

The Design Recipe using Classes

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 9.5



© Mitchell Wand, 2012-2017

This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

Note to the reader

- This PPT presentation is based on an earlier presentation that targeted the Racket object system.
- I've gone through Professor Clinger's HTML pages and imported the material back to PowerPoint, which some readers may find easier to use.
- There are a few slides with examples that were too difficult to turn from Racket into Java. Those are marked with bright yellow badges, and the filename for the corresponding page in the HTML (which you can wget or pull from github).
- Sorry about that...

Goals of this lesson

- See how the design recipe and its deliverables should appear in an object-oriented system
- Note: this is about OUR coding standards. Your workplace may have different standards.

Let's review the Design Recipe

The Function Design Recipe

1. Data Design
2. Contract and Purpose Statement
3. Examples and Tests
4. Design Strategy
5. Function Definition
6. Program Review

In an OO system, the steps are a little different, but they are all there

The Object-Oriented Design Recipe

Step	Description
1. Interface Design	Identify the kinds of things in your system and the messages they need to respond to. For each method in an interface, write a contract and purpose statement.
2. Class Design	Identify the kinds of things that may be behind each interface. For each class, give a purpose statement. For each field of a class, give an interpretation.
3. Method Design	For each method, copy down the contract and purpose statement from the interface. Specialize the purpose statement to specify how the purpose is fulfilled for this class. Include examples as needed.
4. Unit Tests	For each class, write tests that exercise every method
5. Program Review	Same as before

Some differences

- "Design Strategy" has been dropped as a separate step, but you should still describe a design strategy if you think it would help readers to understand your code. Most of the time, your methods will be so simple you probably won't need to describe a design strategy.
- "Halting Measure" has been dropped as a deliverable, but you should still describe a halting measure for methods that might be involved in a recursion, unless the halting measure is obvious.
- "Contract and Purpose Statement" is now part of the "Interface Design" and "Class Design" steps.
- Unit tests are shown as the fourth step, after method definitions, but it is still a good idea to write at least some tests for each method after you write the contracts for the method and before you actually define the method.

Step 1: Interface Design

- What kinds of things will exist in your system?
- What messages will they need to respond to?
- List the messages (methods) in each interface
- Write a purpose statement for the interface
- For each method in the interface, write a contract and purpose statement.
- Write the contracts in terms of data types and interfaces (never classes).
- This is similar to the wishlist in the functional model.

Example 1: StupidRobot

```
// A StupidRobot is an object of any class that implements StupidRobot
//
// Interpretation: A StupidRobot represents a robot moving along
// a horizontal line starting at position 0.

interface StupidRobot {

    // a new StupidRobot is required to start at position 0.

    // RETURNS: a robot just like this one, moved one position
    // to the right.

    StupidRobot moveRight ();

    // RETURNS: the current x-position of this robot

    int getPos ();
}
```


Example 2: Widget

```
// Every object that lives in the world must implement the
// Widget interface.
```

```
interface Widget {

    // RETURNS: the state of this object that should follow
    //         the next tick

    Widget afterTick ();

    // GIVEN: coordinates of a location
    // RETURNS: the state of this object that should follow
    //         the specified mouse event at the given location

    Widget afterButtonDown (int mx, int my);
    Widget afterButtonUp (int mx, int my);
    Widget afterDrag (int mx, int my);


    // GIVEN: a key event
    // RETURNS: the state of this object that should follow
    //         the given key event

    Widget afterKeyEvent (KeyEvent ke);

    // GIVEN: a scene
    // RETURNS: a scene like the given one, but with this
    //         object painted on it

    Scene addToScene (Scene sc);
}
```

Another way to write a
purpose statement for an
interface



Step 2: Class Design

- For each interface, consider the different kinds of objects that will implement this interface. Each kind becomes a class.
- For each class, include a purpose statement that says what information is represented by objects of that class.
- For each class, give a constructor template showing how to build an object of that class.
- Each field should have an interpretation, just as every field in a **struct** has an interpretation.

Example

```
// Constructor template for Bomb:
//      new Bomb(x, y)
// Interpretation:
//      x and y are the x and y coordinates for the center of this bomb

class Bomb implements Widget {

    int x;                // x coordinate for this bomb's center
    int y;                // y coordinate for this bomb's center

    // image for displaying the bomb
    // (declared static to avoid creating a separate image
    // for every bomb we create)

    static Image BOMB_IMG = Image.circle (10, "solid", "red");

    static int SPEED = 8; // the bomb's speed, in pixels/tick

    Bomb (int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

What happened to the Observer Template?

- An interface is implicitly itemization data
- Each class that implements the interface is like an alternative of the itemization data.
- The object system does all the **cond**'s for you.
- All that's left for you to do is to write the right-hand side of each **cond**-line.
 - When referring to the fields of **this** object, you can use fields instead of selectors.
 - So there's no need for a separate observer template.
(Yay!)

Coding Standards, Part 1

- A public method is a method whose definition begins with **public**.
 - If a method is listed in an interface, its definition must be public. (The Java compiler enforces this.)
 - If a method overrides a method inherited from **Object**, its definition must be public. (The Java compiler enforces this.)
 - A static method (such as **main**) may be public.
- Every public method of the class **MUST**
 - be listed in an interface the class implements, or
 - override a method inherited from **Object**, or
 - be a static method (such as **main**).
- Your class may also define non-public help methods. If a help method is called only from within the class, then it should be declared **private**.
- Your non-public methods should come after all of the public methods.
- You must not declare anything to be **protected**.

What happened to **protected**?

- In Java, declaring something to be protected makes it less protected, so programmers who are new to Java seldom use that keyword correctly. Java's protected keyword does have a few legitimate uses, but those uses are beyond the scope of this course.

Coding Standards, Part 2

- Interface and class names begin with a capital letter. Instead of using a hyphen to separate words in the name of a type or class, use Camel Case, as in **StupidRobot**.
- Variable and method names begin with a lower-case letter. Instead of using a hyphen to separate words in a variable or method name, use Camel Case: **addToScene** instead of **add-to-scene**.
- Constant names are entirely in upper case, using underscores instead of hyphens to separate words in a constant's name: **BOMB_IMAGE** instead of **BOMB-IMAGE**.
 - In Java, the definition of a constant begins with **static final**.

Coding Standards Illustrated

See [ooCodingStandards3.html](#)

```
;; A Foo is an object of any  
class that implements Foo<%>  
;; Module such-and-so expects to  
work with a list of Foo's.
```

```
(define Foo<%>  
  (interface ()
```

```
    ; -> Integer  
    ; purpose statement  
    omitted...  
    m1
```

```
    ; Bar ->  
    ; purpose statement  
    omitted...  
    add-bar)
```

Data Definitions
go with
Interfaces

No methods except those
listed in the interface

If you think you need a private
method, use a function instead.
Functions can refer to fields and
to **this**. These functions will not be
accessible outside the class

Exception: methods named **for-
test:...** need not be in the
interface, but they may only be
used for testing.

```
Class1%:  
;; (new Class1% [a Int][b Bool][c Foo])  
;; Interp: an object of Class1% represents a ....
```

```
(define Class1%  
  (class* object% (Foo<%>)
```

```
    (init-field a b c)  
    ;; interpretations omitted...
```

```
    (field [LOCAL-CONSTANT ...])  
    ;; interpretation omitted
```

```
    (super-new)
```

```
    ; m1 : -> Integer  
    ; purpose statement omitted...  
    (define/public (m1) ...)
```

```
    ; add-bar : Bar -> Foo  
    (define/public (add-bar b) ...)
```

```
    (define/public (method-not-in-interface ...) ...)
```

```
    (define (function1 ...) a b c this ...)  
    (define (function2 ...) a b c this ...)
```

```
    ;; for-test:... methods don't need to be  
    ;; in the interface
```

```
    (define/public (for-test:test-fcn1 ...) ...)
```

```
  ))
```

Classes have
Constructor
Templates and
Interpretations

Constants used only
in one class should
be fields.

Step 3: Method Design

- Each method definition should have a contract that is the same as the contract in the interface. (In Java, the compiler enforces this.)
- A method should have a purpose statement if that would be helpful to a reader. A public method's purpose statement may specialize the purpose statement given in an interface by adding details that explain how that purpose is achieved in this particular class.
- Methods should have examples as needed to clarify the purpose statement.
- Each method should have tests associated with it.
- A method should have a design strategy if that would help readers to understand its definition.
- A recursive method should have a halting measure if that would help readers to understand why it terminates.

Remember, a strategy is a tweet-sized description of how your function works

Contracts and Purpose Statements in a Class Definition

See [methodDesign2.html](#)

```
(define Bomb%  
  (class* object% (Widget<%>)  
    ...  
    ;; after-tick : -> Widget  
    ;; RETURNS: A bomb like this one, but as it should be after a tick  
    ;; DETAILS: the bomb moves vertically by BOMB-SPEED  
    (define/public (after-tick)  
      (new Bomb% [x x][y (+ y BOMB-SPEED)])))
```

Since **Bomb%** implements the **Widget<%>** interface, the value of **(after-tick)** is a **Widget**. So **after-tick** satisfies its contract.

Here's an example of a refined purpose statement

This one is so simple it doesn't need any examples.

Examples and Tests

- Examples and tests will generally be different.
- Put examples with the method.
- Phrase examples in terms of information (not data) whenever possible.
- Use meaningful names, etc., just as before.

Step 4: Unit Tests

- Your programming language, testing framework, and organizational standards will influence where you put your unit tests.
- In Java, you can put your unit tests for a class at the end of the class, following the non-public help methods, or you can put them in a separate class within the same file or in a separate file.
- Regardless of where you put unit your tests for a class, it is convenient to define a public static **main** method that runs all of the unit tests for that one class independently of the unit tests for other classes. That main method can then be called by the **main** method that runs all of the tests for your entire program.

Step 4: Unit Tests, part 2

- We still want 100% test coverage.
- Test observable behavior, as in the previous lesson.
- Don't assume the **equals** method can be used to compare objects.
 - The next module will discuss the **equals** method in more detail. We are talking about it here to help you avoid mistakes we often see in unit tests.
- In Java, the **equals** method might test all and only the observable behavior of objects it is comparing. If so, you can use it to compare objects. Sometimes, however, the **equals** method defines some notion of equality that does not correspond to identity of observable behavior.
- In Java, all objects have an equals method. and **x.equals(x)** is true, so a false value for **x.equals(y)** counts as an observable difference between **x** and **y**. On the other hand, a true value for **x.equals(y)** does not necessarily mean there are no observable differences between **x** and **y**. The behavior of the equals method becomes even harder to relate to observable behavior when the objects being compared are mutable: **x.equals(y)** may be true at one moment and false a moment later.

What happened to the strategy?

- We no longer require you to state a design strategy for every function and method.
- Early in the course, the design strategies you stated helped us to understand what you were trying to do even if your definition was completely wrong. As your programming skills have improved, that should happen less often now.
- You should still state a design strategy if you think it would help readers to understand your definition.

The real OO Design Strategies are the Patterns

- In OO world, the important design strategies are at the class level.
- Examples:
 - composite pattern (eg, composite shapes)
 - functional visitor pattern
 - MapReduce pattern
 - static factory method pattern
 - strategy pattern (eg, next week's **Fmap** example)

Remember:

- The design recipe is a process, not just a list of deliverables.

Properties of a good OO design

- One bundle of operations = one interface
 - If the interface consists of two kinds of things, working on disjoint pieces of data, consider splitting it
- One structure = one class
- Keep the interface as small as possible
- Keep the operations near the data
- Keep values local whenever possible
- All the other criteria of a good data design still hold
 - need good contracts, purpose statements, and invariants
 - If not every combination of values is meaningful, write an invariant (precondition) to document this.

This is not a course in OO Design, but we can write down some general principles. If you stray too far from these, that is an indication of a bad design

Step 6: Program Review

- Same as before, plus one more

The Program Review Recipe

1. Do all the tests pass?
2. Are the contracts accurate?
3. Are the purpose statements and interpretations clear and accurate?
4. Are there ugly pieces of code that should be broken out into their own functions?
5. Are there pieces of code that are duplicated (or almost duplicated) and should be made into independent functions?
6. Does your design follow the Principles of a Good OO Design (on the preceding slide)?

Summary

- The Design Recipe is still there, but has been adapted to the object-oriented paradigm.
- The deliverables are in different places
- You should be able to follow the OO design recipe, putting the deliverables where they should go in your object-oriented programs.

Next Steps

- Study the files in the Examples folder. Did we get all the deliverables in the right places?
- If you have questions about this lesson, ask them on the Discussion Board.