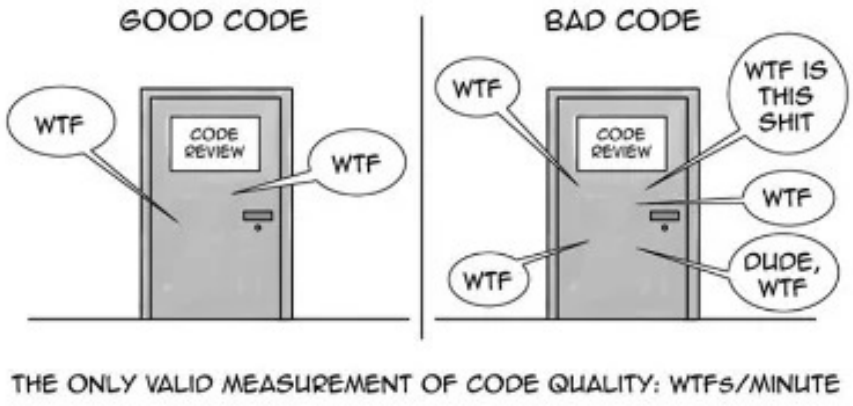
The SOLID Principles

Jan Wedekind

Tuesday, Jan 21st 2025

Motivation



Find guiding design principles to maintain software quality over time.

Software Rot

Symptoms of rotting software design:[a]

- **Rigidity**: software difficult (a lot of work) to change
- **Fragility**: changes easily break the software
- **Immobility**: it is easier to rewrite than reuse parts
- **Viscosity**: design preserving methods are harder to employ than hacks

[a] Robert C. Martin: Design Principles and Design Patterns

Aims

In contrast we want to achieve the following:[a]

- Keep software application **flexible**
- Keep software application **robust**
- Keep software application **reusable**
- Keep software application **developable**

[a] Robert C. Martin: Design Principles and Design Patterns

SOLID Authors

Robert C. Martin

- Author of Clean Code, Functional Design, and more books
- Author of Design Principles and Design Patterns paper based on his experience and on work by Bertrand Meyer, Barbara Liskov, and Erich Gamma et al.
- `http://cleancoder.com/`

Michael Feathers

- Author of Working Effectively With Legacy Code
- Summarized Robert C. Martin's paper using the SOLID acronym
- `https://www.r7krecon.com/`

The SOLID Principles

1. **S**ingle responsibility
2. **O**pen-closed
3. **L**iskov substitution
4. **I**nterface segregation
5. **D**ependency inversion

Single Responsibility - Before

```clojure
(defn adults-to-html [people]
  (str
    "<ul>\n"
    (apply str
           (for [person people :when (>= (:age person) 18)]
                (str "<li>" (:name person) "</li>\n")))
    "</ul>"))
; ...
(def page (adults-to-html people))
```

Single Responsibility - After

```clojure
(defn select-adults [people]
  (filter
    (fn [person] (>= (:age person) 18))
    people))

(defn people-to-html [people]
  (str
    "<ul>\n"
    (apply str
           (for [person people]
             (str "<li>" (:name person) "</li>\n")))
    "</ul>"))
; ...
(def page (people-to-html (select-adults people)))
```

Open-Closed - Before

```
(defn total-area [shapes]
  (reduce +
    (map
      (fn [shape]
        (case (:type shape)
          ::circle (* Math/PI (:radius shape) (:radius shape))
          ::rectangle (* (:width shape) (:height shape))))
      shapes)))
```

Open-Closed - After

```
(defmulti area :type)

(defmethod area ::circle [shape]
  (* Math/PI (:radius shape) (:radius shape)))

(defmethod area ::rectangle [shape]
  (* (:width shape) (:height shape)))

(defn total-area [shapes]
  (reduce +
    (map area shapes)))
```

Liskov-Substitution - Before

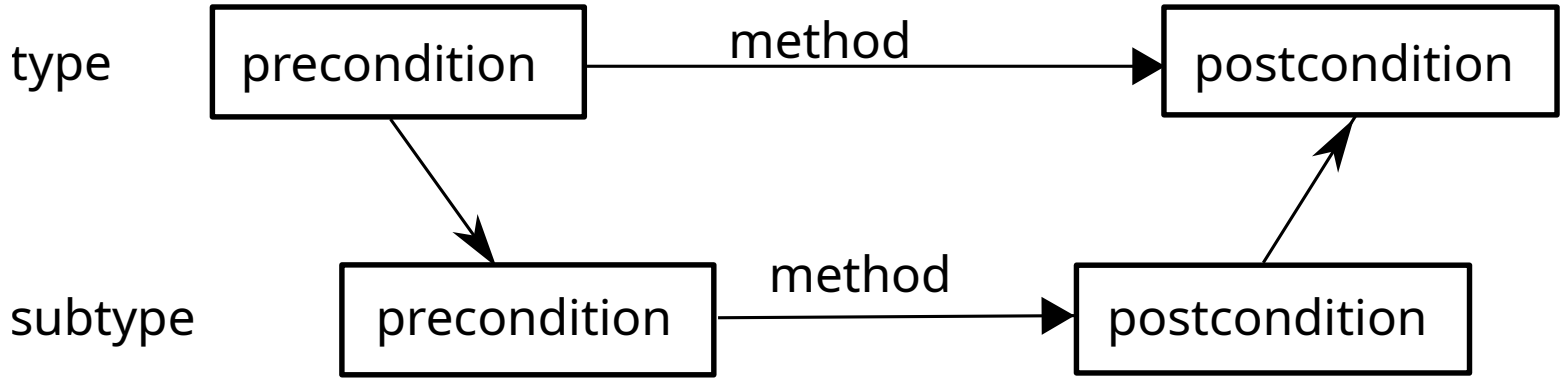
```
(defmulti set-width (fn [x width] (:type x)))
(defmethod set-width ::rectangle [x width]
  (assoc x :width width))
(defmethod set-width ::square [x width]
  (assoc x :width width :height width))

(defmulti set-height (fn [x height] (:type x)))
(defmethod set-height ::rectangle [x height]
  (assoc x :height height))
(defmethod set-height ::square [x height]
  (assoc x :width height :height height))

(defmulti area :type)
(defmethod area ::rectangle [x] (* (:width x) (:height x)))
(derive ::square ::rectangle)
```

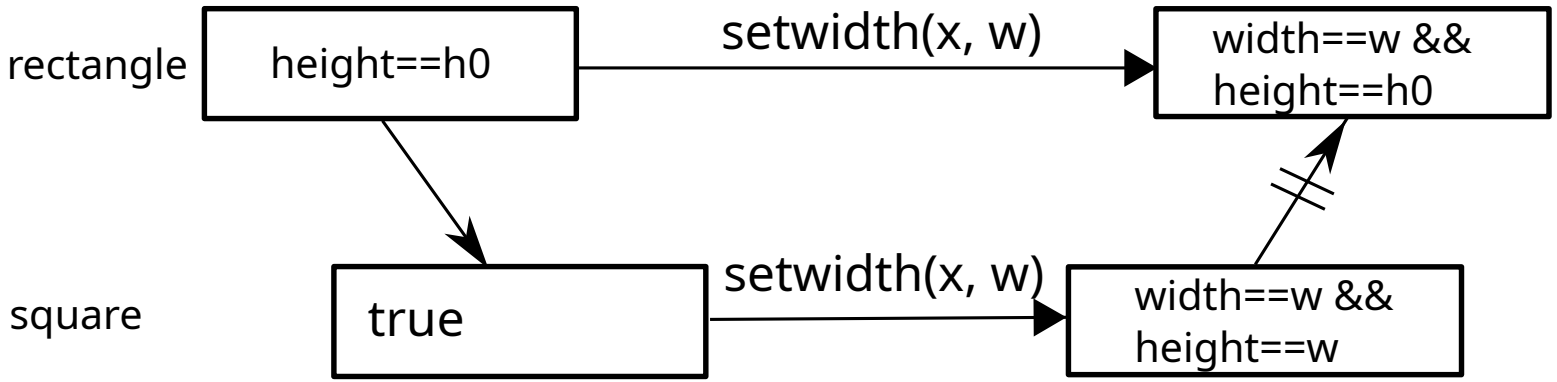
Liskov-Substitution - Contracts

"The Liskov Substitution Principle states, among other constraints, that a subtype is not substitutable for its super type if it strengthens its operations' preconditions, or weakens its operations' postconditions" [a]



[a] Baniassad: Making the Liskov Substitution Principle Happy and Sad

Liskov-Substitution - Example

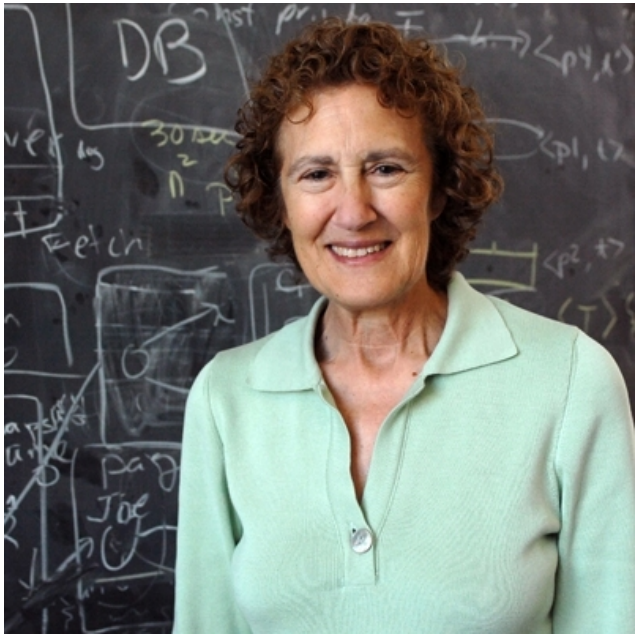


Liskov-Substitution - After

```
(defn set-width [x width]
  (assoc x :width width))
(defn set-height [x height]
  (assoc x :height height))

(defn set-side [x side]
  (assoc x :side side))

(defmulti area :type)
(defmethod area ::rectangle [x]
    (* (:width x) (:height x)))
(defmethod area ::square [x]
    (* (:side x) (:side x)))
```



Barbara Liskov

Interface Segregation - Before

```clojure
(defrecord AccountHolder [name age balance])

(defn is-adult [account-holder] (>= (:age account-holder) 18))

(defn deposit [account-holder amount]
  (update account-holder :balance + amount))

(defn withdraw [account-holder amount]
  (update account-holder :balance - amount))
```

Interface Segregation - After

```clojure
(defrecord Person [name age])

(defn is-adult [person]
  (>= (:age person) 18))

(defrecord Account [balance])

(defn deposit [account amount]
  (update account :balance + amount))

(defn withdraw [account amount]
  (update account :balance - amount))

(defrecord AccountHolder [person account])
```

Dependency Inversion - Before

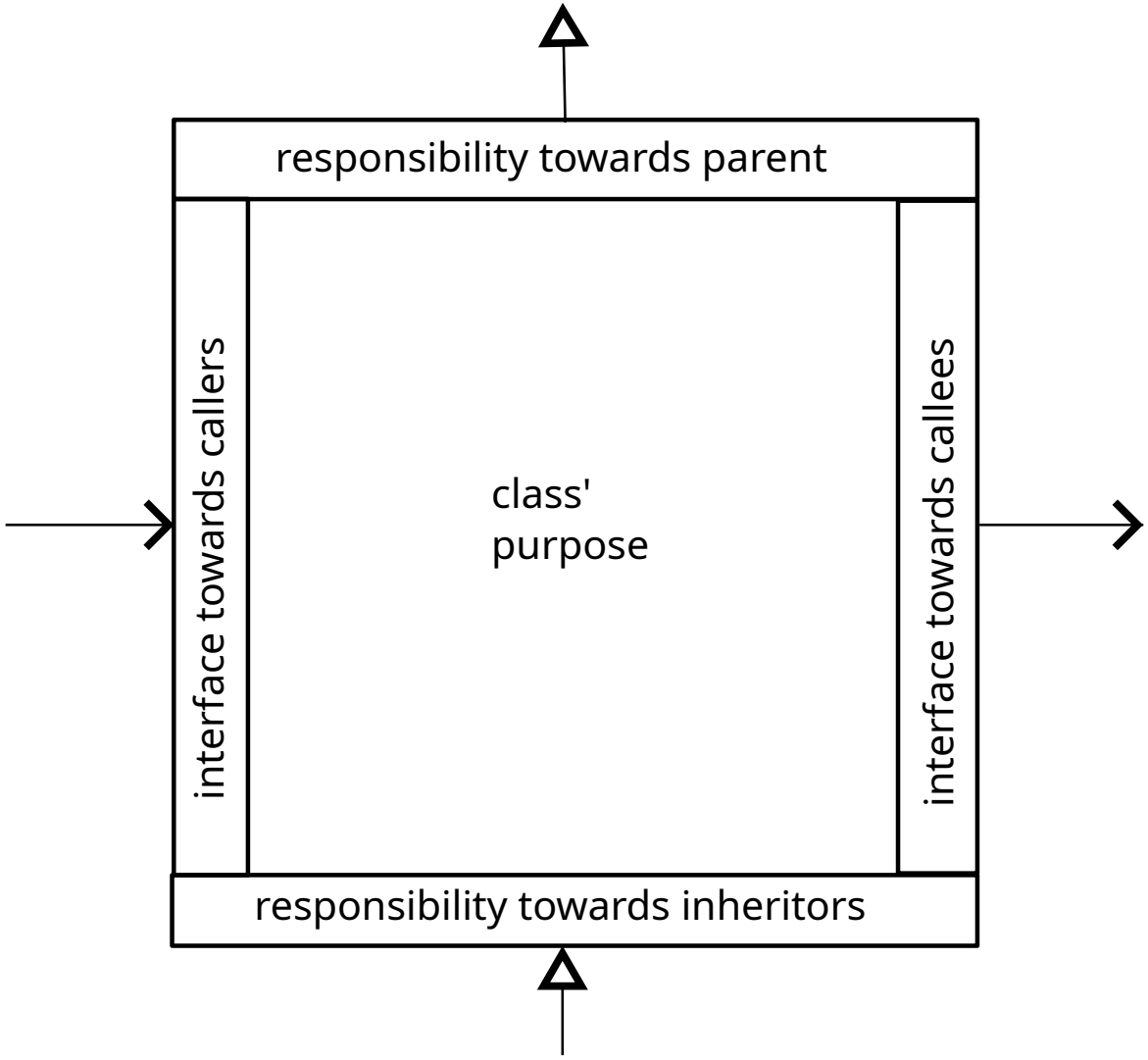
```clojure
(defn print-names [people print-writer]
  (doseq [person people]
         (.println print-writer (:name person))))
; ...
(print-names [{:name "Alice"} {:name "Bob"}] *out*)
```

Dependency Inversion - After

```clojure
(defn print-names [people writer]
  (doseq [person people]
         (.write writer (:name person))
         (.write writer "\n")))
; ...
(print-names [{:name "Alice"} {:name "Bob"}] *out*)
; ...
(with-open [f (java.io.FileWriter. "names.txt")]
  (print-names [{:name "Alice"} {:name "Bob"}] f))
```

Aspects of a Class

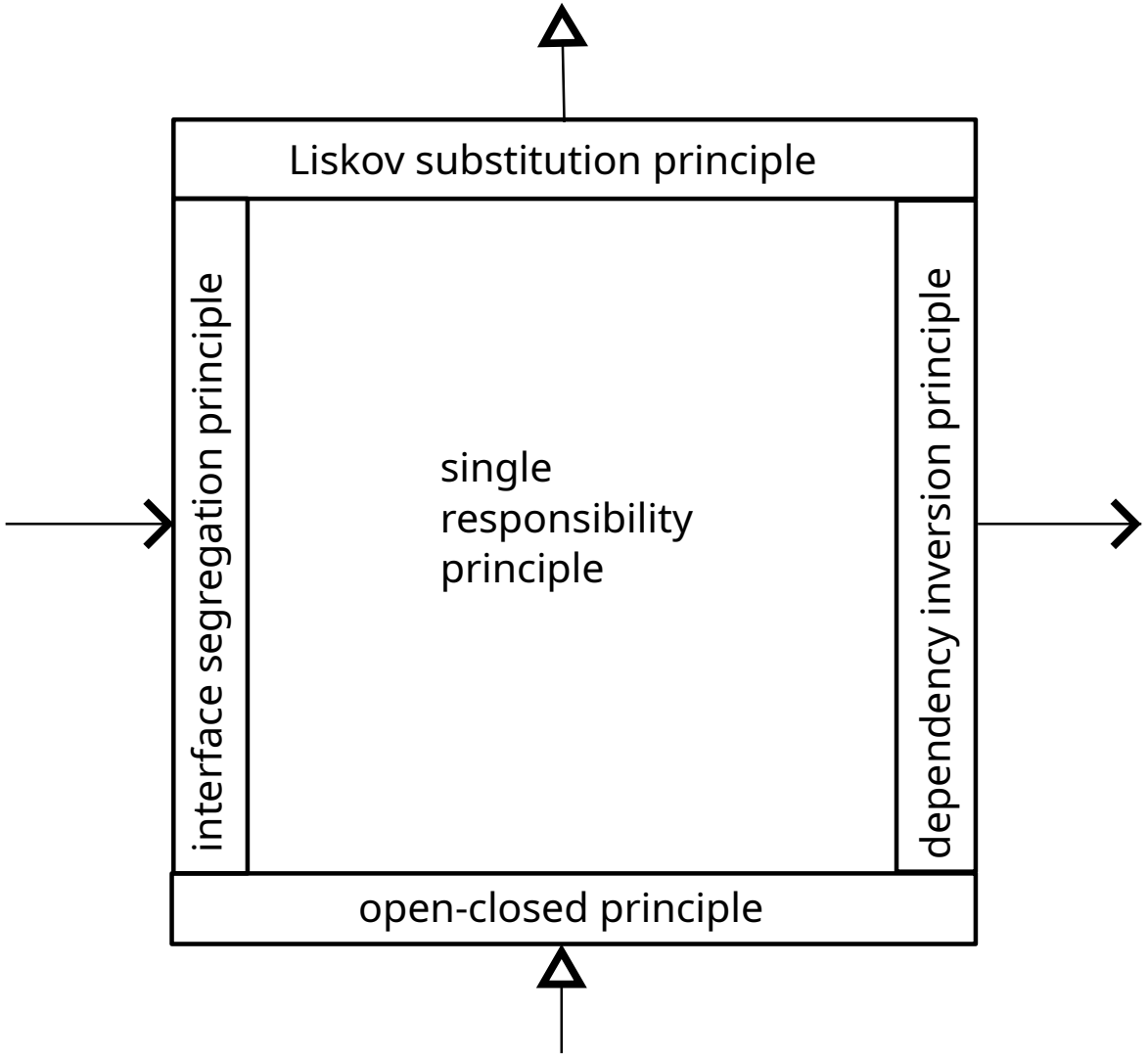
The 5 aspects of the class are:[a]



[a] Mike Lindner: The Five Principles For SOLID Software Design

The 5 Principles

The 5 corresponding principles are:[a]



[a] Mike Lindner: The Five Principles For SOLID Software Design

Thanks