SOLID Clojure

Colin Jones / Software Craftsman / 8th Light
@trptcolin

What paradigm is home for you?

What paradigm is home for you?

Object-oriented?

What paradigm is home for you?

Functional?

What paradigm is home for you?

Logic?

What paradigm is home for you?

Stack-based? Concatenative?

What paradigm is home for you?

Which one is Clojure again?

#Clojure is one of the most object-oriented languages you can find today.

- @takeoutweight

Axiom:

Everybody wants the same thing


easy
STAPLES

SOLID principles

Principles

how do they work?

SOLID principles

~~SOLID~~ DILOS principles

DILOS principles

DILOS

Dependency Inversion

Interface Segregation

Liskov Substitution

Open-Closed

Single Responsibility

Dependency Inversion Principle

Problem to solve:

A change in one place causes too many other changes

Problem to solve:

Rigidity, fragility

A common cause

Higher-level modules depend on lower-level details

Higher level

forced to reuse lower level

Clients of the higher level

forced to reuse lower level

Clients of clients of the higher level

forced to reuse lower level

Let's make this concrete

Example: nREPL

One approach to transport might have been:

```
(defn recv []
  (be/payload>string (be/read-bytes (.getInputStream socket))))
(defn send [msg]
  (let [os (.getOutputStream socket)]
    (.write os (be/string>payload (.getBytes msg)))
    (.flush os)))
(defn echo []
  (when-let [msg (recv)]
    (send msg)
    (recur)))
```

Depends on:

bencode

sockets

```
(defn recv []
  (be/payload>string (be/read-bytes (.getInputStream socket))))
(defn send [msg]
  (let [os (.getOutputStream socket)]
    (.write os (be/string>payload (.getBytes msg)))
    (.flush os)))
(defn echo []
  (when-let [msg (recv)]
    (send msg)
    (recur)))
```

Java

Aspect-oriented programming

Dependency injection

Clojure

Dynamic binding

Clojure

Dynamic binding all over the place: good idea?

We can do better

Depend on abstractions

...not concretions

OOP

Interfaces & abstract classes

Clojure, the language:

ISeq, IDeref, IPersistentVector, etc.

Clojure

Interfaces & abstract classes

Clojure

definterface, gen-class

Clojure

That's it! NEXT PRINCIPLE.

Clojure

(just kidding)

Clojure

Abstraction: defprotocol

Concretion: defrecord, deftype

Clojure

Abstraction: defmulti

Concretion: defmethod

Example: nREPL

The naive version, again

```
(defn recv []
  (be/payload>string (be/read-bytes (.getInputStream socket))))
(defn send [msg]
  (let [os (.getOutputStream socket)]
    (.write os (be/string>payload (.getBytes msg)))
    (.flush os)))
(defn echo []
  (when-let [msg (recv)]
    (send msg)
    (recur)))
```

Example: nREPL

The real deal

```
(defprotocol Transport
  "Defines the interface for a wire protocol implementation
  for use with nREPL."
  (recv [this] [this timeout]
    "Reads and returns the next message received. Will block.
    Should return nil if the message is not available after
    `timeout` ms or if the underlying channel has been closed.")
  (send [this msg]
    "Sends msg. Implementations should return the transport."))
```

Example: nREPL

```
(deftype FnTransport [recv-fn send-fn close]
  Transport
  (send [this msg]
    (-> msg clojure.walk/stringify-keys send-fn)
    this)
  (recv [this] (.recv this Long/MAX_VALUE))
  (recv [this timeout]
    (clojure.walk/keywordize-keys (recv-fn timeout)))
  java.io.Closeable
  (close [this] (close)))
```

Clojure

What is an abstraction?

Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time

-Wikipedia

Example: nREPL

Where are the abstractions?

```
(deftype FnTransport [recv-fn send-fn close]
  Transport
  (send [this msg]
    (-> msg clojure.walk/stringify-keys send-fn)
    this)
  (recv [this] (.recv this Long/MAX_VALUE))
  (recv [this timeout]
    (clojure.walk/keywordize-keys (recv-fn timeout)))
  java.io.Closeable
  (close [this] (close)))
```

Functions are abstractions

in Java-land: IFn

Functions are abstractions

what details do we need to call a function?

Inject abstractions as arguments

```
(deftype FnTransport [recv-fn send-fn close]
  Transport
  (send [this msg]
    (-> msg clojure.walk/stringify-keys send-fn)
    this)
  (recv [this] (.recv this Long/MAX_VALUE))
  (recv [this timeout]
    (clojure.walk/keywordize-keys (recv-fn timeout)))
  java.io.Closeable
  (close [this] (close)))
```

Namespaces are abstractions

Interface Segregation Principle

Problem to solve:

An abstraction is too fat

Concrete problem:

```
(defrecord User [attributes]
  ActiveRecord
  (find [this conditions])
  (save [this])
  (valid? [this])
  (before-create [this])
  (after-update [this])
  ; etc, etc, etc.
  (to-xml [this])
  (to-json [this]))
```

I don't need all that crap

So don't make me implement it!

```
(defrecord User [attributes]
  Storable
  (find [this conditions])
  (save [this])
  Validatable
  (valid? [this]))
```

Clients shouldn't have to depend on abstractions they don't use.

Reasonably small, cohesive protocols

Reasonably small, cohesive namespaces

Easy to do

Common practice in Clojure code

Liskov Substitution Principle

Problem to solve:

A particular subtype is not substitutable for its base type

Problem to solve:

A reasonable reading of the abstraction becomes wrong

Classic OOP Example

```
class Rectangle
  attr_accessor :height, :width
end
```

Classic OOP Example

```
class Square < Rectangle
  def height=(new_height)
    @height = new_height
    @width = new_height
  end
  def width=(new_width)
    @height = new_height
    @width = new_width
  end
end
```

Classic OOP Example

```
r = procure_a_rectangle
r.width = 10
r.height = 5
r.area.should == 50
```

Clojure

Is this problem possible outside Java interop?

Clojure

Concrete inheritance is not in the language

But of course!

Don't let the mutable state fool you

```
(defprotocol Rectangle
  (with-height [this h])
  (with-width [this w])
  (area [this]))
(defrecord Square [side]
  Rectangle
  (with-height [this h] (Square. h))
  (with-width [this w] (Square. w))
  (area [this] (* side side)))
(-> (Square. 3) (with-height 10) (with-width 5) (area))
;=> 25
```

What is a subtype, really?

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

- Barbara Liskov

Let's rephrase the problem:

~~A particular subtype is not substitutable for its base type~~

A particular concretion is not substitutable for its abstraction

Clojure concretions

Protocol/interface implementors

(reify, proxy, defrecord, deftype, etc.)

Clojure concretions

Function implementations

(defn, defmethod)

Liskov substitution:

What can clients expect of an abstraction?

How can we help define expectations?

Clojure

Function pre- and post-conditions (built in)

Trammel (https://github.com/fogus/trammel)

Unit tests

Generative tests

Documentation

Open-Closed Principle

Problem to solve:

Adding a new feature requires changes to existing code

When / why is this a real problem?

(outside the ideal world)

When you don't own the existing code

or you do own it, but it's hard to change

Things should be open for extension, closed for modification

Traditional OO solutions

Avoid switching on type: use polymorphic dispatch instead

Use wrappers for external libraries

The Expression Problem (tm) (r) (c)

Watch chouser's talk & read stuartsierra's article

`extend-type` / `extend-protocol` / `extend`

Multimethods

always open?

Depends on the dispatch fn

```
(defmulti print-dup (fn [x writer] (class x)))
```

vs.

```
(defn collection-tag [x]
  (cond
   (instance? java.util.Map$Entry x) :entry
   (instance? java.util.Map x) :map
   (sequential? x) :seq
   :else :atom))
(defmulti is-leaf collection-tag)
```

We can't extend the dispatch function

All problems in computer science can be solved by another level of indirection

- David Wheeler

```
(defmulti collection-tag class)
(defmethod collection-tag java.util.Map$Entry [x] :entry)
(defmethod collection-tag java.util.Map [x] :map)
(defmethod collection-tag clojure.lang.Sequential [x] :seq)
(defmethod collection-tag :default [x] :atom)

(defmulti is-leaf collection-tag)
```

Problem?

```
(defn collection-tag [x]
  (cond
   (instance? java.util.Map$Entry x) :entry
   (instance? java.util.Map x) :map
   (sequential? x) :seq ;;; <- this
   :else :atom))

(defmulti collection-tag class)
(defmethod collection-tag java.util.Map$Entry [x] :entry)
(defmethod collection-tag java.util.Map [x] :map)
(defmethod collection-tag clojure.lang.Sequential [x] :seq)
(defmethod collection-tag :default [x] :atom)
```

Sci-fi-future-solution: predicate dispatch?

Someone *please* clone David Nolen

...and watch his Conj 2011 talk / help out if you can

Meantime...

for multimethods, think hard about dispatch functions

Single Responsibility Principle

Problem to solve:

A program unit has too many reasons to change

Concrete problem to solve:

A program unit has too many clients
asking for potentially-conflicting changes

Concrete problem to solve:

Reuse of sub-units is impeded

A solution

Give a unit a single reason to change

...aka modularity

OOP

class level

Clojure

protocol/type/record level

Clojure

function level

Project Euler #1

Find the sum of all the multiples of 3 or 5 below 1000.

```
(defn solve-euler-1 []
  (reduce +
          (filter #(or (zero? (mod % 3)) (zero? (mod % 5)))
                  (range 1000))))
```

Project Euler #1

```
(defn old-solve-euler-1 []
  (reduce +
          (filter #(or (zero? (mod % 3)) (zero? (mod % 5)))
                  (range 1000))))

(defn divides? [n divisor]
  (zero? (mod n divisor)))

(defn divides-any? [n & divisors]
  (some #(divides? n %) divisors))

(defn solve-euler-1 []
  (reduce + (filter #(divides-any? % 3 5)) (range 1000)))
```

DILOS

Dependency Inversion

Interface Segregation

Liskov Substitution

Open-Closed

Single Responsibility

~~DILOS~~ SOLID

Dependency Inversion

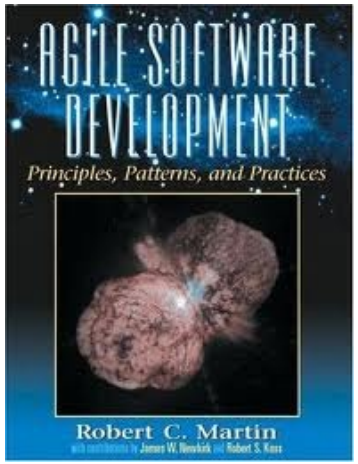
Interface Segregation

Liskov Substitution

Open-Closed

Single Responsibility

More learnings from OO



Don't reinvent the wheel

Think about *why* things are hard

Understand the underlying problems

What else can we learn?

Thank you

Colin Jones / Software Craftsman / 8th Light
@trptcolin
https://github.com/trptcolin/reply
https://github.com/functional-koans/clojure-koans