Table of Contents

SOLID Design Principles In Common Lisp

Learn how to organize your code with CLOS for better maintainability.

v0.1 (going through major update)

If you find any problem, want to suggest an improvement or commit changes to this book, please visit the project repository and open a pull request or an issue at https://github.com/common-lisp-reserve/solid-design-principles-in-common-lisp

Disclaimer

Common Lisp's CLOS is a very powerful object system. While SOLID principles can and often be helpful in other languages, it provides little benefits in Lisp. However, we will demonstrate an idiomatic Lisp alternative of some of the SOLID principles, with multi-method (multiple dispatch) and protocol approaches. We will show you why Lisp approaches are better (in Lisp) and why problems that SOLID principles are trying to solve in other languages, almost disappears in Lisp with the existence of CLOS.

We are going to use two different programming languages for this book code examples.

- PHP (version 7)
- Common Lisp

This is not an introductory book to object-oriented programming neither to Common Lisp nor PHP. This book assumes the reader has some basic programming experience with some OOP in Lisp and other languages.

What is SOLID?

SOLID is an acronym from

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

It is a set of principles made popular by Robert C. Martin intended to make software designs more understandable, flexible and maintainable. Read further at https://en.wikipedia.org/wiki/SOLID.

S: Single Responsibility

A class (or method, function, etc) should have one reason to change.

The idea is to make a class or similar only handle a single relevant responsibility (although, this is up to the programmer to decide).

Martin suggests that we define each responsibility of a class as a reason for change. If you can think of more than one motivation for changing a class, it probably has more than one responsibility.

See below for a SRP violation example.

SRP Violation (Lisp)

```
(defclass status-report-mailer ()
  ((address
    :initarg :address
    :reader address
    :type string)

   (status-number
    :initarg :status-number
    :reader status-number
    :type integer)

   (boot-time
    :initarg :boot-time
    :reader boot-time
    :type integer)))

(defmethod deliver ((status-report-mailer status-report-mailer))
  (format t
          "~a:~a~%"
          (address status-report-mailer)
          (generate-report status-report-mailer)))

(defmethod generate-report ((status-report-mailer status-report-mailer))
  (format nil
          "Status Number: ~a~%Boot Time: ~a"
          (status-number status-report-mailer)
          (boot-time status-report-mailer)))

(defparameter *mailer*
  (make-instance 'status-report-mailer
                 :address "admin@email.com"
                 :status-number 205
                 :boot-time 82))
```

As you can see above, `status-report-mailer` class is handling both distinct functionalities. Report generation and report delivery. This will force you to modify `status-report-mailer` class if you wish to set up a new value or generate different kind of report template, which it has nothing to do with.

Let's fix this by moving `generate-report` method into its own class.

```
(defclass status-report-mailer ()
  ((address
    :initarg :address
    :reader address
    :type string)

   (report
    :initarg :report
    :reader report
    :type string)))

(defclass status-report-generator ()
  ((status-number
    :initarg :status-number
    :reader status-number)

   (boot-time
    :initarg :boot-time
    :reader boot-time)))

(defmethod deliver ((status-report-mailer status-report-mailer))
  (format t
          "Send email to ~a with content:
           ~a~%"
          (address status-report-mailer)
          (report status-report-mailer)))

(defmethod generate ((status-report-generator status-report-generator))
  (format nil
          "Status Number: ~a
           Boot Time: ~a"
          (status-number status-report-generator)
          (boot-time status-report-generator)))

(defparameter *mailer*
  (make-instance 'status-report-mailer
                 :address "admin@email.com"
                 :report (generate (make-instance
                                    'status-report-generator
                                    :status-number 323
                                    :boot-time 167))))

(deliver *mailer*)
```

and you can apply SRP to other things, like pure functions too!

O: Open/Closed

Objects or entities should be open for extension, but closed for modification.

What this means is that we should write code that doesn't have to be changed every time the requirements changes. For instance, an object should be easily extendable without modifying the object itself.

Take a look at the open/closed principle violation example below.

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader radius)))

(defclass compound-shape ()
  ((shapes
    :initarg :shapes
    :reader shapes)))

(defmethod total-area ((compound-shape compound-shape))
  (reduce #'+
          (mapcar #'(lambda (x)
                      (* pi
                         (radius x)
                         (radius x)))
                  (shapes compound-shape))))

(defparameter *circle-one*
  (make-instance 'compound-shape
                 :shapes
                 (list (make-instance 'circle :radius 5)
                       (make-instance 'circle :radius 6)
                       (make-instance 'circle :radius 2))))

(total-area *circle-one*) ;; 204.20352248333654d0
```

If we do want `total-area` method to calculate a sum of Rectangle areas instead of Circle, we won't be able to do that due to its specific area calculation formula (a = pi * r^2) for a circle area without modifying `total-area` method.

So how can we go over this limit?

Take a look below..

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader get-radius)))

(defmethod area ((circle circle))
  (* pi (get-radius circle) (get-radius circle)))

(defclass compound-shape ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((compound-shape compound-shape))
  (reduce #'+
          (mapcar #'area
                  (get-shapes compound-shape))))

(defparameter *circle-one*
  (make-instance 'compound-shape
                 :shapes
                 (list (make-instance 'circle :radius 5)
                       (make-instance 'circle :radius 6)
                       (make-instance 'circle :radius 2))))

(total-area *circle-one*) ;; 204.20352248333654d0
```

As you've noticed, we moved the function to calculate circle area into its Circle class. This way, if we want to calculate a Rectangle shape area (or triangle, etc), we only have to create a new class with its own method to handle Rectangle area calculation.

but a better way to do it in Lisp, is through multi-method.

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader radius
    :type integer)))

(defclass rectangle ()
  ((width
    :initarg :width
    :reader width
    :type integer)

   (height
    :initarg :height
    :reader height
    :type integer)))

(defclass compound-shape ()
```

```
  ((shapes
    :initarg :shapes
    :reader shapes
    :type list)))

(defgeneric area (shape)
  (:documentation "calculate an area given the shape class"))

(defmethod area ((circle circle))
  (* pi
     (radius circle)
     (radius circle)))

(defmethod area ((rectangle rectangle))
  (* (width rectangle)
     (height rectangle)))

(defmethod total-area ((compound-shape compound-shape))
  (reduce #'+
          (mapcar #'area
                  (shapes compound-shape))))

(defparameter *total-circle-area*
  (total-area
   (make-instance 'compound-shape
                  :shapes
                  (list
                   (make-instance 'circle
                                  :radius 5)
                   (make-instance 'circle
                                  :radius 6)))))

(defparameter *total-rectangle-area*
  (total-area
   (make-instance 'compound-shape
                  :shapes
                  (list
                   (make-instance 'rectangle
                                  :width 5
                                  :height 12)
                   (make-instance 'rectangle
                                  :width 6
                                  :height 10)))))

*total-circle-area*
*total-rectangle-area*
```

With multi-method, you don't have to follow the same way of how open/closed principle implemented in other languages. Just make a generic function that accepts different kind of shapes, and implement a distinct behavior based on the given classes through `defmethod`. There is no need for Open/Closed principle anymore with multi-method.

L: Liskov Substitution

Let *Φ(x) be a property provable about objects x of type T*. Then *Φ(y) should be true for objects y of type S where S is a subtype of T.*

in other words: Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

```
(defclass rectangle ()
  ((width
    :initarg :width
    :initform 0
    :reader get-width
    :accessor width)

   (height
    :initarg :height
    :initform 0
    :reader get-height
    :accessor height)))

(defmethod area ((rectangle rectangle))
  (* (get-width rectangle)
     (get-height rectangle)))

(defclass square (rectangle)
  nil)

(defmethod set-width ((square square) w)
  (setf (width square) w)
  (setf (height square) w))

(defmethod set-height ((square square) h)
  (setf (height square) h)
  (setf (width square) h))

(defparameter square-area (make-instance 'square))

(set-width square-area 5)
(set-height square-area 10)
(area square-area) ;; 100 instead of 50
```

You probably want to do like above but see this - https://en.wikipedia.org/wiki/Circle-ellipse_problem#Change_the_programming_language

I: Interface Segregation

Clients should not be forced to depend upon interfaces that they do not use.

Because Common Lisp in particular doesn't have interface (but we got ***macros*** to properly add one, but let's another story), we will be using PHP 7 code to demonstrate bad use of interface.

So, what is all the fuss about Interface Segregation?

Basically, you don't have to implement and to depend on methods that are irrelevant for the client (eg. a class).

Let's see why this is bad, below.

```php
<?php

interface iBird {
    public function eat();
    public function sleep();
    public function fly();
    public function swim();
}

class Parrot implements iBird
{
    public function eat()
    {
        echo "the parrot eats\n";
    }

    public function sleep()
    {
        echo "the parrot sleeps\n";
    }

    public function fly()
    {
        echo "the parrot flies\n";
    }

    public function swim()
    {
        echo "this is wrong. a parrot can't really swim!\n";
    }
}

class Penguin implements iBird
{
    public function eat()
    {
```

```
        echo "the penguin eats\n";
    }

    public function sleep()
    {
        echo "the penguin sleeps\n";
    }

    public function fly()
    {
        echo "this is wrong. a penguin cannot fly!\n";
    }

    public function swim()
    {
        echo "the penguin swims\n";
    }
}

$blue = new Parrot();
$blue->eat();
$blue->sleep();
$blue->fly();
$blue->swim();

$ned = new Penguin();
$ned->eat();
$ned->sleep();
$ned->fly();
$ned->swim();
```

A Penguin can't fly. A Parrot can't (let's say) run. These classes doesn't have to depend on `fly` and `swim` methods where they doesn't make sense. You see, both Penguin and Parrot `is a` Bird, both also can eat and sleep, but they don't behave the same way and both differs in abilities and weaknesses (one can fly, but the other doesn't). These extra useless code can lead to redundancy.

Let's make it better, shall we?

```
<?php

interface iBird {
    public function eat();
    public function sleep();
}

interface iFlyingBird extends iBird {
    public function fly();
}

interface iFlightlessBird extends iBird {
    public function swim();
}
```

```
class Parrot implements iFlyingBird
{
    public function eat()
    {
        echo "the parrot eats\n";
    }

    public function sleep()
    {
        echo "the parrot sleeps\n";
    }

    public function fly()
    {
        echo "the parrot flies\n";
    }
}

class Penguin implements iFlightlessBird
{
    public function eat()
    {
        echo "the penguin eats\n";
    }

    public function sleep()
    {
        echo "the penguin sleeps\n";
    }

    public function swim()
    {
        echo "the penguin swims\n";
    }
}

$blue = new Parrot();
$blue->eat();
$blue->sleep();
$blue->fly();

$ned = new Penguin();
$ned->eat();
$ned->sleep();
$ned->swim();
```

Better.

We placed them into two distinct categories that also implements `iBird`, and extended it into `iFlyingBird` and `iFlightlessBird` interfaces. Both can eat and sleep, but only the one that implements `iFlightlessBird` have to also implement `run` method. This is the same for the client that

implements `iFlyingBird`. It just need to care for extra `fly` method. No need to implement methods that a certain client won't need. Hurray!

D: Dependency Inversion

- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
- **Abstractions should not depend upon details. Details should depend upon asbtractions.**

Dependency Inversion Principle encourages us to create higher level modules with its complex logic in such a way to be reusable and unaffected by any change from the lower level modules in our application. To achieve this kind of behavior in our apps, we introduce abstraction which decouples higher from lower level modules.

Wait! What are these *low-level* and *high-level* modules again?

Well, low-level modules are more specific to the individual components focusing os smaller details of the application. This *low-level* module should be used in *high-level* modules in the application.

While *high-level* modules are more abstract and general in nature. They handle *low-level* modules and decide the logic for what goes where.

Think about a computer CPU (high-level) handling bunch of hardware inputs (low-level) including keyboard and mouse inputs.

See DI violation below.

```
(defclass printer ()
  ((amount
    :initarg :amount
    :reader amount
    :type integer)))

(defclass epub-formatter () nil)
(defclass mobi-formatter () nil)

(defmethod print-epub ((printer printer))
  (let ((e (make-instance 'epub-formatter)))
    (process e)))

(defmethod print-mobi ((printer printer))
  (let ((m (make-instance 'mobi-formatter)))
    (process m)))

(defmethod process ((epub-formatter epub-formatter))
  (format t "~a~%" "epub formatter logic goes here"))

(defmethod process ((mobi-formatter mobi-formatter))
  (format t "~a~%" "mobi formatter logic goes here"))


(defparameter *epub* (make-instance 'printer :amount 100))
(print-epub *epub*)

(defparameter *mobi* (make-instance 'printer :amount 200))
(print-mobi *mobi*)
```

`printer` (high-level) class had to depend on `print-epub` and `print-mobi` which are both low-level modules. This breaks the " *High level modules should not depend upon low level modules. Both should depend upon abstractions.*" keypoint.

Let's fix it below.

```
(defclass printer ()
  ((amount
    :initarg :amount
    :reader amount
    :type integer)))

(defclass epub (printer) nil)
(defclass mobi (printer) nil)

(defgeneric print-book (book-format)
  (:documentation "print a book given the book format class"))

(defmethod print-book ((epub epub))
  (format t
          "~a: ~a~%"
          "epub formatter logic goes here"
          (amount epub)))

(defmethod print-book ((mobi mobi))
  (format t
          "~a: ~a~%"
          "mobi formatter logic goes here"
          (amount mobi)))

(defparameter *gatsby-in-epub*
  (make-instance 'epub
                 :amount 100))

(defparameter *dracula-in-mobi*
  (make-instance 'mobi
                 :amount 900))

(print-book *gatsby-in-epub*)
(print-book *dracula-in-mobi*)
```

We have moved both `print-epub` and `print-mobi` into separate classes. Now they can do their own things, and whenever we need to print an ebook, we are going to pass either `epub` or `mobi` classes to `printer` constructor, then call `print-book` to execute the method based on the class we passed.

Now, our classes are not tightly coupled with the lower-tier objects and we can easily reuse the logic from the high-tier modules.

And if you've already noticed, this approach is the same one just like the multi-method in SRP and O/P chapters!