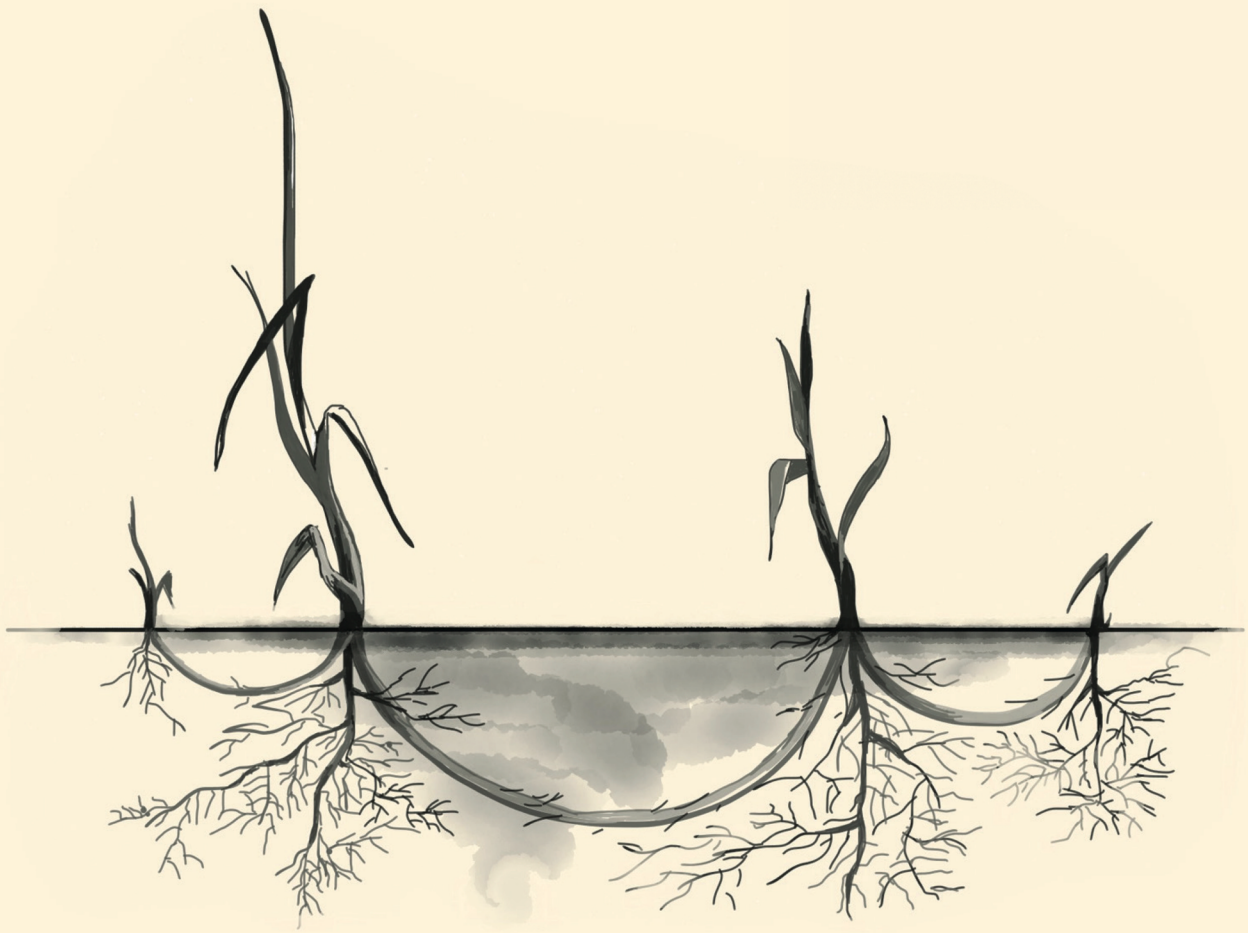
Elements of Clojure



Zachary Tellman

Elements of Clojure

Zachary Tellman

This book is for sale at http://leanpub.com/elementsofclojure

This version was published on 2019-02-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2019 Zachary Tellman

Contents

Introduction **1**

Names **3**

Naming Data 11

Naming Functions 17

Naming Macros 20

Introduction

This book tries to put words to what most experienced programmers already know. This is necessary because, in the words of Michael Polanyi, "we can know more than we can tell." Our design choices are not the result of an ineluctable chain of logic; they come from a deeper place, one which is visceral and inarticulate.

Polanyi calls this "tacit knowledge", a thing which we only understand as part of something else. When we speak, we do not focus on making sounds, we focus on our words. We understand the muscular act of speech, but would struggle to explain it.

To write software, we must learn where to draw boundaries. Good software is built through effective indirection. We seem to have decided that this skill can only be learned through practice; it cannot be taught, except by example. Our decisions may improve with time, but not our ability to explain them.

It's true that the study of these questions cannot yield a closed-form solution for judging software design. We can make our software simple, but we cannot do the same to its problem domain, its users, or the physical world. Our tacit knowledge of this environment will always inform our designs.

This doesn't mean that we can simply ignore our design process. Polanyi tells us that tacit knowledge only suffices until we fail, and the software industry is awash with failure. Our designs may never be provably correct, but we can give voice to the intuition that shaped them. Our process may always be visceral, but it doesn't have to be inarticulate.

And so this book does not offer knowledge, it offers clarity. It is aimed at readers who know Clojure, but struggle to articulate the rationale of their designs to themselves and others. Readers who use other languages, but have a passing familiarity with Clojure, may also find this book useful.

The first chapter, **Names**, explains why names define the structure of our software, and how to judge whether a name is any good.

The second chapter, **Idioms**, provides specific, syntactic advice for writing Clojure which is clean and readable.

The third chapter, **Indirection**, looks at how code can be made simpler and more robust through separation.

The final chapter, **Composition**, explores how the constituent pieces of our code can be combined into an effective whole.

Names

Names should be narrow and consistent. A **narrow** name clearly excludes things it cannot represent. A **consistent** name is easily understood by someone familiar with the surrounding code, the problem domain, and the broader Clojure ecosystem.

Consider this function:

```
(defn get-sol-jupiter
  "Does a deep lookup of key `k` within `m` under
   `:sol` and `:jupiter`, returning `not-found` or
   `nil` if no such key exists."
  ([m k]
    (get-sol-jupiter m k nil))
  ([m k not-found]
    (get-in m [:sol :jupiter k] not-found)))
```

We name the first parameter `m` because it can represent any map, and naming it `map` would shadow the function of the same name. The second parameter is named `k` because it can represent any key, and avoid naming it `key` for the same reason. We name the optional third parameter `not-found` because that's the name used by Clojure's `get` function, as is the default value of `nil`.

The function name itself, however, is potentially confusing. Without reading the doc-string or implementation, a reader might reasonably assume it did any of the following:

```
(get-in m [:sol-jupiter k])
```

```
(get (.sol-jupiter m) k)
```

```
(http/get (str "http://sol-jupiter.com/" k))
```

This name introduces a lot of ambiguity, considering the function can be replaced by its implementation without losing much concision:

```
(get-sol-jupiter m :callisto)
```

```
(get-in m [:sol :jupiter :callisto])
```

But what if we were to change the name to describe its purpose, rather than its implementation?

```
(get-jovian-moon m :callisto)
```

```
(get-in m [:sol :jupiter :callisto])
```

Suddenly, the function begins to justify its existence. Jupiter's moons may be stored under `[:sol :jupiter]` for the moment, but that's just an implementation detail, hidden away behind the name. Our name is now a layer of **indirection**, separating *what* the function does from *how* it does it. We can introduce even more indirection by renaming the first parameter:

```
(get-jovian-moon galaxy :callisto)
```

Now the data structure used for our `galaxy` is also an implementation detail, hidden behind a name.

Indirection, also sometimes called abstraction[1], is the foundation of the software we write. Layers of indirection can be peeled away incrementally, allowing us to work within a codebase without understanding its entirety. Without indirection, we'd be unable to write software longer than a few hundred lines.

Names are not the only means of creating indirection, but they are the most common. The act of writing software is the act of naming, repeated over and over again. It's likely that software engineers create more names than any other profession. Given this, it's curious how little time is spent discussing names in a typical computer science education. Even in books that focus on practical software engineering, names are seldom mentioned, if at all.

Luckily, other fields have given names more attention. Philosophers, in particular, have a special fascination with names. In their terminology, the textual representation of a name is its **sign**, and the thing it refers to is its **referent**. Until the late 19th century, the prevailing theory was that signs and referents were arbitrarily related. A town named Dartmouth doesn't necessarily sit at the mouth of the Dart River. If it did, and the river dried up, the name wouldn't have to change. In the right context, 'Dartmouth' might refer to a crater on the moon. The sign was just a means of pointing at something.

Then a logician named Gottlob Frege pointed out an issue: in Ancient Greece, there were two celestial bodies named *Phosphorus* (Morning Star) and *Hesperus* (Evening Star), both of which happened to be Venus. At first glance, this doesn't seem to be a problem; both signs share a referent, so they're just different ways of talking about Venus. But if Evening Star and Morning Star are just synonyms for each other, then these sentences should be interchangeable:

- Homer believed the Morning Star was the Morning Star.
- Homer believed the Morning Star was the Evening Star.

[1] 'Abstraction' can describe this separation, but can also describe other, different concepts. 'Indirection' is preferable, because it is narrower. This distinction is expanded upon in the third chapter.

The first sentence is obviously true, but the second one is almost certainly false: that fact wasn't discovered until hundreds of years after Homer's death. It's clear, then, that they are not synonyms. We cannot only consider *what* a name references, we must also consider *how* it is referenced. Frege called this the **sense** of a name.[2]

We can construct a similar example using Clojure's semantics. Consider two vars, `a` and `b`:

```
(def a 42)
(def b 42)
```

While `a` and `b` point to the same value, we cannot claim these two statements are equivalent:

```
(= a a)
(= a b)
```

A `var` is a **reference**, a means of pointing at a referent. Clojure does its best to blur the line between reference and referent; `vars` are automatically replaced by their runtime value. But references are a form of indirection, and this gives us a degree of freedom in how the code changes over time. While `a` and `b` are equal today, that may change tomorrow.

The sense of a `var` describes what it is, but also what we expect it to become. If we've defined separate `vars` for the same value, it's because we expect them to diverge. They have the same referent but different senses.

Let's consider a higher-level example: an `id`. We need a means of generating and representing unique identifiers, and after some discussion we settle on UUIDs, which

[2] In the following century, many philosophers have expanded on Frege's work, but their work isn't directly relevant to names in software. Anyone interested in following this thread should begin with Saul Kripke's *Naming and Necessity*.

are randomly generated 128-bit values. Typically, a UUID is displayed as a collection of hexadecimal characters and hyphens, such as `4a4c7d8b-bb8a-441a-982f-80fc90e80e47`.

Having settled on this implementation, we can consider two sentences:

- Our unique identifiers are unique.
- Our unique identifiers are 128-bit values.

The first sentence is true, but the second is only true for our chosen implementation. Should the implementation change, it might suddenly become false. Since the second sentence is not timelessly true, we must treat it as effectively false; anything else would enshrine the 128-bit implementation as permanent, constraining our future designs.

Our **sign**, in the philosophical sense, is a name's textual representation: in the case of our identifier, `id`. A name's **referent** is what it points to: in our example, the UUID implementation. A name's **sense** is the set of fundamental properties we ascribe to it: in this case, the identifier's uniqueness. When we encounter a new name, we only need to understand its sense. The underlying implementation, the referent, can change without us ever knowing or caring.

A narrow name reveals its sense. Narrow doesn't necessarily mean specific; a specific name captures most of an implementation, while a general name captures only a small part. An overly general name obscures fundamental properties and invites breaking changes. An overly specific name exposes the underlying implementation, making it difficult to change or ignore the incidental details. A narrow name finds a balance between the two.

Narrowness doesn't only derive from our choice of sign; we prefer `id` to `unique-arbitrary-string-id`. The sense can be communicated through the surrounding code, through documentation, and through everyday conversation. This means that narrowness can be created or destroyed without ever touching the code. Carelessly substituting

`uuid` for `id` in emails will distort the sense, no matter how clear our documentation. Without constant care, narrowness may disappear.

This is especially difficult because the sense can remain unspoken. In the case of the Morning and Evening Star, differing senses came with differing signs, but in practice this is rarely true. An engineer working on the serialization format for the `id` may decide to use the 128-bit encoding, implicitly treating that encoding as a fundamental property. Another engineer working on a log parser might write a regex that looks for 36 hexadecimal and hyphen characters, implicitly doing the same. Both can have a reasonable conversation about `id`s without any hint that they are speaking past each other.

This is not a problem that can be fully solved. We speak ambiguous words, we think ambiguous thoughts, and any project involving multiple people exists in a continuous state of low-level confusion. It is, however, a problem that can be minimized through **consistency**.

A name whose sense is consistent with the reader's expectations requires less effort from everyone. If the `map` function is redefined within a namespace to return a data structure, this must be carefully documented. Readers must deliberately remember what context a `map` exists in, and will begin to second-guess their intuitive understanding of the code. The code and documentation, then, must clarify what sort of `map` is being discussed *everywhere*, not just within the inconsistent namespace.

Even if we clearly communicate the sense of a name, there can still be inconsistencies between the sense and the referent. Our `id` example suffers from this; our identifier is unique, but UUIDs are only very likely to be unique. If a poor random-number generator is used, collisions between generated identifiers are not only possible, but plausible. Unless we redefine our identifiers as "probably unique", the assumption of uniqueness will be baked into the surrounding code.

If this is a design flaw, it is a flaw shared across a wide variety of software. We can poke

similar low-probability holes in most invariants using cosmic rays, data corruption that still satisfies checksums, and so on. Errors caused by these inconsistencies can be very expensive; they can only be understood by someone familiar with the implementation *and* the assumptions made in the surrounding code. Despite this, checking to determine whether every UUID is unique is impractical. An inconsistent name is not necessarily a bad name.

Often, we can only choose *how* we wish to be inconsistent. Consider a datatype called `student` in software used for university administration. The intuitive sense of this name will differ by department:

- For the admissions office, a `student` is anyone eligible to apply to the university.
- For the bursar's office, a `student` is anyone attending the university.
- For the faculty, a `student` is anyone registered for classes.

If each department writes their own software, each can use `student` without confusion. A sign's sense is inferred from its context, and defining separate contexts allows us to reuse it. More typically, we'd put each department in its own namespace, but then we risk the `admissions` namespace invoking the `bursar` namespace with the wrong kind of `student`. Keeping contexts separate requires continuous effort by the reader, and failing to keep them separate creates subtle misunderstandings.

If we avoid separate contexts, our datatype can only be as narrow as its most general case. If `student` represents anyone who might apply to the university, then our sense is only consistent for the admissions department. To be consistent for everyone, we'd have to create different names for each sense and use `student` for none of them.

In other words, the only way to be fully consistent is to have a one-to-one relationship between signs and senses. This means that we must invent a sign for each sense, but also that readers must agree on their sense. This is why `student` must be avoided at all costs: a dozen different readers might ascribe a dozen different senses. Most **natural** names have

a rich, varied collection of senses.[3] To avoid ambiguity we must use **synthetic** names, which have no intuitive sense in the context of our code.

Category theory is a rich source of synthetic names. 'Monad', to most readers, means nothing. As a result, we can define it to mean anything. Synthetic names turn comprehension into a binary proposition: either you understand it or you don't. Between experts, synthetic names can be used to communicate without ambiguity. Novices are forced to either learn or walk away.

Conversely, a natural name is at first understood as one of its many senses. Everyone understands, more or less, what an `id` is. In a large group, however, these understandings might have small but important differences. These understandings are refined, and gradually converge, through examination of the documentation and code. At the cost of some ambiguity, novices are able to participate right away.

Natural names allow every reader, novice or expert, to reason by analogy. Reasoning by analogy is a powerful tool, especially when our software models and interacts with the real world. Synthetic names defy analogies,[4] and prevent novices from understanding even the basic intent behind your code. Choose accordingly.

[3]The ambiguity and utility of everyday names is explored more fully in William Kent's *Data and Reality*, which was published in the late 1970s just as relational databases were coming into vogue.

[4]Of course, people will still try. This is how the monad became a burrito.

Naming Data

Every `var`, `let`-bound value, and function parameter must be named. When we define a `var` representing immutable data, we control both the sign and referent:

```
(def errors #{:too-hot :too-cold})
```

However, we do not control the sense; two people can reasonably disagree over whether `:too-hard` and `:too-soft` should be added to the set. Even if we narrow our names, the problem persists:

```
(def porridge-errors #{:too-hot :too-cold})
```

```
(def bed-errors #{:too-hard :too-soft})
```

Can we add `:too-watery` and `:too-gummy` to `porridge-errors`, even if Goldilocks never had those specific complaints? We can sidestep this issue by never changing the value:

```
;; DO NOT CHANGE UNDER PENALTY OF HEAT DEATH
(def errors #{:too-hot :too-cold})
```

But if the data will truly never change, we should consider whether it belongs in a `var`. We prefer `Math/PI` to `3.14159...`, because it's shorter and prevents subtle copy-paste errors. If `errors` is used in multiple places, and we don't want to put threats next to all of them, keeping it around is reasonable. Otherwise, it may be best to replace `errors` with its value.

When we define a function parameter, we only control the sign; the data it represents could be literally anything. This problem is exacerbated by Clojure's lack of a type system, but even in languages with sophisticated type systems, most types can encode values that fall outside the type's sense; we might represent an `id` using a 128-bit value, but not all

possible 128-bit values are valid identifiers in our system. Dependent type systems, like those used in Agda and Idris, try to address this problem by narrowing the possible values that the type can represent. But even these languages don't prevent us from making simplistic assumptions or protect us from the consequences when the world doesn't conform to them. Type systems are a tool, not a solution.

If a parameter's sense assumes certain invariants, we can enforce them at the top of the function. The relationship between *our* functions is not adversarial; we do not need to check and re-check invariants at every level of our system. The relationship between our software and the outside world, however, can be adversarial. Most invariant checks should exist at the periphery of our code.

When defining a `let`-bound value we control the sign, but we also control the right side of the `let` binding. While a function parameter's value may be unconstrained, a `let`-bound value is constrained by all the code that precedes it.

Names provide indirection. For `vars`, the indirection hides the underlying value. For function parameters, the indirection hides the implementation of the invoking functions. For `let`-bound values, the indirection hides the right-hand expression:

```
(let [europa    ...
      callisto  ...
      ganymede  ...]
  (f europa callisto ganymede))
```

In this expression, if it's self-evident what `europa`, `callisto`, and `ganymede` represent, then the right side of the `let` binding can be ignored. The right side is a deeper level of the code, relevant only if the *what* of `europa` doesn't satisfy, and we need to understand the *how*.

This is possibly Clojure's most important property: the syntax expresses the code's semantic layers. An experienced reader of Clojure can skip over most of the code and have a lossless understanding of its high-level intent.

Of course, this is only true when we avoid side effects. If the right side of a `let`-binding does something more than return a value, we have to read it exhaustively to reason about how it affects the surrounding code. Readers' ability to safely skim Clojure relies on both its syntax *and* its emphasis on immutability.

The threshold for self-evidency depends on the reader. Every name we create seems self-evident as we create it. Six months later, it may seem less so. A reader with domain expertise and no engineering background will find only a subset of names self-evident. An experienced engineer with no domain knowledge will find a different subset to be self-evident.

Each time they encounter an unfamiliar name, readers must dive deeper into the code and documentation. In the limit case, where every name is unfamiliar and no name is used twice, readers would have to read everything to make sense of anything. However, if we choose consistent names, only a few deep dives are required to understand the core concepts.

Code buried deep under layers of indirection will have a smaller, more determined audience. From that audience, we can expect familiarity with names used elsewhere in the code, and a willingness to understand unfamiliar concepts. Names at the topmost layers of the code will be read by novices and experts alike, and should be chosen accordingly.

Where a value is used repeatedly, we may prefer to use a short name rather than a self-evident one. Consider this code:

```
(doseq [g (->> planets
               (remove gas-planet?)
               (map surface-gravity))]
  ...)
```

If we renamed `g` to `surface-gravity`, most readers could understand the intent without

reading the right-hand expression. Unfortunately, this shadows the function of the same name and is fairly verbose. By itself, though, `g` doesn't mean anything. The reader is forced to carefully read both sides of the binding to understand the intent.

If the left-hand name isn't self-evident, the right-hand expression should be as simple as possible. This is preferable to the above example:

```
(let [surface-gravities (->> planets
                             (remove gas-planet?)
                             (map surface-gravity))]
  (doseq [g surface-gravities]
    ...))
```

Finding good names is difficult, so wherever possible we should avoid trying. If we're performing a series of transformations on data, we shouldn't name every intermediate result. Instead, we can compose the transformations together using –>> or some other threading operator.

If a function's implementation is more self-explanatory than any name you can think of, it should be an anonymous function. This can be true even for relatively complex functions. A large function, named or anonymous, asserts that it cannot be made easier to understand using indirection. A large function is not necessarily a bad function.

If a function has grown unwieldy, but you can't think of any good names for its pieces, leave it be. Perhaps the names will come to you in time.

There cannot be hard and fast guidelines for choosing a good name, since they have to be judged within their context, but where the context doesn't call for something special, there can be a reasonable collection of defaults. The defaults given here are not exhaustive and mostly come from common practices in the Clojure ecosystem. In a codebase with different practices, those should be preferred.

If a value can be anything, we should call it `x` and limit our operations to `=`, `hash`, and `str`. We may also call something `x` if it represents a diverse range of datatypes; we prefer `x` to `string-or-float-or-map`, but those possible datatypes must be explicitly documented somewhere.

If a value is a sequence of anything, we should call it `xs`. If it is a map of any key onto any value, it should be called `m`. If it is an arbitrary function, we should call it `f`. Sequences of maps and functions should be called `ms` and `fs`, respectively.

A self-reference in a protocol, `deftype`, or anonymous function should be called `this`.

If a function takes a list of many arguments with the same datatype, the parameters should be called `[a b c ... & rst]`, and the shared datatype should be clearly documented.

If a value is an arbitrary Clojure expression, it should be called `form`. If a macro takes many expressions, the variadic parameter should be called `body`.

However, for most code we're able to use narrower names. Let's consider a `student` datatype, which is represented as a map whose keys and values are well defined using either documentation or a formal schema. Anything called `student` should have at least these entries, and sometimes only these entries.

The name `students` represents a sequence of students. Usually these sequences are not arbitrary; all students might, for instance, attend the same class. Any property shared by these students should either be clear from the context or clearly documented.

A map with well-defined datatypes for its keys and values should be called `key->value`. A

map of classes onto attending students, for instance, should be called `class->students`. This convention extends to nested maps as well; a map of departments onto classes onto students should be called `department->class->students`.

A tuple of different datatypes should be called `a+b`. A 2-vector containing a tutor and the student they're tutoring should be called `tutor+student`. A sequence of these tuples should be called `tutor+students`.

Notice that `tutor+students` is ambiguous; it can either be a sequence of `tutor+student` tuples or a single tuple containing `students`. Likewise, `class->students` might be a single map, or a sequence of `class->student` maps. Often, it's clear from context which is meant, but otherwise we have to create a name for our compound datatype. If we call our tutor-and-student tuple a `tutelage`, then we can refer to `tutelages` without ambiguity.

But `tutelage` is a synthetic name, as are most names for compound data structures.[5] As such, we need to carefully document their meaning and only use them where our readers will have read the documentation. The naming conventions given here, like anonymous functions and threading operators, are a way to avoid introducing new names until absolutely necessary.

[5]The English language rarely anticipates our need for a particular permutation of nouns.

Naming Functions

At runtime, our **data scope** is any data we can see from within our thread. It encompasses function parameters, `let`-bound values, closed-over values, and global `vars`. Functions can do three things: pull new data into scope, transform data already in scope, or push data into another scope. When we take values from a queue, we are pulling new data into our scope. When we put values onto a queue, we are making data available to other scopes. HTTP `GET` and `POST` requests can be seen as pulling and pushing, respectively.

Shared mutable state creates asymmetric scopes. Consider a public `var` representing an `atom`:

```
(def unusual-events (atom 0))
```

Any thread can dereference this `atom`; the current count is within scope for every thread within our process. However, if we increment `unusual-events` we are taking information local to our thread and making it visible to all the others. Reading from the shared mutable state isn't a pull, but writing to it is a push.[6]

Most functions should *only* push, pull, or transform data. At least one function in every process must do all three,[7] but these combined functions are difficult to reuse. Separate actions should be defined separately and then composed.

If a function crosses data scope boundaries, there should be a verb in the name. If it pulls data from another scope, it should describe the datatype it returns. If it pushes data into another scope, it should describe the effect it has. Sometimes functions simultaneously push and pull data, usually for reasons of efficiency; in these cases the name should capture both aspects, and the documentation should carefully explain the specific behavior.

[6] This asymmetry, and the broader concept of isolated data scopes, is discussed in greater detail in the final chapter, Composition.

[7] Only trivial processes, like `echo` or `cat` in Unix, do not perform all three actions. This is also expanded upon in the last chapter.

If a function takes an `id` and returns a binary payload, it should be called `get-payload`. If it takes an `id` and deletes the payload, it should be called `delete-payload`. If it takes an `id`, replaces the payload with a compressed version, and returns the result, it should be called `compress-and-get-payload`.

If these functions are in a namespace specific to payloads, they can simply be called `get`, `delete`, and `compress-and-get`. We can assume that other namespaces will `refer` to our namespace with a prefix, such as `payload/get` or `p/get`. This means that shadowing Clojure functions like `get`[8] is safe and useful, but we should take care to specify this at the top of our namespace:

```
(ns application.data.payload
  (:refer-clojure :exclude [get]))
```

This signals to our readers that `get` means something else in this namespace. We should also define our `get` at the bottom of the namespace. Then, if we mistakenly use `get` instead of `clojure.core/get` somewhere in the middle, the compiler will complain that `get` is an invalid symbol rather than silently use our alternate implementation.

If a function only transforms data, we should avoid verbs wherever possible. A function that calculates an MD5 hash, defined in our `payload` namespace, should be called `md5`. A function that returns the timestamp of the payload's last modification can be called `timestamp`, or `last-modified` if there are other timestamps.[9] A function that converts the payload to a Base64 encoding should be called `->base64`. In a less narrow namespace, these functions should be named `payload-md5` and `payload->base64`.

However, when modifying data we often have to use a verb. If a function takes a data structure representing a university and returns a university with a student added to a particular department, the function should be called `add-student`. This name, taken

[8] In any place but Clojure's core implementation, `get` should imply pulling data from another scope.

[9] This means that the example function at the beginning of the chapter should lose the `get` and simply be called `jovian-moon`. It's cleaner.

alone, is ambiguous as to whether the student is being added to a department or to the university as a whole. Since the function will be invoked with a `department` parameter, however, this should be immediately clear in context.

Some verbs, like `conj` and `assoc`, are obviously related to data transformation. Most verbs, though, are ambiguous. In some codebases, functions that affect external data scopes have a `!` added to the end of their name. However, this convention is not universal, even among core Clojure functions. Even if your code uses the `!` marker, the best way to keep things clear for your readers is to avoid impure functions where possible and document where necessary.

In theory, a namespace can hold an unlimited number of functions as long as none of them share the same name. In practice, namespaces should hold functions that share a common purpose so that the namespace lends narrowness to the names inside it.

Typically, this means that all the functions should operate on a common datatype, a common data scope, or both. If all the functions in a namespace operate on a binary payload, we can safely omit `payload` from all the names. If all the functions in a namespace are used to communicate with a database, we can easily understand the scope of the functions. If all the functions in a namespace are used to access a particular datatype in a database, we can both use shorter names and easily understand the data scope.

A large number of namespaces is taxing for our readers; if we have ten tables in a database, creating ten different namespaces just so we can write `europa/get` rather than `db/get-europa` has questionable value. Therefore, we should add new namespaces only when necessary. By questioning the need for new namespaces, we implicitly question the need for new datatypes and data scopes, which will lead to simpler code overall.

Naming Macros

There are two kinds of macros: those that we understand syntactically, and those that we understand semantically. The `with-open` macro is best understood syntactically:

```
(defmacro with-open [[sym form] & body]
  `(let [~sym ~form]
     (try
       ~@body
       (finally
         (.close ~sym)))))
```

If we fail to type-hint `sym` as `java.io.Closeable` or something similar, our `with-open` form will give a reflection warning about a `close` method. Anyone who doesn't know the macroexpanded form of `with-open` will search their code for some reference to `close`, find nothing, and be perplexed. To use `with-open` effectively, we must macroexpand it in our heads whenever it appears in the code.

Macros that we understand syntactically require us to understand their implementation, so they are a poor means of indirection. They can reduce the volume of our code but not its conceptual burden. A good name will tell the reader that it is a macro and prompt them to look at the implementation. Any name with a `with` prefix, or which uses the name of a Clojure special form like `def` or `let`, should have a predictable macroexpanded form.

If we expect our code to have a small audience, these macros may become quite large. This can be especially useful to reduce code size in the lower levels of the code, or in tests. In these cases, the macros should be defined and used within a single namespace; the name is unimportant as long as it isn't misleading.

However, some macros are too complex to be understood through their macroexpanded form. The `go` form in `core.async` is one such macro; not even the authors can easily

describe the macroexpansion for arbitrary code. In these cases, we must understand the semantics of the transformation. Transforming arbitrary code is difficult and sometimes impossible; the `go` macro, for instance, skips over any anonymous functions defined in its scope. Readers must not only understand the semantics of the transformation but also its exceptions and failure modes. For this reason, macros that we understand semantically are also a poor means of indirection.

Since macros cannot be self-evident, the clarity of the macroexpanded syntax or semantics matters more than the clarity of the name. Macro names are usually synthetic and require careful documentation.

Names should be consistent. They should build upon their associations within the code and within natural language. Natural names are a powerful, but broad, means of communicating the sense of a name. Synthetic names are, by definition, inconsistent. They prevent readers from reasoning by analogy and bringing their own intuition to bear upon the problem of understanding the intent behind the code.

Names should be narrow. They should communicate their sense without potential for confusion. Natural names have many senses, and they allow groups to assume different senses for the same sign without ever realizing it. These disparate senses will only converge over time through careful, deliberate communication. The learning curve for a synthetic name, on the other hand, is a sheer cliff.

Narrowness and consistency are often in tension. Finding balance requires understanding your audience. Synthetic names have little downside for an audience that already understands them and enable them to communicate complex ideas. For novices, each synthetic name represents an obstacle that must be surmounted. Natural names allow for continuous progress but at the risk of misunderstandings along the way. In different parts of your code, the size and makeup of the audience will vary. The audience will

also change over time; success with an expert audience will inevitably attract less-expert readers.

Names are a fundamental medium for communicating with your readers. The concepts and terminology in this chapter are not a formula for choosing perfect names, but they will give you the tools to enumerate and discuss your options. These concepts will be used in subsequent chapters to discuss other design considerations when writing Clojure.