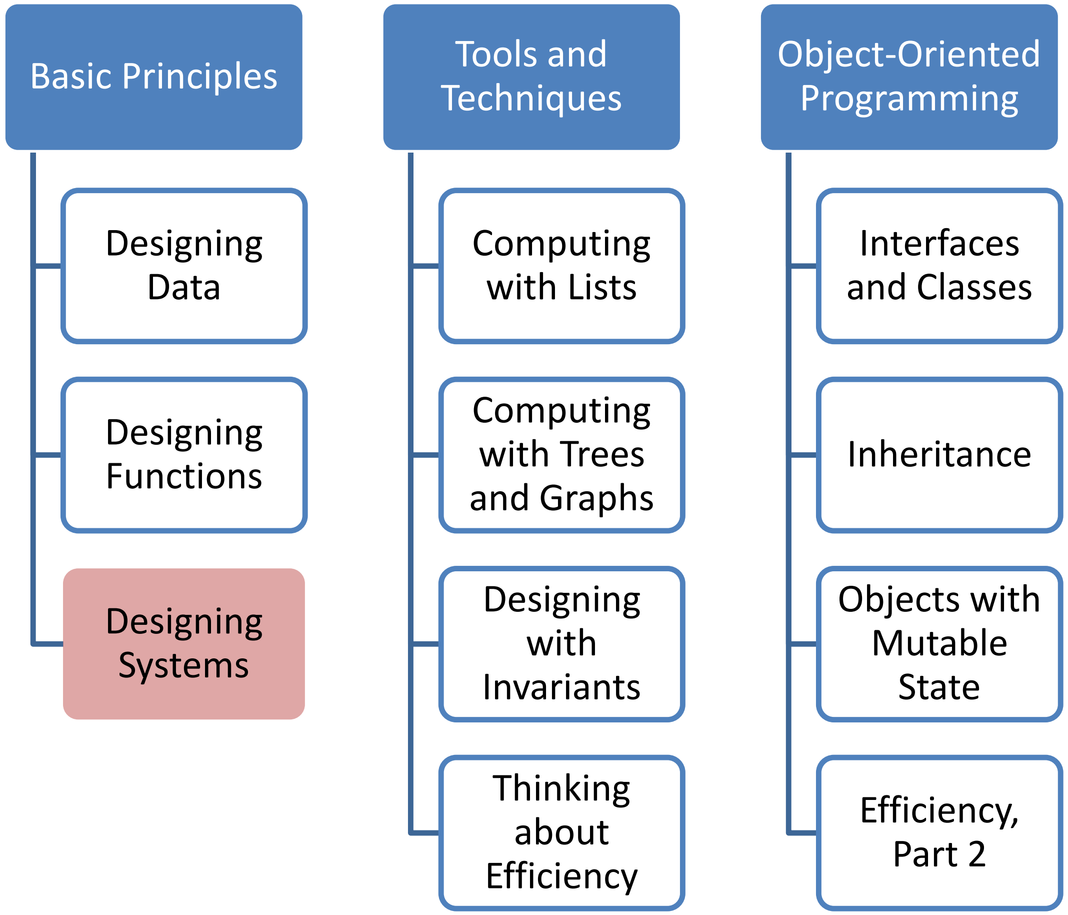
How to Design Systems

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 3.1

Module 03



Basic Principles
Designing Data
Designing Functions
Designing Systems
Tools and Techniques
Computing with Lists
Computing with Trees and Graphs
Designing with Invariants
Thinking about Efficiency
Object-Oriented Programming
Interfaces and Classes
Inheritance
Objects with Mutable State
Efficiency, Part 2

Module Introduction

- In this module, we will learn about the System Design Recipe, which gives you a recipe for building programs with multiple functions
- We will learn how to do “iterative refinement”– that is, adding features to a program.
- We will illustrate the recipe by building 3 versions of a simple animation, using the Racket **2htdp/universe** module.

Learning Objectives for this lesson

- At the end of this lesson, students should be able to:
  - Explain the steps of the System Design Recipe
  - Use the 2htdp/universe module to create a simple interactive animation, including:
    - Analyzing data to determine whether it should be constant or part of the world state,
    - Writing data definitions for worlds, key events, and mouse events, and
    - Writing code to handle the various events during the animation.

The System Design Recipe

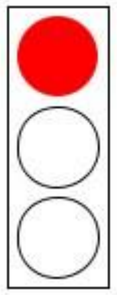
- The Function Design Recipe gave you a recipe (a “workflow”) to help you organize the design of a single function.
- The System Design Recipe gives you a workflow to help you get and stay organized when you need to build a system consisting of many functions.
- Here it is:

The System Design Recipe

The System Design Recipe
1. Write a purpose statement for your system.
2. Design data to represent the relevant information in the system.
3. Make a wishlist of main functions. Write down their contracts and purpose statements.
4. Design the individual functions. Maintain a wishlist of functions you will need to write.

Purpose Statement for a Traffic Light simulation

- The light is a machine that responds to time passing; as time passes, the light goes through its cycle of colors.
- The state of the machine consists of its current color and the amount of time (in ticks) until the next change of color. At every tick, the amount of time decreases by 1.
- When the timer reaches 0, the light goes to its next color (from green to yellow, from yellow to red, from red to green), and the timer is reset to the number of ticks that light should stay in its new color.
- In addition, the traffic light should be able to display itself as scene, perhaps something like this



A second example: The Falling Cat Purpose Statement

- We will produce an animation of a falling cat.
- The cat will starts at the top of the canvas, and fall at a constant velocity.
- If the cat is falling, hitting the space bar should pause the cat.
- If the cat is paused, hitting the space bar should unpause the cat.

falling-cat.rkt demo

YouTube link

The 2htdp/universe module

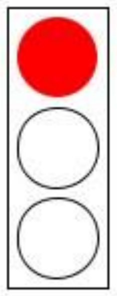
- We can build interactive systems like this using the 2htdp/universe module.
- This module expects the system to be expressed as a *machine*.
- The machine will have some *state.*
- The machine can respond to *inputs.*
- The machine's response to input is described as a *function*.
- The machine can show its state as a *scene.*

Step 2: Information Analysis

- What are the states of the machine?
- What are the inputs?
- What is the machine’s response to each input?
  - What state should the machine go to after each input?
- How would we like to display the state of the machine?

Information Analysis for the Traffic Light

- The state of the traffic light is compound information:
  - its current color AND # of ticks until next change
- Inputs will be the time (ticks). At every tick, the timer is decremented.
- When the timer reaches 0, the light goes to its next color (from green to yellow, from yellow to red, from red to green), and the timer is reset to the number of ticks that light should stay in its new color.
- The traffic light can show its state as a scene, perhaps something like this:



Except for the display, this is what we did in **traffic-light-with-timer2**.rkt

Information Analysis for the Falling Cat

- There are the only two things that change as the animation progresses: the position of the cat, and whether or not the cat is paused. So that's what we put in the state:
- The state of the machine will consist of:
  - an integer describing the y-position of the cat.
  - a Boolean describing whether or not the cat is paused

Let’s work out the Falling Cat example in detail…

Step 2 for the Falling Cat: Data Design

```
;; REPRESENTATION:
;; A World is represented as (make-world pos paused?)
;; with the following fields:
;; pos     : Integer    the y-position of the center of the cat in the
;;                      scene
;; paused? : Boolean    tells whether or not the cat is paused

;; IMPLEMENTATION
(define-struct world (pos paused?))

;; CONSTRUCTOR TEMPLATE
;; (make-world Integer Boolean)

;; OBSERVER TEMPLATE
;; world-fn : World -> ??
(define (world-fn w)
  (... (world-pos w)
       (world-paused? w)))
```

Falling Cat 1: Information Analysis, part 2

- What inputs does the cat respond to?
- Answer: it responds to *time passing* and to *key strokes*

What kind of Key Events does it respond to?

- It responds to the space character, which is represented by the string " " that consists of a single space.
- All other key events are ignored.

Next, make a wishlist

- What functions will we need for our application?
- Write contracts and purpose statements for these functions.
- Then design each function in turn.

Wishlist (1): How does it respond to time passing?

We express the answer as a function:

```
;; world-after-tick: World -> World
;; RETURNS: the world that should
;; follow the given world after a
;; tick.
```

Wishlist (2): How does it respond to key events?

```
;; world-after-key-event :
;;   World KeyEvent -> World
;; RETURNS: the world that should follow the given world
;; after the given key event.
;; on space, toggle paused?-- ignore all others
```

Here we've written the purpose statement in two parts. The first is the general specification ("produces the world that should follow the given world after the given key event"), and the second is a more specific statement of what that world is.

Wishlist (3)

- We also need to *render* the state as a scene:

```
;; world-to-scene : World -> Scene
;; RETURNS: a Scene that portrays the given
;; world.
```

Another response described as a function!

Wishlist (4): Running the world

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
```

Here the function has an effect in the real world (like reading or printing). We document this by writing an EFFECT clause in our purpose statement.

For now, functions like main will be our only functions with real-world effects. All our other functions will be pure: that is, they compute a value that is a mathematical function of their arguments. They will not have side-effects.

Side-effects make it much more difficult to understand what a function does. We will cover these much later in the course.

Next: develop each of the functions

```
;; world-after-tick : World -> World
;; RETURNS: the world that should follow the given
;; world after a tick

;; EXAMPLES:
;; cat falling:
;; (world-after-tick unpaused-world-at-20)
;;  = unpaused-world-at-28
;; cat paused:
;; (world-after-tick paused-world-at-20)
;;  = paused-world-at-20
```

We add some examples. We've included some commentary and used symbolic names so the reader can see what the examples illustrate.

Choose strategy to match the data

- **`World`** is compound, so use the template for World

```
;; strategy: use template for World on w
(define (world-after-tick w)
  (... (world-pos w) (world-paused? w)))
```

- What goes in `...` ?
- If the world is paused, we should return **w** unchanged. Otherwise, we should return a world in which the cat has fallen **CATSPEED** pixels.

Function Definition

```
;; STRATEGY: Use template for World on w

(define (world-after-tick w)
  (if (world-paused? w)
      w
      (make-world
       (+ (world-pos w) CATSPEED)
          (world-paused? w))))
```

Here we've just written out the function because it was so simple. If it were any more complicated, we might break it up into pieces, as on the next slide.

A more complete description of our strategy might be

STRATEGY: Use template for World on w, then cases on whether w is paused.

You don't have to be so detailed.

Alternative Function Definition

```
;; STRATEGY: Cases on whether the world is paused.

(define (world-after-tick w)
  (if (world-paused? w)
      (paused-world-after-tick w)
      (unpaused-world-after-tick w)))
```

Here we've broken the definition into pieces. If either of the pieces is complicated, this would be better code than what we saw on the preceding slide.

Tests

```
(define unpaused-world-at-20 (make-world 20 false))
(define paused-world-at-20   (make-world 20 true))
(define unpaused-world-at-28 (make-world (+ 20 CATSPEED) false))
(define paused-world-at-28   (make-world (+ 20 CATSPEED) true))

(begin-for-tests
  (check-equal?
    (world-after-tick unpaused-world-at-20)
    unpaused-world-at-28
    "in unpaused world, the cat should fall CATSPEED pixels and world should
   still be unpaused")

  (check-equal?
    (world-after-tick paused-world-at-20)
    paused-world-at-20
    "in paused world, cat should be unmoved"))
```

How does it respond to key events?

```
;; world-after-key-event :
;;    World KeyEvent -> World
;; GIVEN: a world w
;; RETURNS: the world that should follow the given world
;; after the given key event.
;; on space, toggle paused?-- ignore all others
;; EXAMPLES: see tests below
;; STRATEGY: cases on kev : KeyEvent

(define (world-after-key-event w kev)
  (cond
    [(key=? kev " ")
     (world-with-paused-toggled w)]
    [else w]))
```

We make a decision based on **kev**, and pass the data on to a help function to do the real work.

Requirements for Helper Function

```
;; world-with-paused-toggled : World -> World
;; RETURNS: a world just like the given one, but with
;;          paused? toggled
```

If this helper function does what it's supposed to, then **world-after-key-event** will do what it is supposed to do.

Tests for **world-after-key-event** (1)

```
;; for world-after-key-event, we have 4
;; equivalence classes: all combinations of:
;; a paused world and an unpaused world,
;; and a "pause" key event and a "non-pause" key
;; event

;; Give symbolic names to "typical" values:
;; we have these for worlds,
;; now we'll add them for key events:
(define pause-key-event " ")
(define non-pause-key-event "q")
```

Tests (2)

```
(check-equal?
  (world-after-key-event
    paused-world-at-20 pause-key-event)
  unpaused-world-at-20
  "after pause key, paused world should become unpaused")

(check-equal?
  (world-after-key-event
    unpaused-world-at-20 pause-key-event)
  paused-world-at-20
  "after pause key, unpaused world should still be paused")
```

Tests (3)

```
(check-equal?
   (world-after-key-event
     paused-world-at-20 non-pause-key-event)
   paused-world-at-20
   "after a non-pause key, paused world should be
    unchanged")

 (check-equal?
   (world-after-key-event
     unpaused-world-at-20 non-pause-key-event)
   unpaused-world-at-20
   "after a non-pause key, unpaused world should be
    unchanged")
```

Tests (4)

```
(define (world-after-key-event w kev) ...)

(begin-for-test
  (check-equal? ...)
  (check-equal? ...)
  (check-equal? ...)
  (check-equal? ...))

(define (world-with-paused-toggled? w) ...)
```

Here's how we lay out the tests in our file.

Contract, purpose function, etc., for **world-with-paused-toggled?**

Now we're ready to design our help function

Definition for Helper Function

```
;; world-with-paused-toggled : World -> World
;; RETURNS: a world just like the given one, but with
;;  paused? toggled
;; STRATEGY: Use template for World on w
(define (world-with-paused-toggled w)
  (make-world
   (world-pos w)
   (not (world-paused? w))))
```

Don't need to test this separately, since tests for **world-after-key-event** already test it.

What else is on our wishlist?

```
;; world-to-scene : World -> Scene
;; RETURNS: a Scene that portrays the given world.
;; EXAMPLE:
;; (world-to-scene (make-world 20 ??))
;;  = (place-image CAT-IMAGE CAT-X-COORD 20 EMPTY-CANVAS)
;; STRATEGY: Place the image of the cat on an empty canvas
;;           at the right position.

(define (world-to-scene w)
  (place-image CAT-IMAGE CAT-X-COORD
               (world-pos w)
               EMPTY-CANVAS))
```

We could have written "use template for World on w", but this is clear and much more informative.

place-image was covered in Lesson 0.4. You can find a refresher in the Resources section of this module.

Testing world-to-scene

```
;; an image showing the cat at Y = 20
;; check this visually to make sure it's what you want
(define image-at-20 (place-image CAT-IMAGE CAT-X-COORD 20 EMPTY-CANVAS))

;; these tests are only helpful if image-at-20 is the right image.

(begin-for-test
  (check-equal?
    (world->scene unpaused-world-at-20)
    image-at-20
    "(world-to-scene unpaused-world-at-20) should display as image-at-20")

  (check-equal?
    (world->scene paused-world-at-20)
    image-at-20
    "(world-to-scene paused-world-at-20) should display as image-at-20"))
```

Last wishlist item

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
```

The real purpose of main is not to return a useful value; instead, its purpose is have some visible effect in the real world– in this case, to display some things on the real screen and take input from the real user. We document this in the purpose statement by writing an **EFFECT** clause.

Template for **big-bang**

```
;; big-bang
;; EFFECT : runs a world with the specified event handlers.
;; RETURNS: the final state of the world
(big-bang
  initial-world
  (on-tick tick-handler rate)
  (on-key  key-handler)
  (on-draw render-fcn)))
```

frame rate in secs/tick

names of events

functions for responses

There are other events that big-bang recognizes. See the Help Desk for details

Putting the pieces together

```
;; main : Integer -> World
;; GIVEN: the initial y-position in the cat
;; EFFECT: runs the simulation, starting with the cat
;;  falling
;; RETURNS: the final state of the world
;; STRATEGY: Combine simpler functions
(define (main initial-pos)
  (big-bang (make-world initial-pos false)
            (on-tick world-after-tick 0.5)
            (on-key world-after-key-event)
            (on-draw world-to-scene)))
```

Here the simpler functions are **big-bang**, **world-after-tick**, **world-after-key-event**, and **world-to-scene**

Let's walk through falling-cat.rkt

- Note: this video differs from our current technology in a couple of ways:
  - it talks about test suites; these are replaced by **begin-for-test**.
  - it talks about "partition data" and gives a template for FallingCatKeyEvents. We've simplified the presentation-- now we just have KeyEvents, which are scalars (no template needed), and we take them apart using the "Cases" strategy.
  - And remember, the “Structural Decomposition” strategy is now called “Use template”.

falling-cat.rkt readthrough

YouTube link

The System Design Recipe

- Let's review the System Design Recipe
- Go back and look at our development of **03-1-falling-cat.rkt** to see how it matched the recipe.

One more time...

The System Design Recipe
1. Write a purpose statement for your system.
2. Design data to represent the relevant information in the system.
3. Make a wishlist of main functions. Write down their contracts and purpose statements.
4. Design the individual functions. Maintain a wishlist of functions you will need to write.

Summary

- We built a system using the *system design recipe*.
- We used the universe module, which provides a way of creating and running an interactive machine.
  - Machine will have some *state.*
  - Machine can respond to *inputs*.
  - Response to input is described as a *function*.
  - Machine can show its state as a *scene.*

Next Steps

- Study the file **03-1-falling-cat.rkt** in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 3.1
- Go on to the next lesson