Table of Contents

SOLID Design Principles In Common Lisp

Learn how to apply SOLID design principles with Common Lisp and the powerful CLOS system.

If you find any problem, want to suggest an improvement or commit changes to this book, please visit this Github repository https://github.com/common-lisp-reserve/solid-design-principles-in-common-lisp

What is SOLID?

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

S: Single Responsibility

A class should have one, and only one, reason to change.

Bad

```
(defclass truck ()
  ((brand
   :initarg :brand
   :accessor brand)))

(defmethod get-brand ((self truck))
  (brand self))

(defmethod set-brand ((self truck) new-brand)
  (setf (brand self) new-brand))

(defmethod send-detail ((self truck) customer-id)
  "send truck's brand detail to customer..")

(defclass truck ()
  ((brand
   :initarg :brand
   :accessor brand)))

(defmethod get-brand ((self truck))
  (brand self))

(defmethod set-brand ((self truck) new-brand)
  (setf (brand self) new-brand))
```

Good

```
(defclass detail-sender ()
  ((customer-id
    :initarg :customer-id
    :accessor customer-id)))

(defmethod get-customer-id ((self detail-sender))
  (customer-id self))

(defmethod set-customer-id ((self detail-sender) new-customer-id)
    (setf (customer-id self) new-customer-id))
```

```
(defmethod send-detail ((self detail-sender))
  (send (customer-id self)))
```

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, a class should be easily extendable without modifying the class itself.

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

Bad

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

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((self area-calculator))
  (reduce #'+
          (mapcar #'(lambda (x)
                      (* pi
                         (get-radius x)
                         (get-radius x)))
                  (get-shapes self))))
```

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) without modifying `total-area` method.

So how can we go over this limit?

Below code shows a better example.

Good

```
(defclass shape ()
  nil)

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

```
(defmethod area ((self shape))
  (* pi (get-radius self) (get-radius self)))

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((self area-calculator))
  (reduce #'+
          (mapcar #'area
                  (get-shapes self))))
```

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.

For example, a new Rectangle class and area method.

```
(defmethod rectangle (shape)
  ((width
    :initarg :width
    :reader get-width)
   (height
    :initarg :height
    :reader get-height)))

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

Full Better Example

```
(defclass shape ()
  nil)

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

(defmethod area ((self shape))
  (* pi (get-radius self) (get-radius self)))

(defmethod rectangle (shape)
```

```
  ((width
    :initarg :width
    :reader get-width)
   (height
    :initarg :height
    :reader get-height)))

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

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((self area-calculator))
  (reduce #'+
          (mapcar #'area
                  (get-shapes self))))
```

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*.

.

L: Interface Segregation

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

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.**