An Adventure in OO Programming with Clojure

Mikael Svahnberg*

2020-10-30

1 Introduction

I wanted to learn more about `Clojure`. Clojure is a dialect of lisp that runs on top of the Java virtual machine. Through this, you also have access to all of Java's API and third party libraries.

Specifically, I wanted to learn more about Clojure because:

- It is lisp. I've done my fair share of programming in emacs-lisp and I like the simplicity and elegance of it. So how hard can it be to learn a new lisp dialect?

. It is lisp. Along with Fortran it is the oldest high-level programming language still in use today. That alone should be a reason top look closer at it.

- It is not tied in to Emacs. To me personally this is not necessarily a benefit, but if I am going to use it in teaching, it may mean that the entry treshold is lowered.
- StackOverflow publishes an annual developer survey, and in 2019 Clojure came out fairly well in terms of "most loved languages" and salary. In 2020 the survey looked at other aspects, but it is notable that the salaries of data scientists, data engineers and other working-with-data related jobs are "above the curve". So if I am going to advocate using a ...sane... programming language to our students, then why not pick one that is also well payed.

As it turns out, working with Clojure forced me to clarify many of the fundamentals and old habits that I've never really bothered to question before. This text is mostly an attempt at writing up these experiences. Possibly it will help you, dear reader, to also reflect upon object oriented programming.

This is not an introduction to Clojure. There are plenty of books and online resources for learning Clojure. If you are interested in learning more, the following list constitute some of the resources I stumbled upon:

- S. Halloway, "Programming Clojure", Pragmatic Bookshelf, 2009.
- Clojure from the Ground Up

*Mikael.Svahnberg@bth.se

- Clojure for the Brave and True
- Practicalli Clojure
- Clojure By Example
- Clojure cheatsheet

2 What is Object Orientation anyway?

The old mantra says that Object Orientation is *"Encapsulation, Inheritance, and Polymorphism"*. Within the functional programming world, there are plenty of people who question why these qualities are intrinsically tied to object orientation.

Another axiom in creative writing is "kill your darlings". Well, that's more or less what I will do in this section.

2.1 Encapsulation

I recently watched a video by Robert "Uncle Bob" Martin, The Last Programming Language, where he makes an interesting case for how each evolution in programming languages *takes away* some freedom from the programmer. For example, procedural programming takes away the `goto` (It's still there in some languages such as C, but you almost never need to use it).

One thing Object Oriented programming languages takes way the ability to reach everything from everywhere in your program (i.e. the global namespace), by *encapsulating* data and operations on that data into classes. Modula did this through modules before object orientation was invented, and even Pascal had modules for a long time before C++ "invented" namespaces. For example, take Java with its one-file-one-class policy (more or less; you can have e.g. embedded classes that sort of breaks this policy). In essence, this means that you have encapsulation of methods and variables on a per-file basis. It is slightly better than this, since one might argue that the encapsulation is done on an object to object basis (i.e. `MyClass:object1` has its own encapsulation which is separate from `MyClass:object2`).

But what is the use of this encapsulation?

In emacs-lisp you have a global namespace and over 1 million lines of emacs-lisp code shipped with Emacs. So when you name a function you have to make sure that the name is unique and that you are not accidentally redefining a function from somewhere else. The convention is to name all functions with the name of the package (or file, or both). So rather than having a function `update`, which obviously many packages or parts of your program would want to claim for themselves, you will have e.g. the function `gnus-summary-update` (The `update` method in the package `gnus` and sub-package `summary`). With namespaces, you might have a function such as `clojure.string/update` (Package `clojure`, module or namespace `string`, method `update` (Nevermind that you would rarely see an `update` method in Clojure for reasons we will get to)).

Note the similarities in syntax between e.g. `clojure.string/update` and `Java.lang.String::update()`.

"Oh", you may say, "but the encapsulation means that we only put the methods that directly operate on the data in a class into that specific class. No-one else can or should have access to the data". To which a functional programmer would respond that the reason *why* you need to guard the internal state of an object so jealously is *because it can change*. If anyone can change the data (i.e. the state of an object), then you need to be able to enforce rules as to *how* the data is changed. What if you weren't able to change the state? We'll get back to that thought.

This *information expert* rule (as Craig Larman would call it), for all its good intentions, can actually lead to some unintended consequences down the line. The intention is to slim down an object's interface and to make it easier to read and understand exactly which methods depend on, and can modify a particular piece of data. But it also assumes that a piece of data has a limited and rather static way in which it can be used. I haven't done or looked for any studies on this, but I would not be surprised to see that over the lifetime of an object you have several distinct phases where different subsets of its methods are being used. You would, for example, have the construction and data population phase where you use a lot of setters. Then you will probably have a phase where the object is connected to other objects but the actual data is more or less set, and then you have a read-phase where data is mostly fetched. Let's say that this object is moved to another place in a data processing pipeline. Suddenly you may need a different set of methods to work with the same data. In summary, as the life of the object progresses you want to do different things with the data it contains, and so your public API of the class grows.

The alternative is that you instead *shrink* the public API to only getters and setters and instead create new classes that encapsulate *work tasks* suitable for different places in your data processing pipeline.

And in the end, you realise that most of your work with the object is done via `get` and `set` methods for each little datum contained in the object. Essentially you've created a whole abstraction layer between yourself and your data.

2.2 Inheritance

Inheritance is interesting. As far as I can see, inheritance is mostly a hack to facilitate object orientation in statically typed languages. Inheritance is intrinsically tied to classes and the need to have a `class` that defines the structure of data and the API for working with that data. For every piece of actual data, you create an *instance*, an `object,` based on the class definition. And you do all this because you need to tell the compiler that "this piece of memory over here, I want it to be interpreted in *this* particular way" (That's actually all you do when you create an object and store a pointer to the object in a pointer with type information, e.g. with `MyClass anObject = new MyClass()` and the type information that `anObject` is of the type `MyClass` is only (mostly) used at compile time).

Inheritance, then. Assume I want to tell the compiler that "this function should work the same way regardless of what data I give it, as long as it has the following minimum set of datum". In strongly typed languages I can achieve this in two (or nearly three) ways. I can lift out the relevant data (the minimum set of datum) into a *new* object and pass that into the function, then take whatever the function returns and put that back into my "real" object again. Assuming

that the function returns all the performed changes. If I am only going to need one or two pieces of the data, I might just give them as parameters to my function (this is the "nearly three" version). But this implies that I deliberatly break the oh-so-valued encapsulation and information expert rules just to be able to have a generic function.

The better solution is to use inheritance. I create a base class with the minimum set of datum, and then I extend this base class with additional data as sub-classes. Now, my function takes an object as a parameter, and this object ostensibly is of the base class type (whereas in reality it may be any one of the sub-classes and thus it may contain *more* data than the minimum set defined in the base class). Now my function is able to work in a generic way regardless of how much additional data I throw at it; it has declared that all it cares about is the data defined in the base class, and as long as the object you throw in there *inherits* from this base clase you're fine.

But in a *weakly typed* language, all of this becomes unnecessary. If a function require a particular datum in order to do what it is supposed to do, it will try to read that datum from the object passed to it. If it is there, fine. If it isn't, the function gets a `nil` value instead and deals with this. In a good programming language, `nil` can be used just as any other piece of information without throwing any errors.

So when do we want inheritance? When the *data* suggests that there is an inheritance relation. Not when the compiler demands it.

2.3 Polymorphism

In many languages, polymorphism has several different meanings.

1. functions with the same name but with different number of parameters
2. functions with the same name but where the types of the parameters differ
3. functions with the same name and parameter list, but belonging to different classes in the same inheritance hierarchy.

In Clojure, case (1) is referred to as *arity*:

```
(defn a-multi-arity-function
  ([one] (println "I got one parameter (an arity of 1)")
   [one two] (println "I got two parameters (an arity of 2)")
   [one two three] (println "this is getting ridiculous (with an arity of 3)")))
```

Case (2) is moot since Clojure is weakly typed. If I treat the parameter as of a specific type (e.g. a list or a vector) and the parameter either is not of that type or the Clojure reader don't know how to convert it to that type, I get a runtime error.

Case (3) is again a place where object orientation trips itself up. Since behaviour is tied to objects and classes, then obviously the only way of dealing with polymorphism is by tying it to the class. A class defines an interface (i.e. the names and parameter lists of functions), and thus if you want to have different implementations that to the rest of the system appears to implement the same interface you must *inherit* this interface (or base class) and rewrite its

methods. As with regular inheritance you can now pass around an object and claim to the compiler that it is of the base class type whereas it is in reality a sub-class. When a polymorph method is called on the object, the method tied to the actual type of the object is called.

But this is a design decision of object oriented languages. Clojure has a different idea. Here, you have *multi-methods*. And with a multi-method you get to define yourself *what part of your data* you want to use to decice which method to call. As in object oriented languages I *could* do this with an enormous switch-case statement instead of using the bultin ideas of polymorphism. But it is much nicer to let the programming language deal with this.

It is often easy to use `:keywords` in Clojure to do tasks like this. But I don't have to. I can use strings, literals, or anything really. If I do decide to use `:keywords`, I can also bring back the idea of inheritance to define relationships between my data types.

```
(derive :conjurer :wizard)
(derive :mage :wizard)
(derive :knight :swordsperson)
(derive :pirate :swordsperson)
(derive :pirate-in-training :pirate)

(isa? :conjurer :wizard) ; ==> true
(isa? :conjurer :swordsperson) ; ==> false

(parents :conjurer) ; ==> #{:user/wizard}
(descendants :wizard) ; ==> #{:user/conjurer :user/mage}

(defmulti fight
  "This is the dispatch-function that decides
  which method to call in order to fight.
  It runs the following code on the first argument,
  and whatever is returned is used. :keywords are
  functions that return their value, so if I give
  this method a key-value-map I can just call the
  right :keyword to get the value of that key-value
  pair. Think of this as a field-name and value in
  an OO class."
  :player-type)

(defmethod fight :wizard [player]
  (comment figure out how to throw a spell))

(defmethod fight :swordsperson [player]
  (comment and here we can try to look scary with a sword))

(defmethod fight :default [player]
  (comment just run away and hide somewhere))

;; And now, let's call the fight method
(fight {:name "Guybrush Threepwood"
```

```
    :player-type :pirate-in-training
    :inventory #{}))
```

Figure 1: Inheritance in clojure: it's all about the domain model.

... and the result of this will be that the "look scary with a sword" method will be called.

2.4 OO Lisp?

In the context of object oriented programming, lisp and Clojure is a particularly interesting beast to study. There is nothing inherently difficult in implementing OO programming in lisp. In Emacs-lisp (or the therein borrowed parts from common-lisp), you have constructs such as `defclass` and `cl-defmethod` for this. But really, these are just macros. Because you don't need much more than that. And there is no (significant) semantic difference between the following two code snippets.

```
MyClass anObject = new MyClass();
anObject.doSomething()
```

```
(let ((an-object (my-class)))
  (do-something an-object))
```

3 Clojure and Functional Programming

Just as the names imply, in *object oriented programming* the main programming construct is (supposed to be) the *object* and in *functional programming* the main construct is the *function*. As it happens, this actually means that your data structures get a much more prominent place in the code. And, importantly, not "just" the data structures, but the data structures *from the problem domain*. Freed of the need to create elaborate inheritance structures and pointer-relations between different datum just in order to convince the compiler to do what you want, you can instead focus on the relations that actually have a meaning in your problem.

Again, I'm not really interested in the origins or the dogma of functional programming; I'm mostly interested in getting started with Clojure. Tom Hickey has a nice rationale for why he created Clojure where he mentions some of this FP-dogma. Three things stand out to me, and many of the other benefits (like concurrency-safe programming) are results of this.

- Clojure is *homoiconic*. Code is data, and data is code. There is no different syntax for describing a data structure or a function. Indeed, I can treat a function as data (and modify it if I will) until I want to execute it as code. Lisp has this same quality, and it makes it easy to work with e.g. lambda functions, or attaching a particular version of a function to a particular object (should I need to).

- Data is *immutable*. This is a really important decision. I can't reassign the value of a datum. I can only create new data and reassign my reference (my variable) to this new data. This particular characteristic of Clojure

will by far impact the way you program with it the most. Especially if you come from an object oriented background where almost everything you do is about mutating the state of objects.

- Clojure encourages *Pure functions*. A pure function – like in mathematics – only operate with its input parameters and have no side effects outside the function. This is broken ever so often (e.g. a `println` has an obvious side effect that something is printed somewhere, and you have variables in your namespace that hold some static data (case in point: a function is a variable where the value is the code of the function, so when you call another function you are in fact reaching beyond your current scope) but the point is that you do not *change* anything outside of the function), but naming conventions suggest that you clearly identify any such digressions with an exclamation mark: `(defn here-be-side-effects! []).`

4 Immutable data: Riding the Data Train

Come in here, dear boy, have a cigar
You're gonna go far, you're gonna fly high
You're never gonna die
You're gonna make it if you try
They're gonna love you

Well, I've always had a deep respect
And I mean that most sincerely
The band is just fantastic
That is really what I think
Oh by the way, which one's Pink?

And did we tell you the name of the game, boy?
We call it Riding the Gravy Train

`s/Gravy/Data/g`

Ok, I just wanted to gratuitously throw that in there. *Immutable data* means that once you have defined a datum – or a piece of information – you cannot change it. The motivation I've heard for this is that "the world just doesn't work that way!" Well... It sort of does. Let's say I have a wallet full of money and I add some more money into it, I don't create a new wallet; I update the state of my existing wallet. Now, the 100kr note I just put in there has a static value that I cannot change, and so the argument is that most information is static.

I suspect that, mostly, immutable data means that you can take some really nice shortcuts when designing a programming language. There is only one assignment operator `(def ...)`, and multi-threaded code is dead easy to write because you do not have to worry about race conditions, mutexes, or deadlocks. So as a programming paradigm it's not that bad. But there is no need to get religious about it.

Mind you, there are ways of having state in a Clojure application; you have `refs` , `transactions`, `agents`, and `atoms`. But it's not as easy as `x=x+1`. (Side note: it is an interesting design choice in itself that assignment is one character

=, and test for equality is two characters ==. Pascal did it the other way arround, with := for assignment and = for equality test).

What this means in practice is that at least I had to re-think how I write even the small functions. Instead of a pattern of "First do `x` and store the result in a local variable. Then, use this local variable to do `y` and store the result of this in a (another?) local variable. When all is done, return the last local variable)". In emacs-lisp, you sometimes see this pattern where all the computation is done in a `let*` statement, and all that is left for the body of the function to do is to return the last value:

```
(defun do-something (param)
  (let* ((first-step (do-step-1 param))
         (second-step (do-step-2 first-step))
         (debug (debug-print second-step))
         (third-step (do-step-3 second step)))
    third-step))
```

This could, of course be rewritten to:

```
(defun do-something (param)
  (do-step-3 (debug-print-and-pass-through (do-step-2 (do-step-1 param)))))
```

But this is messier to read, and not as easy to add new steps to. And removing the debug-printout is a bit of a hassle. In Clojure, you can use the threading macro to tidy this up:

```
(defn do-something [param]
  (-> param
      do-step-1
      do-step-2
      debug-print-and-pass-through
      do-step-3))
```

Personally, I still like to write a first draft of my function in the `let*` way since it allows me to try out each step, give a name to what I expect to get in return, and debug-print wherever I think I might have a problem. Then I try to re-write it in a more Clojure-y form.

But with the threading macro we see the first glimpse of the "data train" (Clojure people talk about a "conveyor belt", but since I wanted to fit this section with the song lyrics at the start I'm going to stick with "data train"). Each step is a filter where you *transform* the data in some way. And then it is relatively easy to view the transformed data as *new* information, a new datum, and that the data pre-transformation remains unchanged.

The limitation of the data train, which in itself is a combination of immutable data and pure functions is that *anything that **can** change has to be passed along in the train*. So for any non-trivial application you are going to have different pieces of program state that is likely to change, and you need to find a nice package for this code such that you can (a) get the most recent version as input to your first processing step, and (b) store whatever transformations are done throughout the entire processing chain.

I'm not sure whether it is the data that rides the train or whether it is the functions. And sometimes you feel like you are hanging on to the side of

the train while it charges along spawning new versions of mutating data and shedding old versions along the ride. Maybe I would have been better off with "Atom Heart Mother" and its basic input-output album cover [1] as an analogy. But misery loves company, so I *wish you were here*.

5 It's All in the Data Structure

Consider the following pattern. You have a list of strings (for example input from a user) and a list of objects. In each object you have a list of strings (e.g. alternative names for the object). For each input string, you want to find all objects where at least one of the names match. This is a three-way map, and if this isn't enough to give you a headache then I don't know what will.

```
(defn input->objects [input-strings all-objects]
  (map #(objects-that-match % all-objects) input-strings))

(defn objects-that-match [match-string all-objects]
  (map #(string-in-names match-string (:names %)) all-objects))

(defn string-in-names [match-string names]
  (map #(= match-string %) names))
```

I need to tidy this up a bit since a `map` is not the best function to use here:

```
(defn input->objects [input-strings all-objects]
  (filter #(match-object input-strings (:names %)) all-objects))

(defn match-object [input-strings object-names]
  (some #(string-in-names % object-names) input-strings))

(defn string-in-names [match-string names]
  (some #(= match-string %) names))
```

But this is just as ugly. The problem here is that I wasn't clear enough on my data structures. In defiance to all my lisp heritage I should *not* have used a list. Clojure suggests that you should rarely use lists for data anyway, so maybe I should have used a vector? No, that would give me the same problem. So what remains? Shirley not a hash-map? No. But what about a set?

A set has some nice properties, and there are some nice functions already implemented in `clojure.set` that I may be able to use... First off, I don't care if the same word is repeated in the input; it's going to match to the same object anyway. Likewise with the names of the objects. So both of these can just as well be represented as sets. That only leaves me with the vector of objects. And what do I really want to do with my set of input strings and my set of names? I want to find all objects where the *intersection* of input and names is not empty. So:

```
(defn input->objects [input-strings all-objects]
  (filter #(seq (clojure.set/intersection input-strings (:names %))) all-objects))
;; the Clojure documentation says to prefer (seq x) over (not (empty? x))
```

[1] I.e. a cow, aptly identified by Terry Pratchett as being your basic input-output processor.

Not having mutable variables forces you to think in terms of lambda functions that operates on each and every element in a collection. And once you do that, you will take a closer look at the data structures themselves and what you are really trying to achieve. And then it's a short hop over to good old fashioned discrete mathematics. *:notes: riding the data train :notes:*

6 The God Object

Let's say I want to implement an old-fashioned adventure game. Doing a bit of domain modelling, an adventure game has:

- Scenes
- Characters
- Items
- Current Location
- Player Inventory

As for relations:

- A scene may contain characters and items.
- Characters may hold items.
- The player inventory may contain items.
- Items may contain other items.

And, to further complicate matters, each `#{scene, character, item}` has a list of *permitted actions* and a list of *denied actions*, and these lists may vary depending on what happens in the game.

From a Clojure perspective, the problem with this setup is that this is not immutable information. A player may enter a scene, pick up an item, enter another scene, and give the item to a character. Thus, the game has a *state* comprised of the individual states of each scene, character, and item (and player, but let's restrict ourselves – for now – to only have a single player).

The *Game loop* is the top-level implementation of the data train for an adventure game:

1. Print a prompt based on the `current state`
2. Read input
3. Parse input and *act on input*
4. Store result of action in `updated state`
5. Recurse, using `updated state` as the current state.

So far, so good. We're still aligned with the data train idea. But what do we meen by *act on input*? And how do we do it? Well, what *can* we do? We call a function. And this function calls another function. And anywhere where we need to work with a scene, item, or a character, we need access to the state, and so we need to pass this along as a parameter. And whenever we need to change the state we return this updated state as a new object. And eventually these new objects get merged into the `updated state` in the game loop.

Can we use global variables for this? Let's say one variable for each scene, another set of variables for each character and item? No, because there is no re-assignment operator. We could of course use `def`, or choose another data format so that we can use `alter` or `swap!`, but that's not really playing nice with the immutable data precept. And that would mean that we definitely and finitely lock us into a single-player game. We *want* to allow each running instance of the game loop to have its own state. And so we create a super-object where we keep the state for all the objects (scenes, characters, items) that this player have access to. And, don't forget, we also need to include the relations between our objects as well. For example, if the item "Kettle" is available in the scene "Pottery", we need to have a reference (at least in one direction) between these two objects. So our super-object need to have support for *pointers*.

So I have all my stateful objects and their relations all bundled up and contained in a super-object that contains *everything*, and needs to be passed around *everywhere* in order to make things work.

... and what do we call that? A God Object. And what do we think of those?

7 Design Patterns

Most (if not all) design patterns in the GoF book are, when you think about it, tricks to circumvent limitations in the programming language. Since many of them are derived from the *strategy pattern*, let's take a look at this pattern.

The problem that *Strategy pattern* addesses is that you want to isolate the execution of a particular algorithm from the rest of the system so that you can replace one algorithm with another (and I use the term algorithm loosely). You may for example have one algorithm to *look* at an item in an adventure game, and another algorithm to *take* said item. *Look* and *take* are examples of *strategies*.

Once you have decided which strategy that is the most suitable in the current situation, you want to have *one place* in your code that "picks" the right implementation and returns it for you. And wherever you want to use the strategy, you call the same method(s) to execute the strategy. So every strategy implements at least a method `execute()`. You may also have `setup()` and `teardown()` methods, and other methods too since there is no limit to what may be part of a single strategy. The point is that once the decision of which strategy to use has been made, no-one needs to worry about it anymore. From "your" side you perform "your" task in the same way and invoke the currently instantiated strategy with the same method calls, letting the compiler worry about which strategy-implementation to actually route your calls to (with a little help from polymorphism).

In C/C++ I can implement this by having a set of function pointers, one

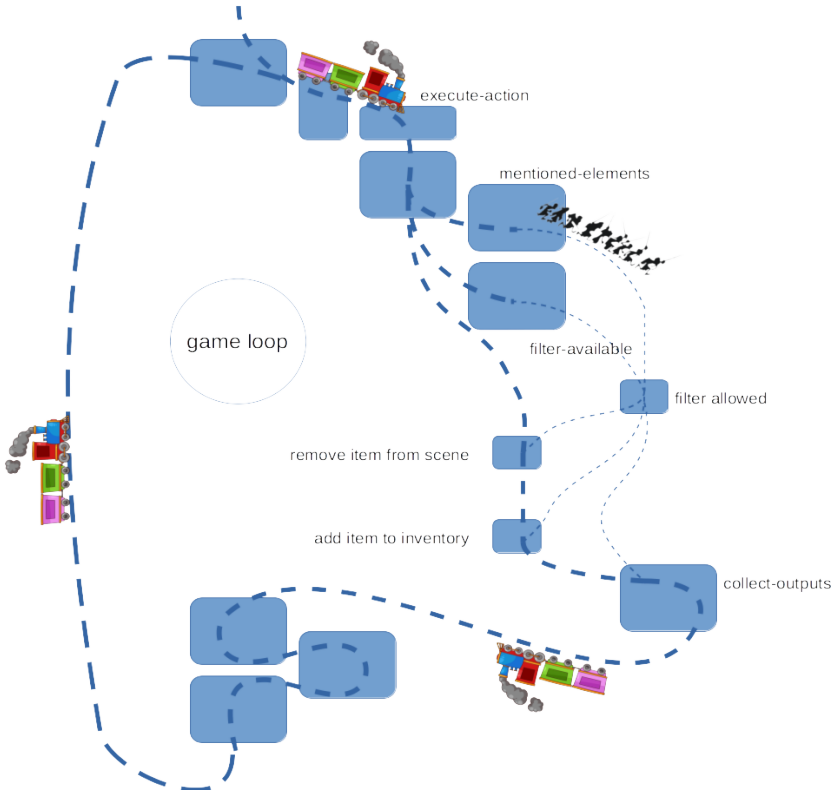
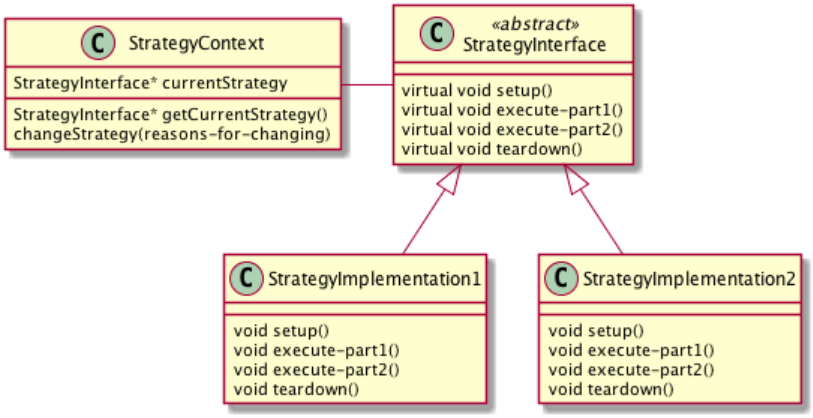


Figure 2: Riding the game-state data train for the operation “Take” in an aventure game

for each method in my strategy interface. When I select a new strategy I replace all my function pointers to point to the strategy's implementation of the corresponding method. But function pointers are a PiTA. It is much easier for me to let the compiler deal with these and fortunately that is exactly what I get if I put all my methods into a class (and in C++ I have to do additional trickery by declaring the methods `virtual`, because ... well, just because!). Now, when I have an object I have a built-in function pointers to all the right method implementations for that particular object type. And this is the solution used in the Strategy design pattern:

```
class StrategyContext {
 StrategyInterface* currentStrategy
 StrategyInterface* getCurrentStrategy()
 changeStrategy(reasons-for-changing)
}

StrategyContext - StrategyInterface

class StrategyInterface <<abstract>> {
 virtual void setup()
 virtual void execute-part1()
 virtual void execute-part2()
 virtual void teardown()
}

class StrategyImplementation1 {
 void setup()
 void execute-part1()
 void execute-part2()
 void teardown()
}

class StrategyImplementation2 {
 void setup()
 void execute-part1()
 void execute-part2()
 void teardown()
}

StrategyInterface <|-- StrategyImplementation1
StrategyInterface <|-- StrategyImplementation2
```



For completeness sake, we also introduce the *context* class that is responsible for creating an instance of the current strategy and storing a pointer to this instance so that the rest of the system can find it when needed.

In languages such as Javascript some of this falls away. I no longer need the abstract base class; I can just call the methods on my concrete implementation. If the method is implemented, then fine. If it isn't, I'll get a runtime exception. Hence, I no longer really need the context class either. Or rather, the context is merged with the current strategy. Mind you, defining classes and their relationships according to the Strategy pattern still makes for much nicer and more readable code. So at least we get to use design patterns for something more meaningful than to fool the compiler: *we enhance readability of the code.*

But what about Clojure? Objects do not really have methods in Clojure. ... Unless we go to town with the homoiconicity and throw in code in e.g. a Hashmap:

```
(def an-object {:name "something" :behaviour (fn [] (str "la-di-do-dah"))})
(:name an-object) ; ==> "something"
(:a-key-that-does-not-exist an-object) ; ==> nil
(:behaviour an-object) ; ==> #function[first-app.core/fn--6749]
((:behaviour an-object)) ; ==> "la-di-do-dah"
```

In Clojure, the key programming concept is a function, and so it make sense that if we are going to implement a strategy pattern, we are going to do it using functions. Data is data, and behaviour is behaviour, and Clojure does a good job of permitting us to keep these two separate. We do not have some objects just to encode some behaviour and other objects to represent data. In fact, the solution has already been presented as how Clojure does 2.3 and the 1 example. Because what we really need to implement this (and most other) design pattern is polymorphism.

7.1 Strategy pattern in Clojure

Let's break down the 1 - example from previously. We have an object that represent a player:

```
(def a-player {:name "Guybrush Threepwood"
```

```
                :player-type ::pirate-in-training
                :inventory #{}})
```

We may of course add a whole bunch of other attributes to the player object as well. We may even decide that we want a player `constructor` to make sure that we get all the relevant attributes under the right names. This constructor is, of course, just an ordinary function because why would we need more?

We specify that the `:player-type` is `:pirate-in-training`, and if you remember we had a whole elaborate scheme of different player types:

```
(derive ::conjurer ::wizard)
(derive ::mage ::wizard)
(derive ::knight ::swordsperson)
(derive ::pirate ::swordsperson)
(derive ::pirate-in-training ::pirate)
```

There is no "root node" here, i.e. we do not have some abstract ":generic-player" that all the other player types "inherit" from, because we do not need it.

And now for the polymorphism:

```
(defmulti fight
      :player-type)

    (defmethod fight ::wizard [player]
      (str "figure out how to throw a spell"))

    (defmethod fight ::swordsperson [player]
      (str "and here we can try to look scary with a sword"))

    (defmethod fight :default [player]
      (str "just run away and hide somewhere" player))
```

... And that's really all we need. The most important thing with all of this is that there is no "we need to fool the compiler and the type system into doing what we want"-cruft here. We have expressed exactly what the domain looks like and behaves, using only domain concepts, and nothing else.

We may want other methods connected to our strategy pattern. A wizard may, for example `speak` more eloquently than a pirate (arr!). And they may `look` and see the world differently. This would be addressed by just adding more multi-methods; one set for `speak` and one set for `look`.

So far, we have not strayed too far from the GoF definition of the strategy pattern, we are just using a different flavour of syntactic sugar. Rather than having the classes `wizard-strategy` and `swordsperson-strategy` , each containing the methods `#{fight speak look}`, we have the multi-methods `fight`, `speak`, and `look`, each of which has one instance for `:wizard`, and another for `:swordsperson`.

The beauty comes when you realise that for *some* methods, this hierarchy is all wrong. Sometimes, the `pirate-in-training` acts differently to a `knight` (running, for example is easier when you are not lugging around 20kg of steel

barely bent around your limbs). In C++ we would achieve this e.g. by subclassing the `swordperson-strategy` class and overloading the affected methods while leaving the rest untouched. In Clojure, we just add another instance of the multi-method classified to the `:knight` or `:pirate-in-training` instead. ... And then we tell Clojure which to prefer.

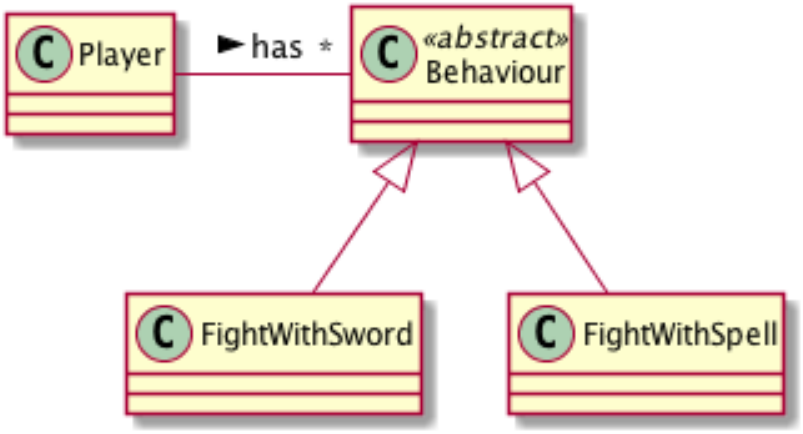
The net result is, again, that we are able to focus much more on the *problem domain* and the *domain concepts*, with considerably fewer lines of code written for the sake of fooling the compiler.

7.2 Composition or Inheritance

Suppose our hero Guybrush spends some time learning a magic spell. Should he then fight like a swordsperson or as a wizard? The lesson learned from object oriented languages is that in order to accomodate this, we should avoid having deep type-hierarchies using inheritance, and instead use composition. *Favour composition over inheritance* is the new battle cry.

Entity Component Systems is the design pattern that encompasses this principle:

```
class Player
class Behaviour <<abstract>>

Player - "*" Behaviour : has >

Behaviour <|-- FightWithSword
Behaviour <|-- FightWithSpell
```



And so we are able to focus on the behaviour that a player has rather than sticking a label onto them. We soon realise that a Knight fights in a different way with his broadsword, and so we can simply add a new subclass `FightWithBroadsword`. If we can re-use stuff from `FightWithSword` we'll inherit from there. But if we can't, we can just as well inherit directly from `Behaviour`. From the `Player`'s perspective, it's all the same.

How would we do this in Clojure? We can either stick with player-type or switch over to defining behaviour as outlined here. The difference in the

player object is that rather than having a single value for `:player-type` or `:player-skills` we have a set of values `#{::pirate-in-training ::conjurer}`. Redundantly, the example below defines both. We then leave it up to the code in the dispatch-function to decide which behaviour to use.

```
(def a-better-player {:name "Guybrush Threepwood"
                      :preferred-role ::pirate-in-training
                      :player-type #{::pirate-in-training ::conjurer}
                      :player-skills #{:swordfighting :spellcasting}
                      :inventory #{}})

(defmulti better-fight
  (fn [player]
    (cond
      (and (isa? (:preferred-role player) ::wizard)
           (contains? (:player-skills player) :spellcasting)) :spell
      (or (isa? (:preferred-role player) ::swordperson)
           (contains? (:player-skills player) :swordfighting)) :sword)))

(defmethod better-fight :spell [player]
  (str "figure out how to throw a spell"))

(defmethod better-fight :sword [player]
  (str "we can try to look scary with a sword"))

(defmethod better-fight :default [player]
  (str "just run away and hide somewhere"))

(better-fight a-better-player)
```

As you see, the dispatch function is now more complex, since we want to decide both whether we have the *desire* and the *skill* to fight in a certain way. The next step would be to throw in a `:currently-fighting` field and then we could decide whether we should use our sword or a spell depending on whom the player is fighting. And pretty soon we realise that there is a whole system of rules to decide how to fight each enemy given the current skillset and desires, so we will probably encode all this into a set of functions that the dispatch method can make use of. The point is, we did not have to invent a whole new inheritance hierarchy or change the relations betwen our objects to get there. We just wrote functions to act on the data given in the object.

8 GRASP Patterns

The GRASP patterns that Craig Larman introduces are not new; they have been around for ages under other names. Larman's contribution is to create a curated list of essential patterns and give them more meaningful names (I mean "Law of Demeter"? Who the F is Demeter and why is she messing with my source code?).

Let's take a look at the GRASP patterns. Without structure, they are:

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

And if we try to organise them a bit, we get:

8.1 High Cohesion

You achieve high cohesion by making sure that you assign *information expert* responsibilities properly and with only a few responsibilities per class. High cohesion is often seen as the flip side of low coupling, but it does not have to be.

Related patterns are *Information Expert*, *Controller*, *Creator*, and *Pure Fabrication*.

Information Expert The responsibility for *doing* something should be placed in the same class as where the *knowledge* is. And the responsibility for *knowing* something should be placed in the class that is in the best position to know it. Yes, this is vague and circular. Fingerspitsgefühl is essential to figure out which class this is. Do note that there are different types of information, for example *domain knowledge, /work flow*, *source code structure*.

A **controller** is essentially an *information expert* on a particular **work flow** and the **source code structure** so that it can delegate and see the work done.

Likewise, a **creator** is an *information expert* on the **source code structure**

Pure Fabrication When there is no obvious candidate for where to assign a responsibility, this pattern gives us a licence to invent.

8.2 Low Coupling

When we have done everything we can to have as *high cohesion* as possible, we should also strive to have as loosely coupled a system as possible. This makes the system more readable, changes become more localised, it is easier to get an overview of the side effects of an operation, and overall the system becomes more maintainable.

Related patterns are *Indirection*, and *Protected Variations*.

Indirection One way to get a more loosely coupled system is to avoid direct pointers between classes. If nothing else, a direct pointer requires us to have knowledge of (a) where to find the other object, and (b) what methods it makes available (i.e., source code structure). By using an intermediary we reduce this

to (a) knowlegde about how to access the general mechanism for *finding* the right object, and (b) the (hopefully) interface for the *domain concept*.

Protected Variations is a special form of indirection where we hide knowledge about the source code structure in an adapter class that exposes an interface for the domain conceot.

8.3 Polymorphism

Polymorphism is more of a tool used to achieve e.g. information experts (e.g. controllers or information experts for a particular strategy in the strategy pattern). I've discussed this tool at length already.

8.4 GRASP and Clojure

High cohesion is still important. Each object should have as high cohesion as possible, and this means that you need to think about what responsibilities it should have, and what each object is an **information expert** on.

For the responsibility of being a **Controller** or a **Creator** it is no longer necessary to have a class, since this should now be a function. Thus, we no longer have to worry about whether we should allow this controller class to also have other responsibilities. If we're good, we may even delegate to the compiler/reader with the help of polymorphism and multi-methods so our source code can be freed of "getting-the-compiler-to-work" cruft and focus more on implementing the domain functionality.

Information expert is now mostly about *domain knowledge*. Objects do not have anything to do with the *work flow* or the *source code structure* anymore.

Pure fabrication is moot in terms of creating objects. If there is no corresponding information container in the real world, then why should there be one in the software program? *Containers!* you may answer, but for that we have the excellent builtin data types with a standardised API.

Low Coupling is also still important, but we get it for free. Because our objects are now only representations of real world information, we have already as few couplings as we can. And because we (probably) implement these couplings e.g. with the help of `:keywords`, we are already loosely coupled.

Of course, we can and do still couple our *code* by letting functions call other functions. But if we call functions from another namespace we need to jump through a few hoops to get there, and so we will likely avoid this if we can. Using functions within our own namespace is equivalent to using methods in the same class/on the same object (not quite but close enough) and is actually even a measure of high cohesion.

9 FOOP

Is object oriented programming dead? Is functional programming the past, present, and future? Should we all learn lisp?

I think yes and no, yes, and absolutely.

An old saying in creative writing is "end with a bang, not a whimper". I'm going to end with FOOP [2]. Here's why.

[2] Functional Object Oriented Programming

9.1 State matters

It is not so easy as the functional programming world would have you believe to just write pure functions working on immutable data and never change any state. The thing we can pick up from functional programming is, however, that we do not need to change states in nearly as many places as we think in the code. We can and should try to limit and clearly mark the places where we actually store a modified state. This makes for more readable and more maintainable code since you no longer need to expect that someone else rewires the rug from under your feet.

But in the end, most applications rely on state changes in some form or other. But maybe it does not have to be as spread throughout the code base as it commonly is when working in the object oriented programming paradigm. And when we think about which state that changes *for one particular user*, it is even less clear-cut. When you think about it, most transactions are built up as:

1. Find the root object for the current user (even if there is only one user, we still look for a root object)
2. Traverse collections of objects that the root object points to (may need to traverse several levels down) until we find the object where the state should change
3. Change the state on that object
4. return all the way up

And, importantly, most objects in this hierarchy probably only ever have one single pointer to them, the one that ultimately originates from the root object for that one particular user. So instead of returning "empty-handed", we may just as well propagate the state change upwards until the top level, where we add a new step *5. update current state*.

The downside is that you have one object that is the super-set of all objects that could possibly change for one particular user. In the adventure game, this super-set consists of at least all scenes, items,and characters the user has modified. After playing a while, and in a reasonably large game world, your garbage collector is going to have to run at full throttle just to keep up every time you touch an item. At this stage, you are probably better off storing game state in a database. But do you need to do this at the leaf-nodes of your programme, or can you do this higher up in the call stack?

9.2 Modularisation

Objects are still going to be used, simply because there is no easier way to pass around a record consisting of several attributes. An object is a container of structured data, and that won't change. The change is whether we have an enclosed space in our code where operations on this data should be defined. In some sense, it is still a good idea in terms of code structure, but as I pointed out when I discussed 2.1, this implies that we have to bloat our class with every method needed throuhgout an object's entire lifecycle. Structuring the code

around the *work tasks* and passing around an immutable object perhaps better represent the way an object is being used.

Classes *can* be used to define the objects, but need not be. Instead of predefined classes and types, Clojure uses duck-typing: "If it walks like a duck and it quacks like a duck, then it must be a duck".

Writing modular software has always been a goal to improve readability. Even when programming Basic the code was separated into blocks of line numbers (lines 1000–1190 does X, lines 2000–2070 does Y, etc). Classes are often used to get modular code when we think that a package is too much. But with packages and namespaces and whatnot, is there still a need to abuse classes for this?

Modular code $\neq$ Class-oriented code. And objects are still needed.

9.3 The Need for Modelling

All of this does not mean that you should skip your software design and go directly to adding things to objects as you need them. If anything, Duck-typing requires *more* of an up-front agreement among the entire development team about what constitutes a duck in order to write maintainable code. Your *domain information model* becomes much more important, i.e. what domain concepts are there, what information represent each domain concept, and how are the domain concepts related to each other.

In software architecture it is stressed how you need to view a system from several different viewpoints; there's the *conceptual view* that describes how the problem domain works, and there's the *module view* that describes the software modules you are going to build in order to represent the concepts from the conceptual view, and there's the *execution view* that describes how you plan on deploying your software modules onto your available hardware platform. The *domain information model* corresponds to the *conceptual view* from the software architecture world, with the added twist that you can represent it more or less straight off as code as well.

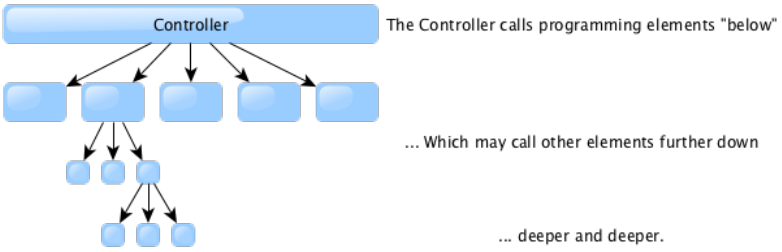
But since this type of objects do not have any functionality, you are going to have to take a look at your *module view* as well. In this view you will be looking at which software packages you need and which namespaces you are going to have, and then which functions and object generators you are going to put in each namespace/package/module. You may, for example, structure your code around which *domain concepts* they work with, or you may structure it around the *work tasks* that you do. Or both.

9.4 The Rebirth of Design Patterns

The GRASP patterns are on so low a level that we are always going to have to deal with them. The fact that we have seen them survive several programming paradigm shifts is a testament to this. For example, just as "information expert" can be interpreted in the context of *objects*, it can also apply to *functions*. And, as argued above, both of these are still needed. The same goes for the meta-GRASP patterns low coupling and high cohesion: they serve just as important a purpose for functions as they do for objects.

A reflection on the *controller* pattern. In OO programming, you place a controller at some level to direct the workflow. This is (mostly) done using

functions, classes, and objects one level below. Anywhere in this call stack there may be state changes and side effects, which the top-level controller simply has no idea about. Debugging a program like this can quickly become less than entertaining.



The raison d'être for the controller was probably to collect information and assemble a work order of some sort in order to dispatch one big side effect in the end. Latching on to this, and adding the "immutable data" and "pure functions" doctrines, the controller pattern is turned on its side. You may and probably will still have deep call hierarchies *but without the side effects*. And that makes all the difference in the world when it comes to debugging your application.

The continued relevance of the GoF patterns is less certain. The key quality attribute that most GoF design patterns address is *modifiability*. Interpreted broadly, this encompasses understandability of the code as well as ability to extend with new behaviours or correct/adjust the existing behaviour. The design patterns make use of two mechanisms for this, i.e. modularisation (in the guise of classes) and polymorphism.

When the programming language provides a different mechanism for polymorphism, such as Clojure does, the remaining purpose of the classic GoF design patterns is modularisation. And, as argued, maybe there are better ways to structure your code into modules than "everything dealing with this object type should go into one class".

But design patterns also contribute with a *common language*. By saying that a class is e.g. part of a Strategy pattern, you are also saying what other classes can be found in its vincinity, and how to extend with a new strategy. If a strategy encompasses more than one or two multi-methods, you are going to desire some way of enshrining this common vocabulary in the code. Don't worry, Clojure's got you covered: `defprotocol` y `defrecord` para rescatar! But then we are moving back towards wanting our *objects* to be created according to predefined templates, and having a fixed set of operations on them, and maybe we are going to need the good old fashioned design patterns after all. Although I strongly suspect that design patterns for duck-typed pure functions with immutable data is going to look very different to the GoF patterns.

9.5 End

Shirley I can't be the first to harbour thoughts on how to program object-oriented in Clojure, but there is surprisingly little available on the internet about it. So maybe there's a reason why nobody talks about FOOP.

But I hope I have arged at least semi-successfully that the amalgamation of object oriented programming and functional programming has already happened in the programming languages, and what remains now is for best practices to follow. Maybe there's a bachelor or master thesis or two here to be written to explore this further?