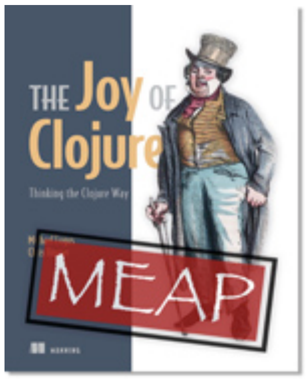
MANNING PUBLICATIONS

Exploring Clojure multimethods with the universal design pattern

An article from



[Joy of Clojure](#)
EARLY ACCESS EDITION

Thinking the Clojure Way
Michael Fogus and Chris Houser
MEAP Release: January 2010
Softbound print: November 2010 (est.) | 400 pages
ISBN: 9781935182641

This article is taken from the book Joy of Clojure. *The authors discuss using multimethods and the UDP to build abstractions.*

The most specific event can serve as a general example of a class of events.

– Douglas R. Hofstadter

In Douglas Hofstadter's Pulitzer-Prize-winning work *Godel, Escher, Bach: An Eternal Golden Braid* he describes a notion of the *Prototype Principle* that is the tendency of the human mind to use specific events as models for similar but different events or things. He presents the idea that there is "generality in the specific" (Hofstadter 1979). Building on this idea, programmer Steve Yegge coined the term the universal design pattern (UDP), extrapolating on Hofstadter's idea and presenting it in terms of prototypal inheritance.

The UDP is built on the notion of a map or map-like object. While not at all groundbreaking, the flexibility in the UDP derives from the fact that each map contains a reference to a *prototype* map used as a parent link to inherited fields.

Now, you might wonder how anyone could model a software problem in this way, but we assure you that countless programmers do just that every single day when they choose JavaScript. We will implement a subset of Yegge's universal design pattern and discuss how it might be used as the basis for abstraction-oriented programming and polymorphism using Clojure's multimethods and ad-hoc hierarchies.

The parts

In addition to the aforementioned prototype reference, the UDP requires a set of supporting functions to operate, namely: `beget`, `get`, `put`, `has?`, and `forget`.

The entire UDP is built on these five functions, but we will need the first three for this section.

beget

The `beget` function performs a very simple task; it takes a map and associates its prototype reference to another map, returning a new map:

```
(ns joy.udp
  (:refer-clojure :exclude [get]))

(defn beget [o p] (assoc o ::prototype p))
(beget {:sub 0} {:super 1})
;=> {:joy.udp/prototype {:super 1}, :sub 0}
```

To participate in the UDP, maps must have a `:joy.udp/prototype` entry.

put

The function `put` takes a key and an associated value and puts them into the supplied map, "overwriting" any existing key of the same name:

```
(defn put [m k v]
  (when m
    (assoc m k v)))
```

The `put` function is asymmetric to the functionality of `get`. That is, `get` retrieves values anywhere along the prototype chain, while `put` only ever inserts at the level of the supplied map.

get

Because of the presence of the prototype link, `get` requires more than a simple one-level lookup. Instead, whenever a value is not found in a given map, the prototype chain is followed until the end:

```
(defn get [m k]
  (when m
    (if (contains? m k)
      (m k)
      (recur (::prototype m) k))))

(get (beget {:sub 0} {:super 1})
     :super)
;=> 1
```

We do not explicitly handle the case of "removed" properties but, instead, treat it as any other associated value. This is fine because the "not found" value of `nil` acts like false. Note the somewhat uncommon use of `contains?`. Most of the time it's sufficient to rely on the fact that looking up a nonexistent key will return `nil`. However, in cases where you want to allow users of your functions to store any value at all, including `nil`, you'll have to be careful to distinguish `nil` from "not found", and `contains?` is the best way to do this.

Usage

Using only `beget`, `put`, and `get`, we can leverage the UDP in some simple yet powerful ways. Assume that, at birth, cats like dogs but only learn to despise them when goaded. Morris the cat has spent most of his life liking 9-Lives cat food and dogs, until the day comes when a surly Shih Tzu leaves him battered and bruised.

We can model this unfortunate story below:

```
(def cat {:likes-dogs true, :ocd-bathing true})
(def morris (beget {:likes-9lives true} cat))
(def post-traumatic-morris (beget {:likes-dogs nil} morris))
```

```
(get cat :likes-dogs)
;=> true

(get morris :likes-dogs)
;=> true

(get post-traumatic-morris :likes-dogs)
;=> nil
```

The map `post-traumatic-morris` is like the old `morris` in every way except for the fact that he has learned to hate dogs. Modeling cat and dog societal woes is, indeed, interesting but far from the only use case for the UDP, as you will see next.

No notion of self

Our implementation of the UDP contains no notion of "self-awareness" via an implicit `this` or `self` reference. While adding such a feature would probably be possible, we have intentionally excluded it in order to draw a clear separation between the prototypes and the functions that work on them. A better solution, and one that follows in line with a deeper Clojure philosophy, would be to access, use, and manipulate these prototypes using Clojure's multimethods.

Multimethods to the rescue

Adding behaviors to the UDP can be accomplished quite easily using Clojure's multimethod facilities. Multimethods provide a way to perform function polymorphism based on the result of an arbitrary dispatch function. Coupled with the UDP implementation above we can implement a prototypal object system with differential inheritance similar to, although not as elegant, as that in the Io language. First, we will need to define a multimethod `compiler` that dispatches on a key `:os`:

```
(defmulti compiler :os)
(defmethod compiler ::unix [m] (get m :c-compiler))
(defmethod compiler ::osx [m] (get m :c-compiler))
```

The multimethod `compiler` describes a simple scenario; if the function `compiler` is called with a prototype map, then the map is queried for an element `:os`, which has methods defined on the results for either `::unix` or `::osx`.

We'll create some prototype maps to exercise `compiler` below:

```
(def clone (partial beget {}))
(def unix {:os ::unix, :c-compiler "cc", :home "/home", :dev "/dev"})
(def osx (-> (clone unix) (put :os ::osx) (put :c-compiler "gcc") (put :home "/Users")))

(compiler unix)
;=> "cc"

(compiler osx)
;=> "gcc"
```

And that's all there is to creating behaviors that work against the specific type of a prototype map. However, a problem of inherited behaviors still persists. Since our implementation of the UDP separates state from behavior, there is seemingly no way to associate inherited behaviors. However, as we will now show, Clojure does indeed provide a way to define ad-hoc hierarchies that we can use to simulate inheritance within our model.

Ad-hoc hierarchies for inherited behaviors

Based on the layout of the `unix` and `osx` prototype maps, the property `:home` is overridden in `osx`. We could again duplicate the use of `get` within each method defined (as in `compiler`) but, instead, prefer to say that the lookup of `:home` should be a derived function:

```
(defmulti home :os)
(defmethod home ::unix [m] (get m :home))

(home unix)
;=> "/home"

(home osx)
; java.lang.IllegalArgumentException: No method in multimethod 'home' for dispatch value: :
```

Clojure allows you to define a relationship stating "::osx is-a ::unix" and have the derived function take over the lookup behavior using Clojure's `derive` function:

```
(derive ::osx ::unix)
```

And now the `home` function just works:

```
(home osx)
;=> "/Users"
```

You can query the derivation hierarchy using the functions `parents`, `ancestors`, `descendants`, and `isa?` as shown below:

```
(parents ::osx)
;=> #{:user/unix}

(ancestors ::osx)
;=> #{:user/unix}

(descendants ::unix)
;=> #{:user/osx}

(isa? ::osx ::unix)
;=> true
(isa? ::unix ::osx)
;=> false
```

The result of the `isa?` function defines how multimethods dispatch. In the absence of a derivation hierarchy, `isa?` can be likened to pure equality but with it traverses a derivation graph.

Resolving conflict in hierarchies

What if we interject another ancestor into the hierarchy for `::osx` and wish to again call the `home` method? Observe the following:

```
(derive ::osx ::bsd)
(defmethod home ::bsd [m] "/home")

(home osx)
; java.lang.IllegalArgumentException: Multiple methods in multimethod
; 'home' match dispatch value: :user/osx -> :user/unix and
; :user/bsd, and neither is preferred
```

As shown in figure 1, since `::osx` derives from both `::bsd` and `::unix`, there is no way to decide which method to dispatch because they are both at the same level in the derivation hierarchy. Thankfully, Clojure provides a way to assign favor to one method over another using the function `prefer-method`; seen below:

```
(prefer-method home ::unix ::bsd)
(home osx)
;=> "/Users"
```

In this case we used 'prefer-method' to explicitly state that for the multimethod 'home' we prefer the method associated with the dispatch value '::unix' over the one for '::bsd', as visualized in Figure 1. As you recall, the home method for ::unix explicitly used get to traverse the prototype chain, which is the preferred behavior.

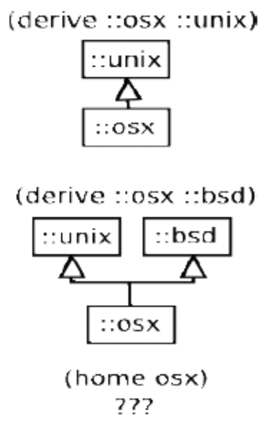


Figure 1 Hierarchy conflict: Most languages allowing type derivations use a built-in object-oriented programming conflict resolution strategy or, in the case of Common Lisp Object System (CLOS), fully customizable. However, Clojure requires conflicts to be resolved with the multimethod's prefer-method.

As you might expect, removing the `home` method for the `::bsd` dispatch value using `remove-method` will remove the preferential lookup for `::osx`:

```
(remove-method home ::bsd)
(home osx)
;=> "/Users"
```

All of the functions above manipulate and operate off the global hierarchy map directly. If you prefer to reduce these potentially confusing side-effects then you can define a derivation hierarchy using `make-hierarchy` and `derive`:

```
(derive (make-hierarchy) ::osx ::unix)
;=> {:parents {:user/osx #{:user/unix}},
     :ancestors {:user/osx #{:user/unix}},
     :descendants {:user/unix #{:user/osx}}}
```

Once you have a separate hierarchy in hand you can provide it to `defmulti` to specify the derivation context, thus preserving the global hierarchy map.

Arbitrary dispatch for true maximum power

Until now, we've only exercised multimethods using a single privileged `:os` property, but this does not accentuate their true power. Instead, multimethods are fully open and can dispatch on the result of an arbitrary function; even one that can pull apart and/or combine its inputs into any form:

```
(defmulti compile-cmd (juxt :os compiler))

(defmethod compile-cmd [::osx "gcc"] [m]
  (str "/usr/bin/" (get m :c-compiler)))

(defmethod compile-cmd :default [m]
  (str "Unsure where to locate " (get m :c-compiler)))
```

The dispatch values for the new `compile-cmd` methods are vectors composed of the results of looking up the `:os` key and calling the `compiler` function defined earlier. You can now observe what happens when `compile-cmd` is called:

```
(compile-cmd osx)
;=> "/usr/bin/gcc"

(compile-cmd unix)
;=> "Unsure where to locate cc"
```

The handy-dandy juxt function

The function is highly useful in defining multimethod `juxt` dispatch functions. In a nutshell, `juxt` takes a bunch of functions and composes them into a function returning a vector of its argument(s) applied to each given function, as shown below:

```
(def each-math (juxt + * - /))

(each-math 2 3)

;=> [5 6 -1 2/3]


((juxt take drop) 3 (range 9))

[(0 1 2) (3 4 5 6 7 8)]
```

Having a convenient and succinct way to build vectors of applied functions is very powerful in defining understandable multimethods—although that is not the limit of `juxt`'s usefulness.

Summary

Using multimethods and the UDP is an interesting way to build abstractions. Multimethods and ad-hoc hierarchies are open systems allowing for polymorphic dispatch based on arbitrary functions.