SPD Checklists

See full recipe page for details

```
(require spd/tags)
(require 2htdp/image)
(require 2htdp/universe)

;; My world program  (make this more specific)
(@htdw WS)
;; =================
;; Constants:


;; =================
;; Data definitions:

(@htdd WS)
;; WS is ... (give WS a better name)



;; =================
;; Functions:

(@htdf main)
(@signature WS -> WS)
;; start the world with (main ...)
;; no tests for main function

(@template-origin htdw-main)

(define (main ws)
  (big-bang ws          ;WS
    (on-tick   tock)    ;WS -> WS
    (to-draw   render) ;WS -> Image
    (on-mouse  ...)     ;WS Integer Integer MouseEvent -> WS
    (on-key    ...)))  ;WS KeyEvent -> WS

(@htdf tock)
(@signature WS -> WS)
;; produce the next ...
;; !!!
(define (tock ws) ws)

(@htdf render)
(@signature WS -> Image)
;; render ...
;; !!!
(define (render ws) empty-image)
```

HtDW

1. Domain analysis (use a piece of paper!)
    1. Sketch program scenarios
    2. Identify constant information
    3. Identify changing information
    4. Identify big-bang options
2. Build the actual program
    1. Constants (based on 1.2 above)
    2. Data definitions (based on 1.3 above)
    3. Functions
        1. main first (based on 1.4 and 2.2 above)
        2. wish list entries for big-bang handlers
    4. Work through wish list until done

```
on-tick
to-draw
on-key
on-mouse
```

HtDD

First identify form of information, then write:

1. A possible structure definition (not until compound data)
2. A type comment that defines type name and describes how to form data
3. An interpretation to describe correspondence between information and data.
4. One or more examples of the data.
5. A template for a 1 argument function operating on data of this type.

HtDF

1. Signature, purpose and stub.
2. Define examples, wrap each in check-expect.
3. Template and inventory.
4. Code the function body.
5. Test and debug until correct

Test guidelines

1. at least 2
2. different argument/field values
3. code coverage
4. points of variation in behavior
5. 2 long / 2 deep

Choosing form of data definition

When the form of the information to be represented...	Use a data definition of this kind
is atomic	simple atomic data (String, Number...)
is numbers within a certain range	number type and CONSTRAINT
consists of a fixed number of distinct items	enumeration (one-of several strings)
is comprised of 2 or more subclasses, at least one of which is not a distinct item	itemization (one-of several subclasses)
consists of items that naturally belong together	compound data
is arbitrary sized	well formed self-referential data definition (or mutually referential)
is naturally composed of different parts	reference to another defined type

Metadata tags

```
@assignment
@cwl

@problem

@htdw
@htdd
@htdf

@signature
@dd-template-rules
@template-origin
@template
```

Data Driven Template Rules

Form of data	cond question (if any)	Body or cond answer

atomic non-distinct	`type predicate`
	`(string? x) (number? x)`
	`etc.`
		`(... x)`

atomic distinct	`equality predicate`
	`(string=? x "red")`
	`etc. possible w/ guard`
		`(...)`

one of		cond w/ one Q&A pair per subclass be sure to guard in mixed data itemizations

compound	predicate
	`(firework? x)`
		all selectors
		`(... (balloon-x b)`
		`(balloon-y b))`

self-reference		form natural recursion `(fn-for-los (rest los))`
reference		call to other type's templates function `(fn-for-drop (first lod))`

for additional parameters with atomic type add parameter everywhere after ...