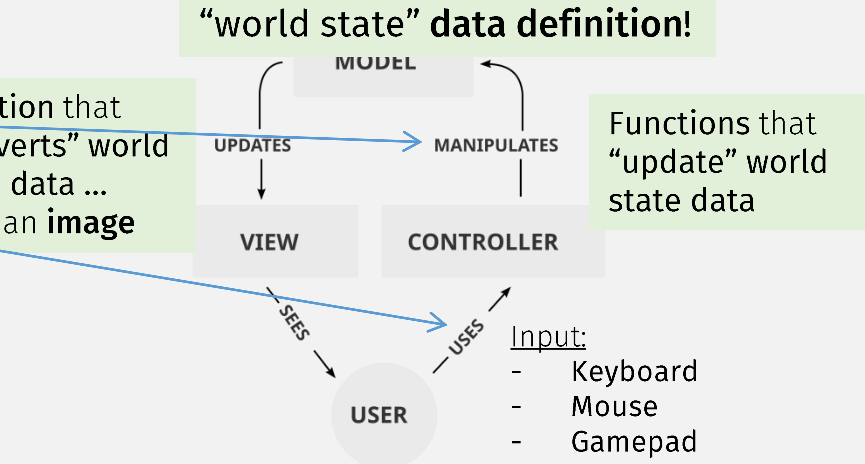
Interactive Programs (with **big-bang**)

`(require 2htdp/universe)`

- **big-bang** starts an (MVC-like) interactive loop
  - repeatedly updates a "world state"
  - Programmer must define what the "World" is ...
  - ... with a Data Definition!

```
;; A WorldState is a Non-negative Integer
;; Interp: y-coordinate of a circle center,
in a big-bang animation
```

- Programmers specify "handler" functions to manipulate "World"
  - Render
  - World update
  - Input handlers

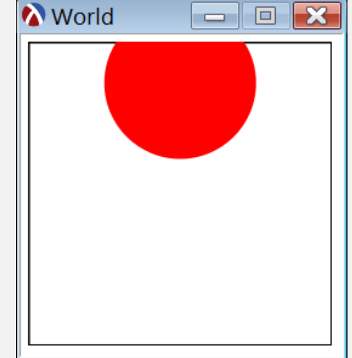


Design Recipe Intro: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has **4** parts:
  1. A defined **Name**
  2. Description of **all possible values** of the data
  3. An **Interpretation** explains the real world concepts the data represents

```
;; A WorldState is a Non-negative Integer
;; Interp: y-coordinate of a circle center,
in a big-bang animation
```



Last Time

Design Recipe Intro: Data Design

Create **Data Definitions**

- Describes the types of data that the program operates on
- Has 4 parts:
  1. A defined **Name**
  2. Description of **all possible values** of the data
  3. An **Interpretation** explains the real world concepts the data represents
  4. A **predicate** is code that checks if a value is in the Data Definition
     - returns **false** if a given value is not in the data definition

```
;; A WorldState is a Non-negative Integer
;; Interp: y-coordinate of a circle center,
in a big-bang animation
```

```
(define (WorldState? x)
  (exact-nonnegative-integer? x))
```

World

Design Recipe

1. **Data Design**
2. **Function Design**

Designing Functions

1. **Name**
2. **Signature**
3. **Description**
4. **Examples**
5. **Code**
6. **Tests**

Designing Functions

1. **Name**
2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (**create new data defs**, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works

Designing Functions

"built-in" data def (from 2htdp/image lib)

```
;; render: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
```

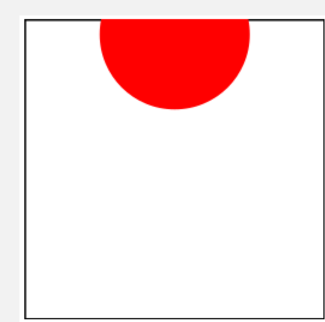
1. **Name**
2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (**create new data defs**, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works

FAQ: What about "error-checking"?

"Error handling is important, but if it obscures logic, it's wrong."
— **Robert C. Martin,** Clean Code: A Handbook of Agile Software Craftsmanship

Designing Functions

```
;; render: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
```



1. **Name**
2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (create new data defs, if needed)
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
   - (put after function definition)
5. **Code** – implement how the functi
6. **Tests** – check (using ra

Style: constant names are in ALL-CAPS

```
(check-equal?
 (render INITIAL-WORLDSTATE)
 (place-image
  BALL-IMG
  BALL-X INITIAL-WORLDSTATE
  EMPTY-SCENE))
```

```
(define (render w)
  (place-image
   BALL-IMG
   BALL-X w
   EMPTY-SCENE))
```

Examples come before (and **help to write**) **Code!**

FAQ: What about "error-checking"?

Designing Functions

1. Name

```
;; render: WorldState -> Image
;; Draws a WorldState as a 2htdp/image Image
```

This declares that the function cannot be given a non-WorldState argument!

... but we can make it more robust

2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (create new data defs, if needed)

The **Signature** *is* **error-checking**

3. Description – explain (in English ...) how the function works

```
> (render "bad arg")
place-image: expects a real number as third argument, given "bad arg"
```

It's **the user's fault** if they **call the function incorrectly**

4. Examples – show (using rackunit) how the function works

BUT: This is a **bad error message** because ... ☹

... it **reveals internal details** that the **user doesn't** (and shouldn't have to) **know about**

5. Code – implement how the function ...

6. Tests – check (using rackunit) that the function works

More Robust Signatures

1. Name

```
;; render: WorldState -> Imag
;; Draws a WorldState as a 2h
```

2. **Signature** – types of the function inpu
   - Refer to Data Definitions (create new data
   - Use `define/contract` with predicates!

3. **Description** – explain (in E

4.

5.

6.

NOTE:
Different languages may have different "signature" or "error handling" mechanisms
- Contracts
- Types
- Asserts
- Try-Catch-Throw
- "return zero"

But the **Design Recipe** is language-agnostic

It can be used no matter what language you're programming in

Function contract

```
(define/contract (render w)
  (-> WorldState? image?)
  (place-image
   BALL-IMG
   BALL-X w
   EMPTY-SCENE))
```

```
> (render "bad arg")
render: contract violation
expected: WorldState?
given: "bad arg"
in: the 1st argument of
    (-> WorldState? image?)
contract from: (function render)
blaming: C:\Users\stchang\Documents\teaching\CS450\Fall23\lecture04.rkt
 (assuming the contract is correct)
at: C:\Users\stchang\Documents\teaching\CS450\Fall23\lecture04.rkt:37:18
```

Good error message: precise, and no internal details! ☺

Designing Functions

1. **Name**
2. **Signature** – types of the function input(s) and output
   - Refer to Data Definitions (**create new data defs**, if needed)
   - Use `define/contract` **with predicates!**
3. **Description** – explain (in English prose) how the function works
4. **Examples** – show (using `rackunit`) how the function works
5. **Code** – implement how the function works
6. **Tests** – check (using `rackunit`) that the function works
   - put in **separate test-suite (file)**

Homework Testing

All HW submissions must include **tests.rkt**, which:

- requires the hw code file, e.g., **hw1.rkt**
- defines a **rackunit test-suite** called **TESTS**
- provide **TESTS**
- includes sufficient **test-cases** (from the **Design Recipe**) for every hw function definition
- runs without error!

hw0-Chang-Stephen/tests.rkt

```
1   #lang racket
2
3   (require rackunit
4            "hw0.rkt")
5
6   (provide TESTS)
7
8   (define TESTS
9     (test-suite
10     "hw0 test suite"
11
20    (test-case
21     "Exercise 1: (dist 1)"
22     (check-equal? ((HW0 dist) 1) 9.9))
23
24    (test-case
25     "Exercise 2: naive pluralize empty"
26     (check-equal? ((HW0 naive-pluralize) "") "s"))
27
28
29  (module+ main
30    (require rackunit/text-ui)
31    (run-tests TESTS 'verbose))
```

Used by (Auto)grader

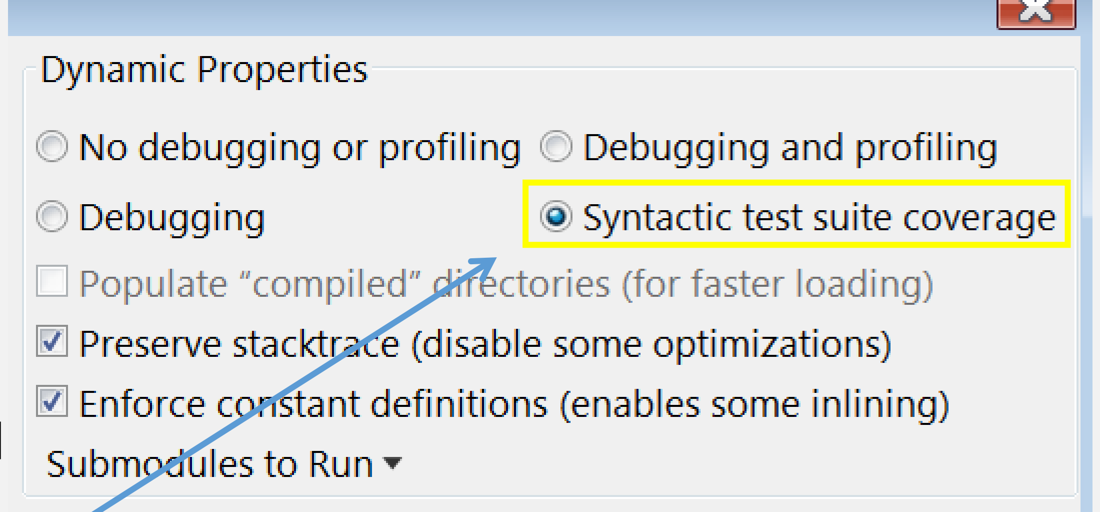
(See **rackunit** docs for more testing functions)

e.g., **check-exn** for fail test cases!

Used for your own testing

What is a “Sufficient” Number of Tests?

- Wishful: **test every possible input**
  - Usually **impossible: infinite cases**
  - Also **redundant …**
- Realistic: **test all “classes” of inputs**
  - **“class” depends on data defs!**
  - E.g., **“positive” / “negative”, “left” / “right”, valid**
  - Try to think of **corner cases !**
- Minimum: **100% (Test / Example) “Coverage”**
  - **All code is run once by some test**
  - In **“Choose Language” Menu**
  - NOTE: **only works with single files**
  - **Doesn’t guarantee “correctness”!** (why?)
- Ideally: **Until 100% confident in “correctness”**



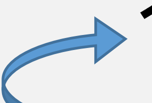
This code was not run

```
;; YCoord is either
;; - before target
;; - in target
;; - after target
;; - out of scene
(define (PENDING-Note? n) (PENDING? (Note-state n)))
(define (HIT-Note? n) (HIT? (Note-state n)))
(define (MISSED-Note? n) (MISSED? (Note-state n)))
(define (OUTOFSCENE-Note? n) (OUTOFSCENE? (Note-state
(define out-Note? OUTOFSCENE-Note?)

;; NEW
;; A WorldState is a List<Note>

(define (num-Notes w) (length w))
```

Design Recipe



1. **Data Design**
2. **Function Design**

Programming is an **iterative** process!

Each iteration should be **incremental**!

Last Time

The *faster* way to program

The Incremental Programming Pledge

At all times, all of the following should be `true` of your code:

1. Comments (**data defs, signatures**, etc) **match code**
2. **Code has no syntax errors**
    1. E.g., **missing / extra parens**
3. **Runs without runtime errors / exceptions**
    1. E.g., **use undefined variables,** `div by zero`, **call a "non function"**
4. **All tests pass**

When you **make a code edit** that **renders one of the above** `false`, **STOP …**

… and **don't do anything else** until **all the statements are true again.**

(this way, it's easy to revert back to a "working" program)

Incremental Programming, in Action

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference “defined” Data Definition names
3. Description
4. Examples
5. Code
6. Tests

```
;; c2f: TempC -> TempF
;; Converts a Celsius temperature to Fahrenheit
```

2. Start with “placeholder” code
(but do not submit this!)

```
(define (c2f ctemp)
  (case
   [(0) 32]
   [(100) 212]
   [(-40) -40]))
```

1. Make Examples runnable tests

```
; (c2f 0) => 32
; (c2f 100) => 212
; (c2f -40) => -40
```

```
(check-equal? (c2f 0) 32)
(check-equal? (c2f 100) 212)
(check-equal? (c2f -40) -40)
```

Incremental Programming, in Action

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference "defined" Data Definition names
3. Description
4. Examples
5. Code
6. Tests

```
;; c2f: TempC -> TempF
;; Converts a Celsius temperature to Fahrenheit
```

1. Make Examples runnable tests

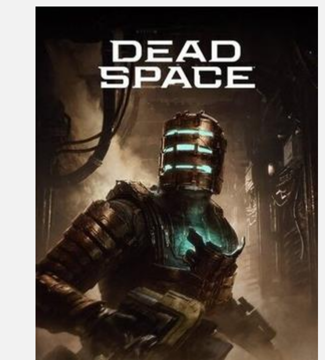
2. Start with "placeholder" code

3. Make small changes only (something easy to revert)

```
(define (c2f ctemp)
  (+ (* ctemp (/ 9 5)) 32))
```

4. Test each (small) change (before making another one)

Incremental Programming: Real-World Example



- https://www.youtube.com/watch?v=1SlGgCxJa3w
- "when you do everything at once …
  you're not sure why it's not working!"
- "when you layer it, when you break it down …
  and you hit a spot when it's not working …
  then you can just focus on that spot!"

3. **Make small changes only** (something easy to revert)

5. Code
6. Tests

4. **Test each** (small) **change** (before making another one)

In-class Office Hours

- Get HW 0 / HW 1 “working”?
- Add `tests.rkt` with `test-suite` named `TESTS` to HW1
- Start HW 2