The Data Design Recipe

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 1.3

Learning Objectives for this Lesson

- By the time you finish this lesson, you should be able to:
  - list the steps in the data design recipe
  - list the pieces of a data definition
  - explain what define-struct does
  - write a constructor template and interpretation for simple data

The Data Design Recipe

1. Information Analysis
2. Representation and Interpretation
3. Implementation
4. Constructor Template
5. Observer Template
6. Examples
7. Review

Brief Explanation of the Data Design Recipe

1. **Information Analysis**: What kind of information needs to be represented in your program? What kind of information is it?
2. **Representation and Interpretation**: how is the information represented as data? What is the meaning of each possible value of the data?
3. **Implementation**: definitions of needed structs.
4. **Constructor template**: tells how to construct a value of this type
5. **Observer template**: tells how to inspect a value of this type
6. **Examples**: samples to make clear to the reader what is intended
7. **Review**: How can your design be improved?

DDR Step 1. What information needs to be represented?

- In general, you are representing information about one of many objects in the world
- Your goal is represent enough information about that object to distinguish it from all the other similar objects
- You may need to represent more information as well, depending on the application.

Example: representing a car

- What cars are we trying to represent? How does one car in the set differ from the others? What information about that car do I need to know?
- In a traffic simulation, I might only need to keep track of each car's position and velocity.
- For TV coverage of an auto race, I might need to keep track of enough information to distinguish it from all the others in the race.
  - The car's number would be enough, but I might want to display the name of the driver as well.
- For an auto dealer, I might need to keep track of enough information to distinguish this car from all the others in the world.
  - The VIN ("Vehicle Identification Number") would be enough, but I might want to have its model, color, etc. available for display.

Output of DDR Step 1

- At the end of step 1, you should know what kind of data you need (scalar, compound, itemization, etc.)
- Where you go from here depends on the kind of data.
- In the lesson, we'll see how to execute the Data Design Recipe for compound data.
- Then we'll go back and see how it works for the other kinds of data.

DDR Step 2. Representation and Interpetation

- In Racket, we represent compound data as a **`struct`**
  - This is like a struct or record in other languages.
- For the interpretation, we need to give an interpretation to each of the fields.

Example:

```
;; A Book in a bookstore

;; REPRESENTATION:
;; a Book is represented as a struct
;;    (make-book author title on-hand price)
;; with the following fields:
;; author : String     is the author's name
;; title  : String     is the title of the book
;; on-hand : NonNegInt is the number of copies
;;                     on hand
;; price   : NonNegInt is the price of the book
;;                     in USD*100
;;                     (e.g. $7.95 => 795)
```

Another example: a Rocket

```
;; An Altitude is represented as a Real, measured in meters

;; A Velocity is represented as a Real, measured in meters/sec
;;  upward

;; We have a single rocket, which is at some altitude and is
;; travelling vertically at some velocity.

;; REPRESENTATION:
;; A Rocket is represented as a struct
;;   (make-rocket altitude velocity)
;; with the following fields:
;; altitude : Altitude  is the rocket's altitude
;; velocity : Velocity  is the rocket's velocity
```

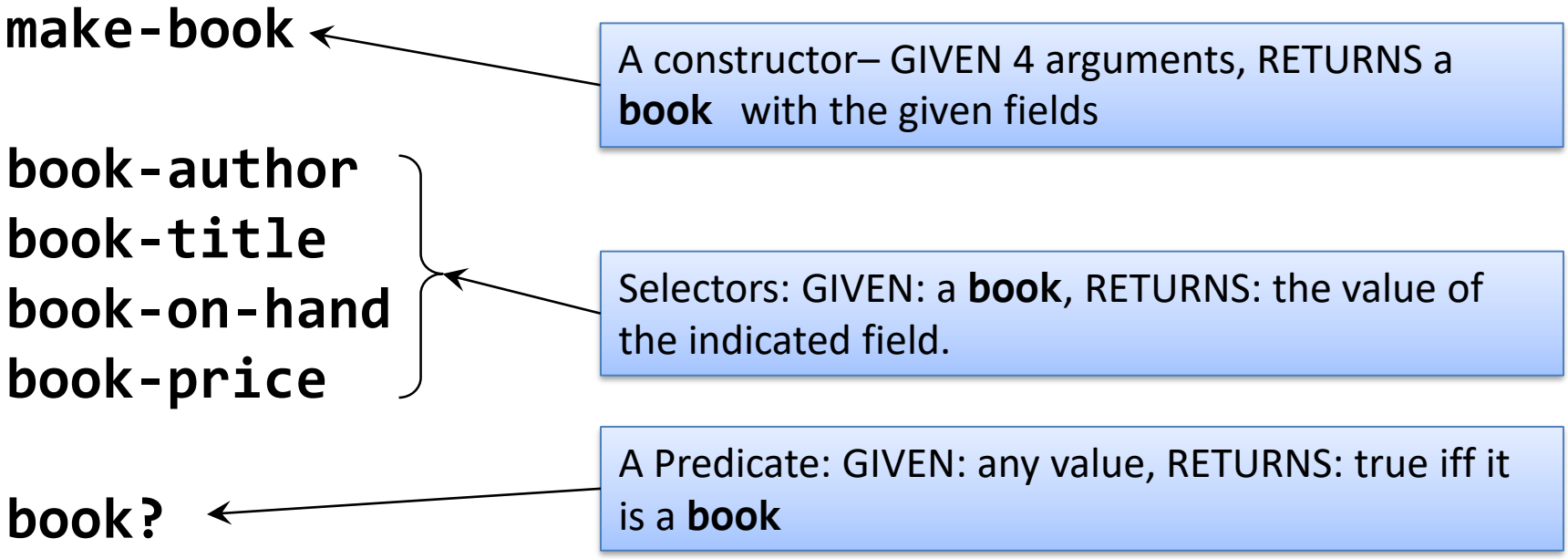
DDR Step 3. Implementation

- In Racket, we define new kinds of structs or records with **`define-struct`**.
- We saw these in Lesson 0.4. Here's a review:

Example of a structure definition in Racket

```
(define-struct book (author title on-hand price))
```

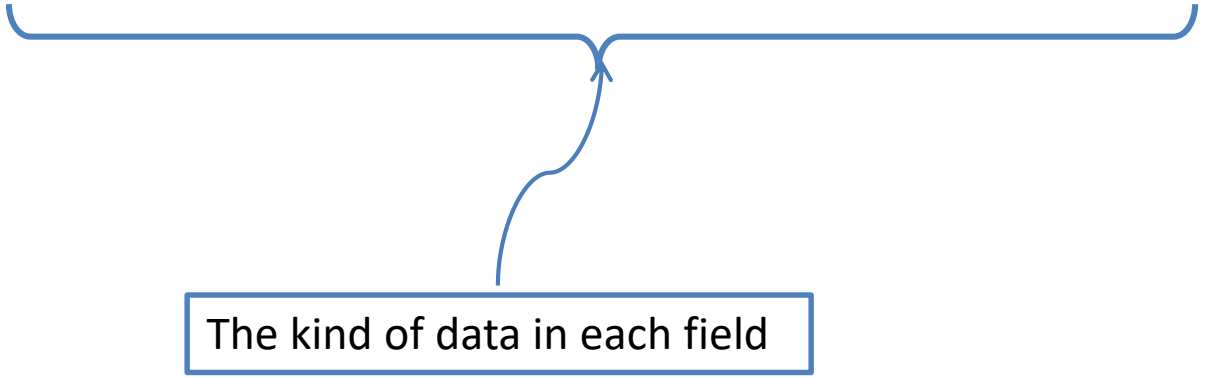
Executing this **`define-struct`** defines the following functions:



DDR Step 4. Constructor Template

- Tells how to construct a value of this type.
- Example:

```
;; CONSTRUCTOR TEMPLATE FOR Book:
;;  (make-book String String NonNegInt NonNegInt)
```

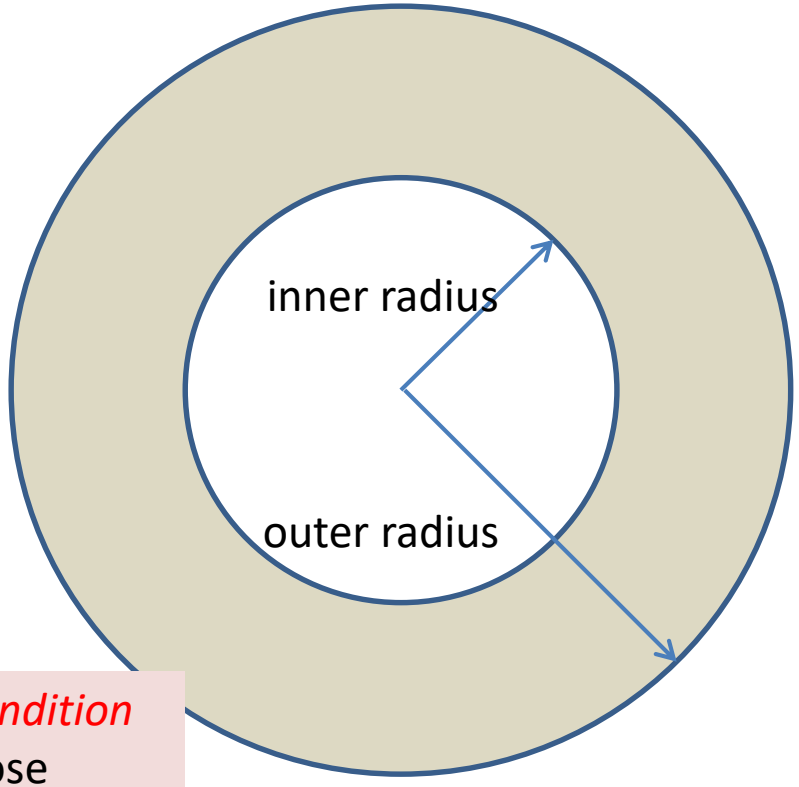


Sometimes this format isn't enough

- Example: a ring

This only makes sense if inner < outer .

We need to document this, so anybody who builds a ring will know that he or she has to satisfy this condition



This condition is called a *precondition* or an *invariant*. Remember those words! Conditions like this will come up over and over again as we go along.

We document this in the representation

```
;; REPRESENTATION
;; A Ring is represented as a struct
;;  (make-ring inner outer)
;; with the following fields:
;; inner : PosReal  is the ring's inner radius
;; outer : PosReal  is the ring's outer radius
;; WHERE (< inner outer) is true

;; IMPLEMENTATION
(define-struct ring (inner outer))

;; CONSTRUCTOR TEMPLATE:
;; (make-ring PosReal PosReal)
```

DDR Step 5: Observer Template

- The observer template (or just the template, for short) gives a skeleton for functions that examine or use the data.
- For compound data, this is easy:

Observer template for compound data

```
;; OBSERVER TEMPLATE
;; book-fn : Book -> ??
(define (book-fn b)
  (...
    (book-author b)
    (book-title b)
    (book-on-hand b)
    (book-price b)))
```

This is an *inventory* of the quantities we can use to construct the answer.

We can fill in the ... with any expression, using any or all of the expressions in the inventory.

Putting it all together

```
;; A Book in a bookstore

;; REPRESENTATION:
;; a Book is represented as a struct (make-book author title on-hand price)
;; with the following fields:
;; author : String     is the author's name
;; title  : String     is the title of the book
;; on-hand : NonNegInt is the number of copies on hand
;; price   : NonNegInt is the price of the book in USD*100
;;                          (e.g. $7.95 => 795)

;; IMPLEMENTATION
(define-struct book (author title on-hand price))

;; CONSTRUCTOR TEMPLATE
;; (make-book String String NonNegInt NonNegInt)

;; OBSERVER TEMPLATE
;; book-fn : Book -> ??
(define (book-fn b)
  (...
    (book-author b)
    (book-title b)
    (book-on-hand b)
    (book-price b)))
```

The DDR for scalar data

- For scalar information, we need representation and interpretation
- No need for implementation or constructor/observer templates

Examples of data definitions for scalar information

```
;; An Altitude is represented as a Real,
;;    measured in meters

;; A Velocity is represented as a Real, measured in
;;   meters/sec upward

;; A BookPrice is represented as a NonNegInt,
;;   in USD*100 (e.g. $7.95 => 795)

;; A Vineyard is represented as a String
;;      (any string will do)
;; A Vintage  is represented as a PosInt in [1800,2100]
```

interpretation specifies units

interpretation specifies units and orientation

interpretation specifies units

When you say “String”, that always means that any string is a legal value!! Any time your data definition says “String”, you should always write “Any string will do”.

The DDR for itemization data

```
;; A Size is represented as one of the following integers:
;; -- 8, 12, 16, 20, 30
;; INTERP: a cup size, in fluid ounces

;; NOTE: it would be wrong to say "the cup", since there is no cup
;; here.  Look at the definition of CoffeeOrder below

;; NOTE: Constructor template is not necessary for itemization data

;; OBSERVER TEMPLATE:

;; size-fn : Size -> ?
(define (size-fn s)
  (cond
    [(= s 8)  ...]
    [(= s 12) ...]
    [(= s 16) ...]
    [(= s 20) ...]
    [(= s 30) ...]))
```

The observer template consists of a cond with as many clauses as there are cases. The predicates in the cond select the relevant case.

Another example

- Do we need an interpretation?

```
A TLState is represented as
one of the following strings:
-- "red"
-- "yellow"
-- "green"
```

- NO: common sense is good enough

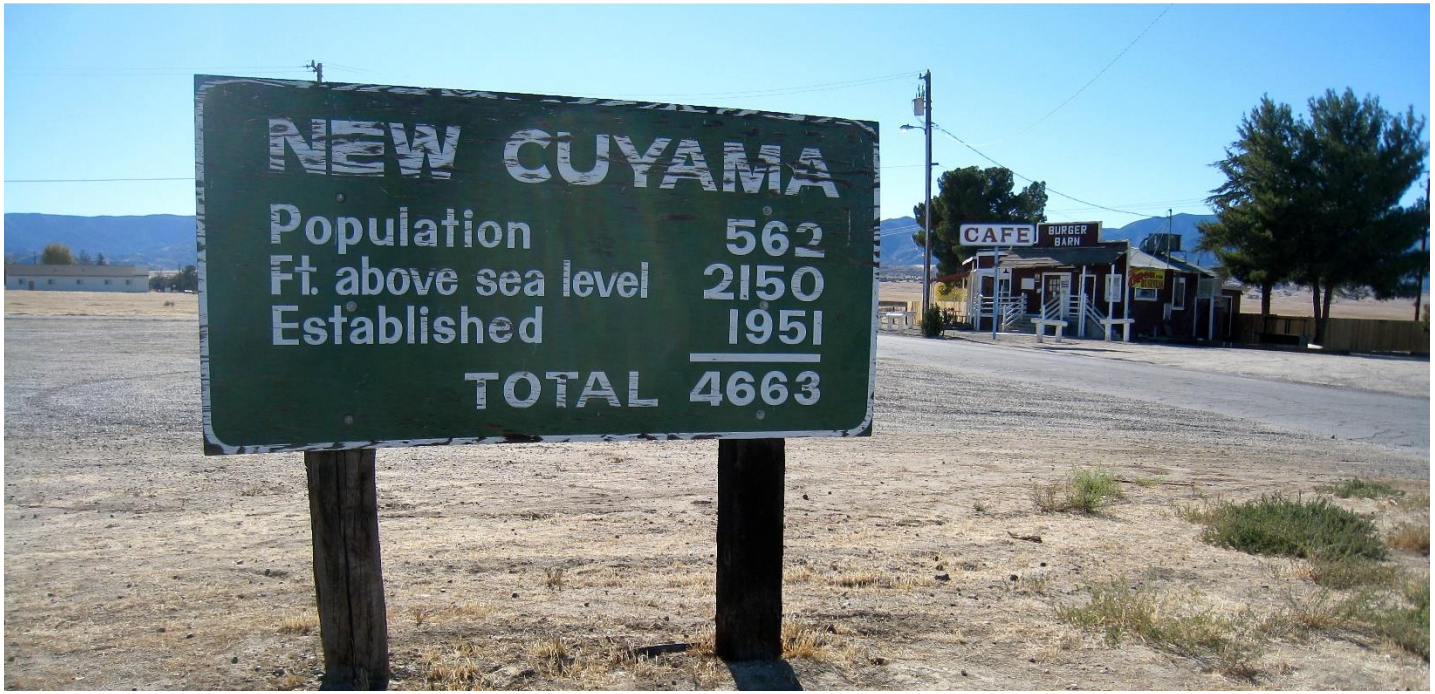
Another example

- Do we need an interpretation?

```
A TLState is represented one
of the following integers:
-- 217
-- 126
-- 43
```

- YES: the reader is unlikely to guess that 217 denotes green, 126 denotes yellow, and 43 denotes red.

Remember: Not all integers are created equal



The interpretation tells you the meaning of each number. It also tells you that you shouldn't be adding these integers!

The DDR for more complicated data

- For example:
  - compound data where one or more fields are themselves compound or itemization data
  - itemization data where one of more fields are themselves compound or itemization data
- We build the data definition for this more complicated data out of the definitions of the pieces.
- This is less scary than it sounds.
- Let’s do some examples

Case Study: BarOrder

In a wine bar, an order may be one of three things: a cup of coffee, a glass of wine, or a cup of tea.

- For the coffee, we need to specify the size (small, medium, or large) and type (this is a fancy bar, so it carries many types of coffee). Also whether or not it should be served with milk.
- For the wine, we need to specify which vineyard and which year.
- For tea, we need the size of the cup and the type of tea (this is a fancy bar, so it carries many types of tea).

Case Study (part 2): CoffeeType

```
;; Preliminaries:

;; A Size is represented as ...

;; A CoffeeType is represented as a string
;;    (any string will do)
```

Any time you write **String**, you MUST write "any string will do".

In a more detailed representation, we might specify **CoffeeType** further.

Case Study (part 3): CoffeeOrder

```
;; Definition of CoffeeOrder:

;; REPRESENTATION:
;; A CoffeeOrder is represented as a struct
;;  (make-coffee-order size type milk)
;; with the following fields:
;; INTERP:
;;   size : Size           is the size of cup desired
;;   type : CoffeeType     is the kind of coffee order
;;   milk : MilkType       is the kind of milk ordered

;; IMPLEMENTATION:
(define-struct coffee-order (size type milk))

;; CONSTRUCTOR TEMPLATE
;; (make-coffee-order Size CoffeeType MilkType)

;; OBSERVER TEMPLATE
;; coffee-order-fn : CoffeeOrder -> ??
(define (coffee-order-fn co)
  (...
   (coffee-order-size co)
   (coffee-order-type co)
   (coffee-order-milk co)))
```

Case Study, completed: BarOrder

```
;; A WineOrder is represented as a ...
;; A TeaOrder is represented as a ...

;; A BarOrder is represented as one of
;; -- a CoffeeOrder
;; -- a WineOrder
;; -- a TeaOrder

;; CONSTRUCTOR TEMPLATE:
;; use the constructor templates for CoffeeOrder, WineOrder, or TeaOrder.

;; OBSERVER TEMPLATE:
;; bo-fn : BarOrder -> ??
;; STRATEGY: Cases on order : BarOrder
(define (bo-fn order)
  (cond
    [(coffee-order? order) ...]
    [(wine-order?   order) ...]
    [(tea-order?    order) ...]))
```

Using the BarOrder Observer Template

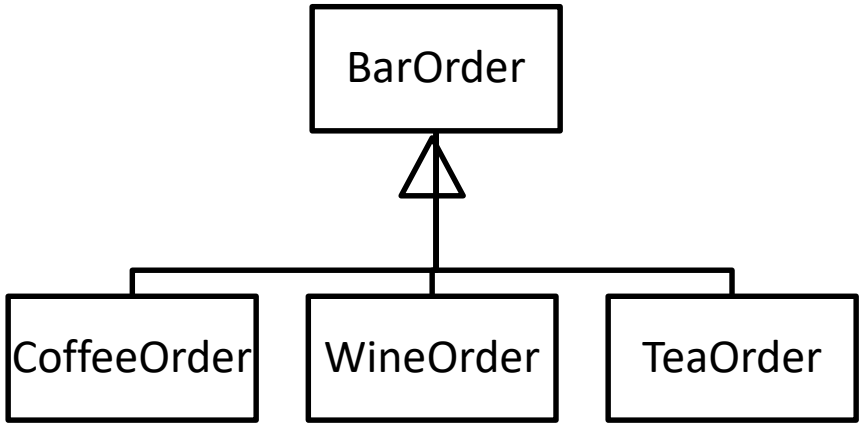
```
;; In the ... you can put a function on the order, or you
;; can expand the observer template for the compound
;; data, eg:

(define (my-bo-fn order)
  (cond
    [(coffee-order? order) (some-function
                            (coffee-order-size order)
                            (coffee-order-type order)
                            (coffee-order-milk order))]
    [(wine-order?   order) (some-other-function order)]
    [(tea-order?    order) (yet-another-function order)]))
```

Best practice is to expand all of them, as we've done here for the coffee-order alternative.

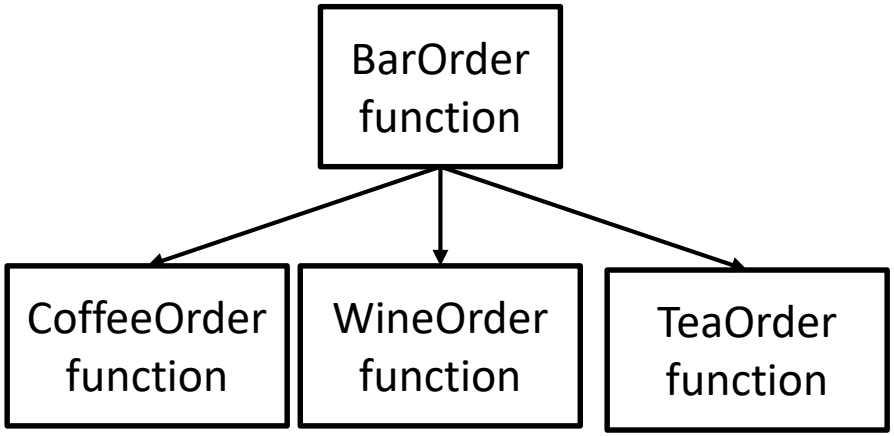
The Shape of the Program Follows the Shape of the Data

- A BarOrder is one of
  - a CoffeeOrder
  - a WineOrder, or
  - a TeaOrder
- The observer template tells that a function on BarOrders may call
  - a function on CoffeeOrders
  - a function on WineOrders, or
  - a function on TeaOrders

The Shape of the Program Follows the Shape of the Data



Data Hierarchy (the open triangle means "OR")



Call Tree (the arrow goes from caller to callee)

We'll see this principle over and over again!

DDR Step 7: Review

- Nothing is done until you review it!
- Before you move on, look at your data definition and ask the following questions

Reviewing a Data Design

Reviewing a Data Design
1. Is the interpretation clear and unambiguous?
2. Can you represent all the information you need for your program?
3. Do you *need* all of the data in your representation?
4. Does every combination of values make sense? If not, document the meaningful combinations with a WHERE clause.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson