

---

# Table of Contents

Introduction	1.1
Disclaimer	1.2
What is SOLID?	1.3
S: Single Responsibility	1.4
O: Open/Closed	1.5
L: Liskov Substitution	1.6
D: Dependency Inversion	1.7

# SOLID Design Principles In Common Lisp

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



This book is released under the [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/) by [Momozor](#).

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>

## Disclaimer

These principles are not rules. Apply them where you think it does make sense. Don't "over-engineer".

We are going to focus on Single Responsibility, Open/Closed, Liskov Substitution and Dependency Inversion principles where they makes the most sense to apply with Common Lisp as a dynamic typed language.

## What is SOLID?

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle (we are not going to cover this)
- Dependency Inversion Principle

## S: Single Responsibility

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

### Bad

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

   (report
    :initarg :report
    :initform ""
    :reader get-report
    :accessor report)))

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

(defmethod generate-report ((status-report-mailer status-report-mailer))
  (let ((r (concatenate 'string
    "status number: "
    (write-to-string (random 500))
    ". this is a status report for slow server boot time "
    "estimating around "
    (write-to-string (random 200))
    " seconds from time to fully boot.")))
    (setf (report status-report-mailer) r)))

(defparameter rm1
  (make-instance 'status-report-mailer
    :address "dummy@email.com"))

;; generated status and boot time may differ from yours
(generate-report rm1)
;; "status number: 361. this is a status report for slow server
;; boot time estimating around 173
;; seconds from time to fully boot."
(deliver rm1)
;; send email to dummy@email.com with email content/body:
;; status number: 361. this is a status report for
;; slow server boot time estimating around
```

```
;; 173 seconds from time to fully boot.
```

## Good

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

   (report
    :initarg :report
    :reader get-report)))

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

(defclass status-report-generator ()
  nil)

(defmethod generate ((status-report-generator status-report-generator))
  (concatenate 'string
    "status number: "
    (write-to-string (random 500))
    ". this is a status report for slow server boot time "
    "estimating around "
    (write-to-string (random 200))
    " seconds from time to fully boot.))

(defparameter report-data
  (make-instance 'status-report-generator))
(defparameter mailer
  (make-instance 'status-report-mailer
    :address "dummy@email.com"
    :report (generate report-data)))

(deliver mailer)
;; send email to dummy@email.com with email content/body:
;; status number: 91. this is a status report for
;; slow server boot time estimating around 70
;; seconds from time to fully boot.
```

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

(defparameter *circle-one*
  (make-instance 'area-calculator
    :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 ( $a = \pi * r^2$ ) 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 ((circle circle))
  (* pi (get-radius circle) (get-radius circle)))

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

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

(defparameter *circle-one*
  (make-instance 'area-calculator
    :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.

For example, a new Rectangle class and area method which calculates a simple Rectangle shape area ( $a = w * h$ )

```
(defclass rectangle (shape)
  ((width
    :initarg :width
    :reader get-width)

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

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



## Full Better Example

```
(defclass shape ()
  nil)

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

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

(defclass rectangle (shape)
  ((width
    :initarg :width
    :reader get-width)

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

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

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

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

(defparameter *circle-one*
  (make-instance 'area-calculator
    :shapes
    (list (make-instance 'circle :radius 5)
          (make-instance 'circle :radius 3)
          (make-instance 'circle :radius 12))))

(defparameter *rectangle-one*
  (make-instance 'area-calculator
    :shapes
    (list (make-instance 'rectangle :height 5 :width 10)
          (make-instance 'rectangle :height 9 :width 20)
          (make-instance 'rectangle :height 23 :width 44))))
```

```
(total-area *circle-one*) ;; 559.2034923389832d0  
(total-area *rectangle-one*) ;; 1242
```

## **L: Liskov Substitution**

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

.

## 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 abstractions.

### Bad

```
(defclass printer ()
  ((data-type
    :initarg :data-type
    :reader get-data-type)))

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

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

(defclass epub-formatter ()
  nil)

(defmethod process ((epub-formatter epub-formatter) data-type)
  (format t "~a-%data-type: ~a~%"
    "epub formatter's process logic goes here"
    data-type))

(defclass mobi-formatter ()
  nil)

(defmethod process ((mobi-formatter mobi-formatter) data-type)
  (format t "~a-%data-type: ~a~%"
    "mobi formatter's process logic goes here"
    data-type))

(defparameter epub-book (make-instance 'printer :data-type "epubs"))
(defparameter mobi-book (make-instance 'printer :data-type "mobis"))

(print-epub epub-book)
;; epub formatter's process logic goes here
;; data-type: epubs

(print-mobi mobi-book)
;; mobi formatter's process logic goes here
```

```
;; data-type: mobis
```

## Good

```
(defclass printer ()
  ((data-type
    :initarg :data-type
    :reader get-data-type)))

(defmethod prints ((printer printer) formatter)
  (let ((f (make-instance formatter)))
    (process f (get-data-type printer))))

(defclass epub-formatter ()
  nil)

(defmethod process ((epub-formatter epub-formatter) data-type)
  (format t "~a-%data-type: ~a~%"
    "epub formatter's process logic goes here"
    data-type))

(defclass mobi-formatter ()
  nil)

(defmethod process ((mobi-formatter mobi-formatter) data-type)
  (format t "~a-%data-type: ~a~%"
    "mobi formatter's process logic goes here"
    data-type))

(defparameter epub-book (make-instance 'printer :data-type "epubs"))
(defparameter mobi-book (make-instance 'printer :data-type "mobis"))

(prints epub-book 'epub-formatter)
;; epub formatter's process logic goes here
;; data-type: epubs

(prints mobi-book 'mobi-formatter)
;; mobi formatter's process logic goes here
;; data-type: mobis
```