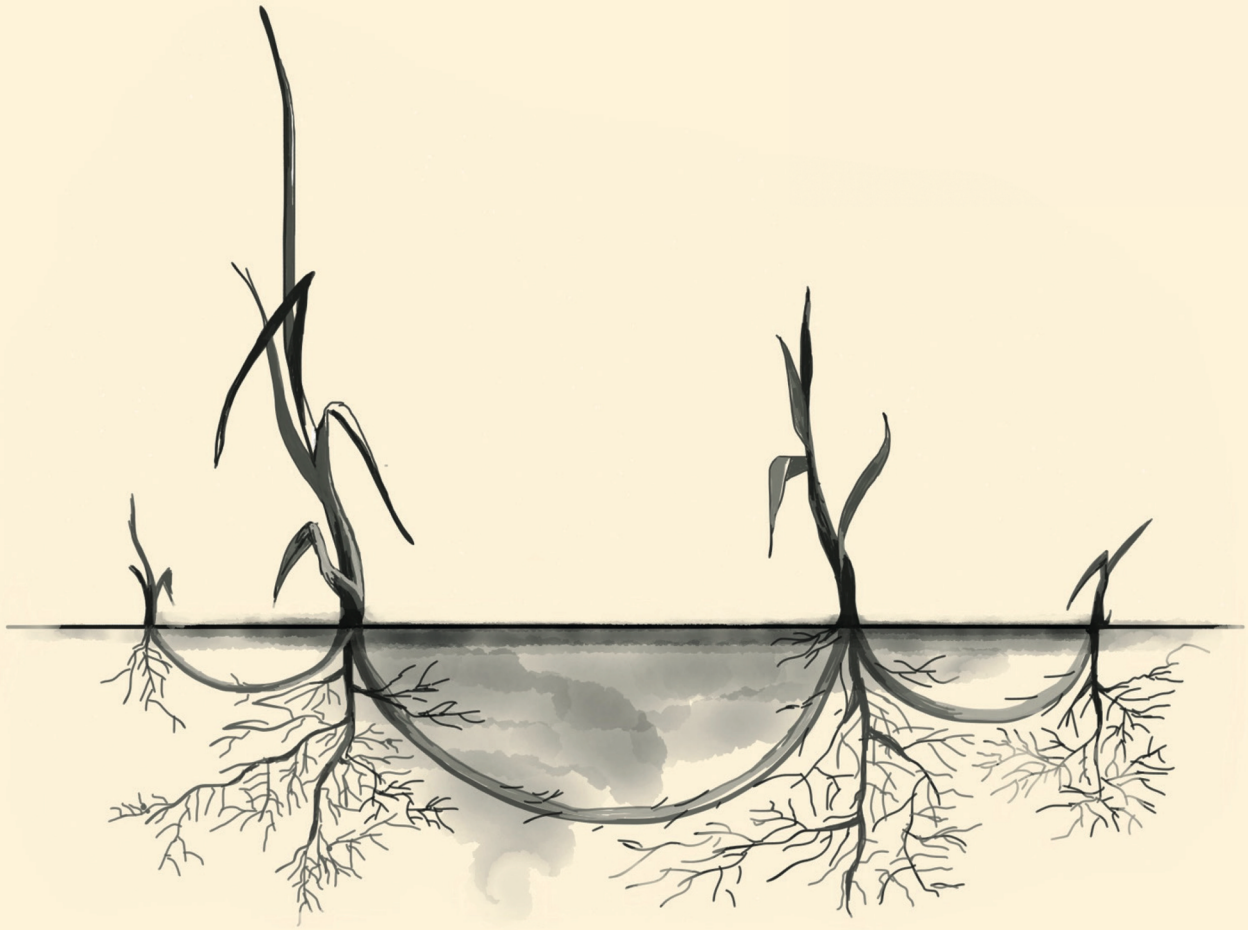
Elements of Clojure



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.

Contents

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.

Acknowledgements

This book began six years ago in a St. Louis bar, where Chas Emerick and I raced to register `elementsofclojure.com` on our phones. I won. In the intervening years, the intent behind that title has been shaped and reshaped by conversations with Kyle Kingsbury, Aaron Crow, Reid McKenzie, Tom Faulhaber, Brandon Bloom, David Nolen, Kovas Boguta, and many others.

During the writing of this book, over a thousand people have purchased early access. My gratitude for their patience and engagement cannot be overstated; I hope everyone finds it was worth the wait.

Above all, I'd like to thank my wife, Sandhya Hegde, who drew the cover art and has participated in countless conversations about the book I was going to write, the book I was writing, and the book I have written. None of this would have been possible without her insight and her support.

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.

Idioms

Software is understood layer by layer. At each layer, readers will guess at the underlying intent. As they descend, they'll sometimes be proven wrong. Once they stop,[10] they'll have only their intuition, and any counterexamples found along the way, to suggest what lies beneath.

A confident reader will, at first, only read the top-most layers of the code. Each time they're proven wrong, this confidence is chipped away. Self-doubt may drive readers to delve deep into the code or may simply drive them away.

Each new layer, if it represents meaningful indirection, should reveal something new. Yet the lessons learned along the way should remain valid; an unfamiliar implementation should represent an unfamiliar idea, not just pointless variation on something we already understand. Each surprise should be meaningful, and any software which constantly surprises its readers is poorly written.

Idioms provide a mapping between code structure and intent. Consistently used, they allow readers to trust their own intuition. Some of the idioms described here have naturally developed in the Clojure community, and others would benefit the community if they saw wider use. This chapter will present each in turn, explain its underlying rationale, and explore when it might not be appropriate.

[10] And everyone has to stop, eventually. Drilling past sixty years of accreted indirection is more than a life's work.

When using inequalities, prefer < and <=

A lifetime of infix notation is hard to shake. Even if prefix notation for arithmetic is more consistent, for most readers it will never feel quite natural.

Addition and multiplication are fairly familiar; `+` returns the sum of everything to its right, and `*` returns the product.

Subtraction and division are a bit more complicated. The expression `(- a b)` reads clearly, but most readers will have to pause to remember whether `(- a b c)` is left or right associative. A useful mnemonic here is to think of

- `(- a b c …)` as `(- a (+ b c …))`; and
- `(/ a b c …)` as `(/ a (* b c …))`

but it's usually best to write this out explicitly.

But the most confusing operators, by far, are the inequalities. As children, we were often taught the meaning of `3 > 1` by giving the `>` tiny teeth, transforming it into a hungry alligator that always wants to eat the largest number. Here the infix notation isn't an incidental detail; it's central to our mental model.

Small wonder, then, that `(> a b)` is difficult to read. The `>` doesn't point at anything; we have to move it over by one term before realizing that the hungry alligator wants to eat `a`. Here, too, there is a useful mnemonic: we can think of `>` as a downward slope, and `<` as an upward slope. This way, we can read `(> a b c)` as a descending collection of values from `a` to `c` and `(< a b c)` as ascending.

Even with this mnemonic, readers can get confused. Subtraction and division, at least, have a fixed relationship between the left and right terms. With each inequality, we have to decide anew whether to use `(< a b)` or `(> b a)`. Almost always, this decision

is arbitrary; we can reduce the burden on the reader by choosing one and sticking with it.

Inequalities should be ordered least to greatest, except where reordering terms hurts code clarity.

```
(cond
  (< a b) ...
  (= a b) ...
  (> a b) ...)

(cond
  (< a b) ...
  (= a b) ...
  (< b a) ...)
```

Here, the first version reads more cleanly, even though it uses >. A typical exception to this rule will be a similar collection of almost-identical predicates, where we'd rather change the operator than the terms.

If a function accumulates values, support every arity

Typically, when calling `reduce` we pass in a 2-arity function, like so:

```
(reduce (fn [x y] (+ x y)) numbers)
```

However, if `numbers` has zero arguments in it, `reduce` will invoke our function with zero arguments, throwing an exception. This wouldn't be a problem if we provided an initial value:

```
(reduce (fn [x y] (+ x y)) 0 numbers)
```

In this case, if `numbers` has zero arguments in it, `reduce` will return `0`. If `numbers` has one argument in it, `reduce` will invoke our function with `0` and the first element. However, many built-in Clojure functions provide their own initial value:

```
> (+)
0

> (+ 1)
1

> (conj)
[]

> (conj [1])
[1]
```

```
> (concat)
()

> (concat [1])
(1)
```

In each of these three cases, the 0-arity method returns a base value, the 1-arity method returns the same value it was passed, and the 2-arity case combines the two parameters. For larger arities, they perform a reduction:

```
(defn concat
  ...
  ([a b & rst]
    (reduce concat (concat a b) rst)))
```

In some circles, this is referred to as a **monoid**, which is a 0-arity function that returns an **identity** value and a 2-arity function that takes two values of the same type and returns a single combined value. Combining the identity value with any other value returns that value unchanged.

A common example of this is a set, whose 0-arity function returns an empty set, and 2-arity function returns the union of the two sets.

```
(require '[clojure.set :as set])

> (set/union)
#{}

> (set/union #{1})
#{1}

> (set/union #{1 2} #{2 3})
#{1 2 3}
```

Unlike `concat`, `+`, and `set/union`, `conj` combines dissimilar types: the first parameter is a collection, and all subsequent parameters are elements. For this reason, it's not a monoid, but the underlying implementation looks very familiar:[11]

```
(defn conj
  ...
  ([coll x & rst]
    (reduce conj (conj coll x) rst)))
```

This is because `reduce` allows dissimilar types. We can use `reduce` with monoids that combine values, but also with any function that *accumulates* values like `conj`. This is true even for functions on specialized datatypes, like `concat-students` or `conj-moon`. Any such function should support 0, 1, and 2-arity calls and also have a variadic implementation.

Of these arities, only the 0 and 2-arity cases are interesting; the 1-arity implementation simply returns the value, and the variadic implementation is a reduction. The value

[11] Neither the `conj` nor `concat` implementations shown here are identical to what's in `clojure.core`, but they are equivalent.

returned by the 0-arity function isn't always obvious; `conj` returns a vector even though it can also aggregate using maps, sets, and lists. Whatever the value, it should be a reasonable default for most uses.

Implementing every arity isn't worthwhile if a function is seldom used and unlikely to be used more in the future. It should also be avoided if there is no appropriate default 0-arity value. In these cases, we can provide only a 2-arity method, which forces every call to `reduce` to provide its own initial value.

Use option maps, not named parameters

If a function takes multiple parameters with default values, we're forced to sort them by importance.

```
(defn pi
  "Calculates pi to `n` digits, with optional parameters
   for whether it should be done efficiently and correctly.
   Both default to `true`."
  ([n]
    (pi n true))
  ([n efficiently?]
    (pi n efficiently? true)
  ([n efficiently? correctly?]
    (cond
      (not correctly?)   3.0

      (not efficiently?) (->> (repeatedly #(pi n))
                              (take 100)
                              last)

      :else              (math/pi-to-n-digits n))))
```

Here, if we specify `correctly?`, we'll also have to specify `efficiently?`, even though one obviates the other. This is true for any function which represents optional parameters using multiple arities: if we specify one parameter, we have to specify all the parameters that come before it. To do this effectively, we have to carefully think about which

parameters are more likely to be specified. With even two or three parameters, there's rarely a clear hierarchy. With the dozen or more parameters required by complex APIs or application logic, it's a lost cause.

So instead, we can specify our parameters by name rather than position:

```
(defn pi
  [n &
   {:keys [efficiently?
           correctly?]
    :or   {efficiently? true
           correctly?   true}}]
  ...)
```

We've defined `n` as a positional parameter because it's required, and we've left all the others as named options. Now, calculating pi incorrectly only requires `(pi n :correctly? false)`. Parameters we don't wish to specify can be ignored. Defaults can be changed, and parameters added, without ever having to update the call sites. For any function with more than one optional parameter, this has obvious benefits.

This convenience, however, does not come free. With each invocation, we must build a `hash-map` and then look up each key in turn. For even moderately complex functions, this can add noticeable overhead.

Also, functions with named parameters don't cleanly compose. Let's assume that our internal function `math/pi-to-n-digits` also takes some named parameters, and we want pass all of our parameters one level deeper:

```
(defn pi
  [n &
   {:keys [efficiently?
           correctly?]
    :or   {efficiently? true
           correctly?   true}
    :as   options}]
  (cond
    (not correctly?)   3.0

    (not efficiently?) (->> (repeatedly #(pi n))
                            (take 100)
                            last)

    :else              (->> options
                            (apply concat)
                            (apply math/pi-to-n-digits n))))
```

First, we take the parameters passed into `pi` and construct a `hash-map` called `options`. But since our inner function also expects individual parameters, we then flatten the map *back* into a list using `(apply concat)` and then `apply` that list to `math/pi-to-n-digits`. Option parameters read nicely when they're written out by hand, but everywhere else they add complexity and noise.

While we write the parameters only once, they typically pass through many layers of our code. This is especially true for the top-level configurations for an application; the parameters are provided at the entry point for the process and have to propagate into many different parts of the code. Fortunately, we can solve this by merely removing the

& symbol:

```
(defn pi
  [n
   {:keys [efficiently?
           correctly?]
    :or   {efficiently? true
           correctly?   true}
    :as   options}]
  (cond
    (not correctly?)   3.0

    (not efficiently?) (->> (repeatedly #(pi n {}))
                            (take 100)
                            last)

    :else              (math/pi-to-n-digits n options)))
```

Here, our options are contained in a map. No additional data structures need to be created, and the map can be passed as-is into our inner function. Any function which accepts optional named values should use this approach. The only cost relative to named parameters is writing out an extra pair of curly braces. However, both approaches introduce measurable overhead; in performance-sensitive contexts, we should only use positional parameters.

The natural shape for a collection of options is a map. Named parameters introduce complexity because, when invoked, they force us to turn that map into something else. But if we only ever write out the parameters by hand, they're harmless. In practice, this

is only ever true of macros; any function we write will eventually be wrapped in another function. Use named parameters sparingly, or not at all.

No one should have to know you've used `binding`

Months ago, in the distant past, we wrote these functions:

```
(defn a [x]
  (b x))

(defn b [x]
  (c x))

(defn c [x]
  (library/compute x))
```

Today, after updating our dependencies, we discover that `library/compute` now has an additional Boolean parameter which makes everything go faster. Unfortunately, a few parts of our code can't handle the raw speed, so we have to make this optional. Since `a` is our public API, we have to thread the parameter all the way through:

```
(defn a [x turbo-mode?]
  (b x turbo-mode?))

(defn b [x turbo-mode?]
  (c x turbo-mode?))

(defn c [x turbo-mode?]
  (library/compute x turbo-mode?))
```

But once we start adding `true` and `false` to every invocation, we realize that `turbo-mode?` is almost always true, so we add default values:

```
(defn a
  ([x]
    (a x true))
  ([x turbo-mode?]
    (b x turbo-mode?))

(defn b [x turbo-mode?]
  (c x turbo-mode?))

(defn c [x turbo-mode?]
  (library/compute x turbo-mode?))
```

This works. Our refactoring is now limited to the parts of our codebase which relied on `library/compute` being slow. The cost, however, is high: we've added a positional parameter which is almost never used. If we ever add more parameters, we'll either have to switch to an option map or start specifying `turbo-mode?` everywhere just so we can specify the new parameter. Any new functions that call `b` or `c` will also pay this tax.

Discontented with this tradeoff, we try something different:

```
(def ^:dynamic *turbo-mode?* true)

(defmacro slowly [& body]
  `(binding [*turbo-mode?* false]
     ~@body))

(defn a [x]
  (b x))

(defn b [x]
  (c x))

(defn c [x]
  (library/compute x *turbo-mode?*))
```

This seems even better. Now we only need to wrap some parts of our codebase in `slowly` and otherwise leave things as they are. Future changes to our code aren't affected either.

After this change, though, things seem a bit flaky. A lengthy and frustrating investigation reveals that this expression is the culprit:

```
(slowly
  (let [values' (map a values)]
    (if (empty? values')
      (throw (IllegalArgumentException. "empty input"))
      values')))
```

Here we map `a` over a sequence and then make sure it isn't empty. Every time `values` has more than 32 elements, though, something seems to go wrong. The issue is that our

`slowly` block returns a lazy sequence, which can be evaluated outside of the `binding` form. This would have been obvious had the expression looked a little different:

```
(slowly
  (if (empty? values)
    (throw (IllegalArgumentException. "empty input"))
    (map a values)))
```

In this expression, we do an `empty?` check on the input and return a completely unrealized sequence. This means that `a` is always evaluated outside the `slowly` block, which probably would have been caught by our tests.

Unfortunately, in the first expression we do an `empty?` check on the result, which realizes the first element. Because `values` happens to be a `chunked-seq`, realizing the first element also realizes the next 31 elements. These elements are evaluated within the `slowly` scope, but any elements which come afterwards aren't. Since our tests use small collections, we only see this failure in production.

Laziness relies on **referential transparency**, which formally means that an expression and its result are interchangeable. We can replace every instance of `(+ 1 1)` with `2`, or vice versa, without changing the semantics of our code. Of course, there are situations where this isn't true:

```
(let [+ (fn [a b]
          (println "I'm adding" a "and" b)
          (+ a b))]
  (+ 1 1))

(let [+ (fn [a b]
          (println "I'm adding" a "and" b)
          (+ a b))]
  2)
```

Side effects mean that we can't just focus on *what* an expression returns, we also have to think about *how* it produces that value. They make the expression **referentially opaque**. Referential transparency also requires an expression to always return the same value, no matter where it's evaluated. Dynamic scope breaks this invariant:

```
(def ^:dynamic *n* 1)

(defn add-n [x]
  (+ x *n*))
```

We can't safely replace `(add-n 1)` with `2`, because in a different context it might return something else. Any use of dynamic `vars` and `binding` suffers from this problem.

Laziness relies on referential transparency, and `binding` breaks it. More generally, almost any higher-order function assumes referential transparency. When we pass a function as a parameter, how and when it's invoked is almost always an implementation detail.

An experienced programmer can, with care, understand and work within these details. They can avoid laziness or be sure to always realize lazy sequences within the proper scope. Clojure also provides mechanisms like `bound-fn`, which captures the dynamic

scope where the function is defined and applies it when it is invoked. But these are easy to forget, and forgetting can lead to subtle bugs.

Consider a database client library which provides a `with-db` macro. In the best case, lazy evaluation might cause us to make a call to the database where `*db*` is undefined. But `chunked-seqs` might hide the issue until our code has been running for months and our dataset has had time to grow. Worse yet, our lazy calls might be invoked not in an undefined context but within the scope of a *different* `with-db` macro, giving us a chimeric sequence of values without any complaint.

This is why, in our original example, the `slowly` macro is a poor design choice. An experienced reader might infer that it uses `binding`, but they can, and will, forget. A less experienced reader might miss it entirely. Only the writer, at the time of writing, can safely rely on dynamic scope.

With a small change, all these problems disappear:

```
(def ^:dynamic *turbo-mode?* true)

(defn a
  ([x]
    (b x))
  ([x turbo-mode?]
    (binding [*turbo-mode?* turbo-mode?]
      (b x))))

(defn b [x]
  (c x))

(defn c [x]
```

```
  (library/compute x *turbo-mode?*))
```

Now the result of `a` depends only on its parameters. It uses `binding`, but that's safe because we know that `c` won't be invoked lazily. Our `slowly` macro could be misused by anyone invoking `a`, while our internal `binding` form can only be misused by someone changing the implementations of `b` or `c`.

Dynamic scope allows us to connect pieces of code without modifying everything in between. It can be a powerful tool for simplifying the internals of a codebase. It can also be a useful way to make complex code testable: a `binding` form at the top-level of a test, and a dynamic `var` with a default value only overriden in the scope of that test, suffers from none of the issues discussed above. However, any dynamic `var` invites re-binding; it may be safer to use `with-redefs` instead.

If you have mutable state, use an `atom`

Clojure's `ref` and `atom` constructs solve the same problem in different ways. Let's consider the implausible example of a bank which stores its account balances in memory, in a single process. We want to implement a `transfer!` function which accepts mutable data representing all balances, transfers money between accounts `a` and `b`, and returns a map with the updated balances for those accounts.

To solve this with `ref`s, we use an `account->balance` map whose values are individual `ref`s containing each account's balance:

```
(defn transfer! [account->balance a b amount]
  (dosync
    {a (-> account->balance
           (get a)
           (alter - amount))
     b (-> account->balance
           (get b)
           (alter + amount))}))
```

To solve this with atoms, we use an `account->balance` atom which contains an immutable map from account identifiers to balances:

```
(defn transfer! [account->balance a b amount]
  (-> account->balance
    (swap!
      #(-> %
          (update a - amount)
          (update b + amount)))
    (select-keys [a b])))
```

Clojure's software transactional memory (STM) implementation allows us to update separate mutable values atomically. However, we can accomplish the same result by putting those separate values into a single data structure and wrapping them in an atom. The difference between these approaches is the expected throughput.

The `swap!` function is implemented using a compare-and-set (CAS) primitive:[12]

```
(defn swap! [atom f & args]
  (loop []
    (let [x  @atom
          x' (apply f x args)]
      (if (compare-and-set! atom x x')
        x'
        (recur)))))
```

The `compare-and-set!` operation allows us to update the atom only if its value hasn't changed since we dereferenced it. If it has changed, we get the latest value and try again. If there are concurrent transfers between any accounts in the atom-based `transfer!` implementation, only one will succeed and all the others will have to retry.

[12]The actual implementation is written in Java, but this is equivalent.

STM uses a similar approach: update values within a transaction, and retry if someone else also did an update during our transaction. However, we only retry if our specific references were touched. In the STM-based `transfer!` implementation, if there are concurrent transfers between accounts, we will only retry if the *same* account is updated multiple times.

The **utilization** of a state container is a measure of how often it is in the process of being updated. In general, we expect to see retries increase dramatically whenever the utilization is greater than 60%.[13] Since the atom-based implementation has only one container, we will approach that threshold more quickly.

But even if the atom has lower throughput, it's almost certainly enough. Inside our `swap!` call, we're just calling `assoc` twice, which even on a very slow machine will take less than a microsecond. Hitting the 60% utilization threshold, then, would require more than 600,000 transfers per second. If each transfer is triggered by an external request, getting anywhere near this threshold will require expensive hardware, a fanatical devotion to efficiency, or both.

So while Clojure's STM offers better throughput in extreme scenarios, it rarely offers a practical improvement. Atoms are simpler and introduce less overhead. In typical situations with little or no contention, atoms are faster.

Furthermore, STM can be difficult to use correctly. Much like dynamic scope, transactions break referential transparency. Lazy realization of `alter`, `commute`, or `ref-set` could happen in the wrong transaction, leading to subtle errors. The `commute` function invites other mistakes; in our original example, we might realize that addition and subtraction are commutative and make this change:

[13] This is simplistic, but still a very useful rule of thumb. It treats the state container as an M/D/1 queue, which has an exponentially distributed interval between each update, and a deterministic cost for each update. Neither of these will ever be exactly true, but they're true enough in most cases. In such a system, overhead is proportional to the square of the service time; halving the time for each update will quarter the number of retries.

```
(defn transfer! [account->balance a b amount]
  (dosync
    {a (-> account->balance
           (get a)
           (commute - amount))
     b (-> account->balance
           (get b)
           (commute + amount))}))
```

Since concurrent calls to `commute` don't cause retries, this gives us optimal throughput. However, the return value is no longer valid; the value returned by `commute` may differ from the one committed at the end of the transaction. This inconsistency won't be visible in tests, or even in low throughput production systems, making it all the more dangerous.

More broadly, transactions make it difficult and expensive to get a consistent snapshot of our state. In the atom-based implementation, if we want a snapshot of all balances we can simply dereference `account->balance`. With STM, it's a bit more complicated. We can't, for instance, just do this:

```
(defn balance-snapshot [account->balance]
  (->> account->balance
    vals
    (map deref)
    (zipmap (keys account->balance))))
```

Since a transaction can occur midway through, the values returned will not be consistent. To fix this, we need to make two changes:

```
(defn balance-snapshot [account->balance]
  (dosync
    (->> account->balance
      vals
      (map ensure)
      (zipmap (keys account->balance)))))
```

First, we need to wrap our reads in their own transaction. Second, since using `deref` inside a transaction does not guarantee the values won't change later in the transaction, we also need to change `deref` to `ensure` to make sure our values are completely consistent. Unfortunately, this means that transfers between any accounts may cause our snapshot function to retry. Likewise, taking a snapshot may cause transfers to retry.

When Clojure was first announced, much of the focus was on its state primitives, and specifically its STM implementation. Years later, it seems clear that the real value derives from its immutable data structures and that atoms solve most problems involving state. Agents, which avoid retries by introducing an unbounded queue, are best avoided. STM is useful, but only in a narrow set of cases involving write-heavy workloads that can't be offloaded to a database.

If you have mutable state, make sure it belongs inside your process. If it does, try to represent it as a single atom. If that causes performance issues, try spreading the work across more processes. If that isn't possible, see if the atom can be split into smaller atoms that don't require shared consistency. Finally, if that doesn't help, you should start looking into Clojure's STM primitives.

An explicit `do` block implies side effects

Any time we ignore the value returned by an expression, a side effect is occurring. Any function which returns `nil` exists solely to perform a side effect. Understanding these side effects is a crucial part of understanding a codebase. A `do` block tells the reader that something important is happening.

Many forms in Clojure, however, contain an *implicit* `do` block. These include the special forms `fn`, `let`, and `loop`, and macros such as `when`. Such forms rarely contain side effects and do not invite the reader to look closely. If there is a side effect, we must take extra care to get the reader's attention.

One possible approach is to add a redundant `do` block:

```
(let [moon (choose-moon)]
  (do
    (fire-rocket! moon)
    (await-landing moon)))
```

It's often simpler, however, to draw attention through the use of negative space:

```
(let [moon (choose-moon)]

  (fire-rocket! moon)

  (await-landing moon))
```

Specifically within `let` bindings, we can call out inline side effects by assigning the return value to _:

```
(let [moon (choose-moon)
      _    (fire-rocket! moon)]
  (await-landing moon))
```

Any approach is fine, so long as it is used consistently throughout the codebase.

Use the narrowest possible data accessor

Clojure's data structures have many guises. A vector can be treated as a map of its indices onto its elements:

```
> (get [0 1] 1)
1

> (contains? [0 1] 2)
false
```

A map can be treated as a sequence of "entry" objects, which themselves can be treated as vectors:

```
> (map key {:a 1, :b 2})
(:a :b)

> (map first {:a 1, :b 2})
(:a :b)
```

Similarly, Java `List` and array objects can be coerced into a `lazy-seq`, and every sequence operator in `clojure.core` does this implicitly.

```
> (->> [1 2 3]
    int-array
    (map inc))
(2 3 4)
```

This allows us to focus on the ways these data structures are similar, rather than their subtle differences. However, it also means that there are many paths to the same data transformation. If we want a sequence of the keys in a map, we can use the approaches shown above or simply call `(keys m)`. Often a single codebase will use all three approaches, even if there's a single author.

While these are functionally equivalent, they are very different for the reader. Seeing `(map first x)` only tells us that `x` is a sequence of sequences, while `(map key x)` implies that `x` is a sequence of entries; only `(keys x)` tells us, definitively, that `x` is a map.

Less ambiguous names can help, but are not enough on their own. Naming a map `m` or `a->b` clarifies our intent, but this clarity must be mirrored within the code itself.

Clojure makes it possible to ignore the differences between data structures, but that doesn't mean we should. Often, these differences matter. By using generic accessors, such as invoking a collection as a function rather than using `get`, `nth`, or `contains?`, we strongly imply that they don't matter. The subtext of our code should always reflect our intent.

Use `letfn` for mutual recursion

The `let` form is familiar to any Clojure reader; the name goes on the left and the evaluated form on the right. For multiple bindings, it often helps legibility to align our columns:

```
(let [two        (+ 1 1)
      twenty-two (+ 11 11)]
  ...)
```

This is true for `let`, `loop`, and any other macros that provide lexical bindings. The only outlier, perhaps in the entire Clojure ecosystem, is `letfn`:

```
(letfn [(cube [x] (* x x x))]
  ...)
```

This encloses the left and right sides of the binding in a single form, which is unusual enough that it will cause even experienced readers to stumble. The above form is equivalent to this:

```
(let [cube (fn cube [x] (* x x x))]
  ...)
```

Note that `cube` appears twice in the expanded form: first as the lexical name and second as the name that is shown in stack traces. The fact that `letfn` avoids this duplication doesn't make up for its structural irregularity. The only real value of `letfn` is that it allows out-of-order references between functions:

```
(letfn [(a [x] (b x))
        (b [x] (a x))]
  ...)
```

These functions will endlessly recurse without doing anything useful, but they will compile. Use of `letfn` should be confined to cases of mutual recursion and avoided everywhere else.

Java interop should be obvious

Clojure's data types are best understood via their innards. When reading unfamiliar code, our focus is on how the four fundamental datatypes (maps, sets, vectors, and seqs) have been nested within each other. This tells us not only what the data is but how to interact with it.

A Java object, on the other hand, is best understood via its name. Even if we know what it contains, we can only know how to interact with it by reading the documentation. This requires a fundamentally different mindset from the reader.

Fortunately, Java interop looks noticeably different from normal Clojure code. Unfortunately, this is easily subverted:

```
(.. (java.util.HashMap.)
  (put :ganymede :jupiter)
  (put :phobos :mars)
  (put :oberon :uranus))
```

Clojure's `..` macro reduces the amount of punctuation in our code, but it turns Java interop into something that is recognized through a form's context rather than the form itself. A reader could easily mistake `put` for a Clojure function, rather than a Java method. This macro and others like it come at too high a cost. Unusual code should be allowed to look unusual.

Use `for` to create cartesian products

In Clojure, transformations are best done piecemeal:

```
(->> s
  (remove nil?)
  (map :user)
  (group-by :department))
```

In this expression, steps can easily be added, removed, and reordered. If we're especially concerned with performance, we can use transducers, but this only works for the subset of supported functions:

```
(->> s
  (eduction
    (comp
      (remove nil?)
      (map :user)))
  (group-by :department))
```

Since `group-by` isn't implemented as a transducer, we have to nest the surrounding operations in `(eduction (comp ...))`. This obscures our intent, making it harder to think of each transformation as a separable piece of computation.[14]

Similarly, the `for` macro provides a syntax for simple list comprehensions:

[14]The real value of transducers is not performance, but rather that non-standard data representations like `core.async` channels can use `clojure.core` directly rather than having to define their own `map`, `filter`, and so on.

```
(group-by :department
  (for [record s
        :when  record
        :let   [user (:user record)]]
    user))
```

This syntax encompasses simple operations like `map` and `filter` but not more complex operations like `group-by`. This means our declarative list comprehension will often sit awkwardly within a larger chain of transformations.

However, `for` does have a unique ability:

```
(for [a [1 2 3]
      b [:a :b :c]]
  [a b])
```

This will generate every possible combination of `a` and `b`, also known as their **cartesian product**. Whenever we require such a thing, we should use `for`, unadorned by any `:let`, `:when`, or `:while` clauses.

The declarative nature of `for` can also be useful when defining data literals:

```
[:html
  [:ul
    (for [item todo-items]
      [:li item])]]
```

Here we define an HTML document containing a to-do list. The `for` macro functions like a template, the round brackets clearly differentiated from the square brackets of the surrounding data literals. Here again, however, we should avoid special `for` clauses, as they would only serve to confuse between data literals and executable expressions.

`nil` should be the absence of only a few values

The `nil` value is one of the trickiest parts of Clojure, because it represents an absence. The absence of what, exactly, depends on the context. For `conj`, `cons`, and `nth`, it is the absence of a `seq`:

```
> (conj nil :callisto)
(:callisto)

> (cons nil :callisto)
(:callisto)

> (nth nil 42)
nil
```

For `count`, it is an empty collection:

```
> (count nil)
0
```

For `assoc` and `get`, it is the absence of a map:

```
> (assoc nil :callisto 1610)
{:callisto 1610}

> (get nil :callisto :unknown-year-of-discovery)
:unknown-year-of-discovery
```

For `if`, it is the absence of truth:

```
> (if nil
    :true
    :false)
:false
```

If we don't define a default value, `nil` is the absence of whatever we're looking up:

```
> (get {} :callisto)
nil
```

This is important to remember, because Clojure treats anything that isn't a map as an empty map:

```
> (get 1 :callisto)
nil

> (get (Object.) :callisto)
nil
```

As a result, we cannot consider a function that returns `nil` in isolation; we have to look at the downstream functions to make sure they interpret our `nil` correctly. Consider a map of keywords onto vectors of numbers:

```
(def key->numbers
  {:a [1 2 3]
   :b [4 5 6]})
```

If we look up a nonexistent keyword, we'll get a `nil` representing the absence of a vector. Unfortunately, none of Clojure's standard functions interpret `nil` that way.

```
> (-> key->numbers
    :a
    (conj 8 9 10))
[1 2 3 8 9 10]

> (-> key->numbers
    :c
    (conj 8 9 10))
(10 9 8)
```

To fix this, we must be explicit about our absent values:

```
> (-> key->numbers
    (:c [])
    (conj 8 9 10))
[8 9 10]
```

Composing `nil`-friendly functions can create an explosion of ambiguity:

```
(-> solar-system :jupiter :callisto :mass)
```

Here, `nil` may represent the absence of the `:mass`, `:callisto`, or `:jupiter` keys, or the absence of the entire `solar-system`. Sometimes, the differences between these explanations may not matter; one way or another, we don't know the moon's mass. Explicitly representing each case in our code is needlessly verbose:

```
(if solar-system
  (if-some [jupiter (:jupiter solar-system)]
    (if-some [callisto (:callisto jupiter)]
      (if-some [mass (:mass callisto)]
        mass
        nil)
      nil)
    nil)
  nil)
```

If we simply pass along an ambiguous `nil`, however, the ambiguity will grow, and the first peson bit by a `NullPointerException` will have to walk backward through the code, testing each hypothesis in turn. Ambiguity makes our code more concise, but unbounded ambiguity makes it impossible to reason about. To protect ourselves, we must interpret `nil` at regular intervals throughout our code.

```
(-> solar-system :jupiter :callisto (:mass :mass-not-found))
```

This conflates the meanings of `nil` within our expression but separates it from the meanings of `nil` everywhere else. Treating mass as a number-or-keyword datatype is inelegant, but so is number-or-nil; the keyword, at least, is harder to ignore or misinterpret.

This means that effective Clojure should avoid this all-too-common idiom:

```
(defn some-function [x & args]
  (when x
    ...))
```

Wrapping a function in a `when` clause simply passes the buck, making `nil` someone else's problem. If `nil` can be coerced to an empty collection, we should do that. If not, we should throw an error. Anything else sows needless confusion.

Indirection

Indirection provides separation between *what* and *how*. It exists wherever "how does this work?" is best answered, "it depends." This separation is useful when the underlying implementation is complicated or subject to change. It gives us the freedom to change incidental details in our software while maintaining its essential qualities. It also defines the layers of our software; indirection invites the reader to stop and explore no further. It tells us when we're allowed to be incurious.

There are two fundamental tools for indirection: **references** and **conditionals**. A reference is a value that points to another value, its referent. Getting the referent is called **dereferencing**. A name is a lexical reference that is dereferenced at compile time. A pointer is a memory reference that is dereferenced at runtime.

In both cases, it is possible for a reference to point to nothing. In the case of names, this will cause a compile error. In the case of a pointer, we represent "nothing" with the `nil` value, which has been known to cause runtime errors.

Any function that takes non-primitive values uses references. The behavior of `filter`, for instance, depends on references to both a predicate function and a sequence. These values are implicitly dereferenced, unlike Clojure's concurrency primitives, which require explicit dereferencing.

A conditional is any expression that uses an `if` or `case` form, making its behavior dependent on the input values. This is necessary when only a subset of possible values is valid or, more generally, when different subsets of possible values have different semantics.

We can see both of these in the semantics of Clojure's `nth` function, which is partially reproduced here:

```
(defn nth [x idx]
  (cond
    (string? x)         (.charAt ^String x idx)
    (instance? List x) (.get ^List x idx)
    ...))
```

In this function, the value of `idx` is an arbitrary integer value, but only values within `[0, size)` are valid. The value of `x` can be any reference type, but only a fixed set of collection types is valid. For each of these collection types, the lookup method is different. Conditionals can be used to segment behavior for a given type, unify behavior across many types, or both.

A reference *conveys* values, and a conditional *decides* based upon values. These are complementary primitives and are present in every modern language. Through their composition, we can create software of arbitrary complexity.

The importance of these primitives is reflected in modern computer hardware. Memory indirection requires costly lookups in main memory, and so we created a hierarchy of caches, each smaller and faster than the one above. Conditional jump operations prevent the pipelined execution of instructions, and so we introduced branch prediction. Enormous effort has been poured into these optimizations because we can't live without indirection. All we can try to do is minimize its cost.

These primitives differ in how we change their behavior. A reference is **open**; we can change the behavior of the dereferencing code by conveying different values. Conversely, a conditional is **closed**; we can only change its decision process by changing the underlying code.

It's tempting to say that we can create an "open" conditional by putting all the predicates and clauses into a data structure. Using this approach, we can modify the decision process simply by changing the data. Since any nontrivial data structure uses both references and conditionals, we might expect it to inherit the best qualities of each.

However, one of the fundamental properties of conditional code is that it is *ordered*. The behaviors of these two `cond` expressions are very different:

```
(cond
  (<= 0 n 10) ...
  (<= 5 n 15) ...)

(cond
  (<= 5 n 15) ...
  (<= 0 n 10) ...)
```

In the first expression, we describe one behavior for inputs in `[0, 10]` and another behavior for inputs in `[11, 15]`. In the second expression we describe one behavior for inputs in `[5, 15]` and another behavior for inputs in `[0, 4]`. If our predicates aren't disjoint, order matters.

The predicates in our hypothetical data structure may overlap. That means the order in which we evaluate them matters, and that order is described by the code that populates our data structure. Our decision process is still closed; all we've done is change where the specification lives.

For a decision-making mechanism to be open, it must be unordered. Typically this is implemented using a data structure with a distinct set of keys, which we will refer to as a **table**.

For a table to be useful, it must avoid conflicts. One way to accomplish this is to keep the table private so it reflects only our vision of how keys map onto behavior. Alternatively, we can extend the table using only private keys so no one else can shadow our behavior.

Failure to do either will land us in a situation similar to Ruby, where libraries sometimes make conflicting monkey-patches to fundamental datatypes, each trying to satisfy their

own narrow use case. The errors that arise from this are often subtle and difficult to track down.

Conditionals solve conflicts by making an explicit, fixed decision. Where conflicts are possible, we use conditionals *because* they are closed.

Method Dispatch

Method dispatch allows us to associate a single method with one of many implementations. If the association occurs at compile time, it is called **static dispatch**. If the association occurs at runtime, it is called **dynamic dispatch**.

All of Clojure's dispatch mechanisms – interfaces, protocols, and multimethods – are implemented using tables. All of them are open, but to different degrees. In general, their efficiency is inversely proportional to their openness.

Interfaces dispatch on the class of the object. Any class may implement an interface, but the implementation must be defined within that class. This means that conflicts are impossible; the association, if any, between an interface and a class can only be decided by the author of the class. This also means that static dispatch is possible, as long as we invoke using the concrete type rather than the interface.

Protocols also dispatch on the class of the object, but anyone may define a relationship between a protocol and a class. This means that static dispatch is impossible, and understanding that relationship may require reading the entire codebase. It also means that if both a protocol and class are publicly visible, we risk defining conflicting extensions. In practice, when extending a protocol over a class, either the class or protocol should be a hidden implementation detail. This avoids conflicts and means we only need to examine the code near the hidden class or protocol to understand the relationship.

Multimethods dispatch on a key derived from all arguments to the function. This is much more flexible than either interfaces or protocols, at the cost of some performance. As with protocols, we must take care to avoid defining relationships between multimethods and keys that are both publicly visible. For instance, Clojure's `print-method` allows us to change the behavior of `prn` for any type. If we were to do this for a common class, like `clojure.lang.PersistentVector`, it could be disastrous.

When using Clojure's hierarchy mechanism, it's possible to define multimethod keys

with overlapping scopes. This can be resolved using `prefer-method`, but this is a closed decision process that can be defined and overridden anywhere in the codebase. The complexity and risks introduced by hierarchies are rarely worthwhile.

We're forced to worry about collisions when using protocols and multimethods because both rely on shared global state. This allows us to incrementally define dispatch behavior and saves us from having to thread the dispatch table through all of our function calls. This is usually a worthwhile tradeoff, but when it's not, we're forced to try something else.

In the most trivial case, we can define a `class->method->impl` data structure and define an invocation helper:

```
(defmacro invoke
  [class->method->impl x method & args]
  `((get-in ~class->method->impl [(class ~x) ~method])
    ~x
    ~@args))
```

Then we can define functions that expect the dispatch table as a parameter and use our `invoke` macro rather than standard invocation.[15] This is ungainly but much more flexible; we can freely change dispatch behavior within a local scope without affecting the rest of our code. We can even create higher-order abstractions to describe these local changes. This approach is rarely necessary, but we should never forget that it is within reach.

[15] We can't avoid the explicit parameter by using dynamic scope because it wouldn't be compatible with lazy evaluation.

What is an Abstraction?

Indirection is a mechanism for creating *abstractions*, which is a word we've carefully avoided until now. To explain why, we'll look at two concepts that are fundamental parts of Clojure's lineage: the cons cell and Church numerals.

In its most common usage, the cons cell represents a list. It contains two values, the first representing an element and the second representing a reference to the next cell. By creating a new cell which references a list, we effectively prepend to it.

A Church numeral represents the natural numbers through function composition. The number `n` is a function which takes a value and a function `f`, and applies `f` to the value `n` times. By composing a "successor" function with an existing numeral, we effectively increment it.

These concepts have a structural similarity: they accumulate through creating references to a previous value. However, while the cons cell has seen widespread use in software, the Church numeral has only been used in mathematical proofs. This is because the cons cell is a practical model for representing lists in memory, while Church numerals are a deeply impractical model when compared to a binary representation.

Of course, this isn't a fair criticism of Church numerals; they were never meant to be an actual method for performing arithmetic. They were designed to be a useful tool for mathematical proofs, and in this they were successful. They are, in a sense, timeless.

But the cons cell is not timeless; since its invention in the late 1950s, computers have changed. Notably, processor speed has improved more than memory latency, and so the relative cost of following a reference has grown over time. For this reason, Clojure favors data structures which represent lists using 32-element blocks, such as vectors and `chunked-seqs`.[16]

[16]The only meaningful exception to this rule is Clojure's syntax trees, which are small and rarely processed at runtime.

Both the cons cell and the Church numeral would commonly be called abstractions. They both fit our informal definition: they are conceptual tools built using indirection. However, one is meant to run on a physical machine, and the other is not. One is judged with respect to a changing context, and the other is not.

A common formal definition of abstraction comes from C.A.R. Hoare's paper *Proof of Correctness of Data Representation*, published in 1972. Hoare distinguishes between a data structure's concrete representation, which is its internal model, and its abstract representation, which is the interface it exposes. In his terminology, mapping the concrete representation onto its abstract representation is done via an *abstraction function*.

Consider a data structure that represents an integer set and provides `add` and `contains?` methods. In this case, we could implement it as a simple list of integers, where `contains?` scans the list to see if it can find the number and `add` appends it to the end of the list. This is a bit wasteful since the list may contain duplicate numbers, but it works.

This approach is less acceptable if we need to implement a `remove` method. Ideally, we'd like to search for the first instance of a number and, if it's found, remove it. However, in the above approach we'll need to scan the entire list every time because there might be duplicates. To simplify, we decide that `add` will only append the number if it's not already in our list.

Now our model is not just a list of integers; it's a list of *unique* integers. This is trivially true for an initial empty list and will remain true after each invocation of `add` or `remove`. Hoare calls this an *invariant relation* on the concrete representation. To determine if a method is correct, we only need to consider the model and its invariants. As long as `add` enforces the invariant, its implementation doesn't affect `remove`.

We are also able to change the model without affecting the abstract behavior of the data structure. If we decide that our list should be sorted, `add` and `remove` will have to be changed to enforce this new invariant, but the semantics of our interface will remain the same.

Hoare's paper, as the title suggests, is concerned with constructing proofs. As we saw with the Church numeral, proofs lack context; they are only concerned with being self-consistent. Likewise, Hoare's terminology describes the abstraction's interface and its internals, but it doesn't acknowledge that the abstraction exists within an environment.

This is not a small omission. Consider that the mechanical clock existed for centuries before we devised one which could keep accurate time aboard a ship. This wasn't for lack of trying; keeping accurate time allowed determining the ship's longitude, and many lives and fortunes were lost to poor navigation. The ocean happens to be a very difficult environment: it constantly moves, temperatures fluctuate wildly, and gravity is 0.5% stronger at the poles than the equator.

An early mechanical clock was at the mercy of its environment. It had to sit on a flat surface and couldn't be rocked, shaken, or dropped. It had to be regularly wound. The temperature couldn't change too much or too quickly. These were not invariants, because the clock couldn't enforce them. They were assumptions.

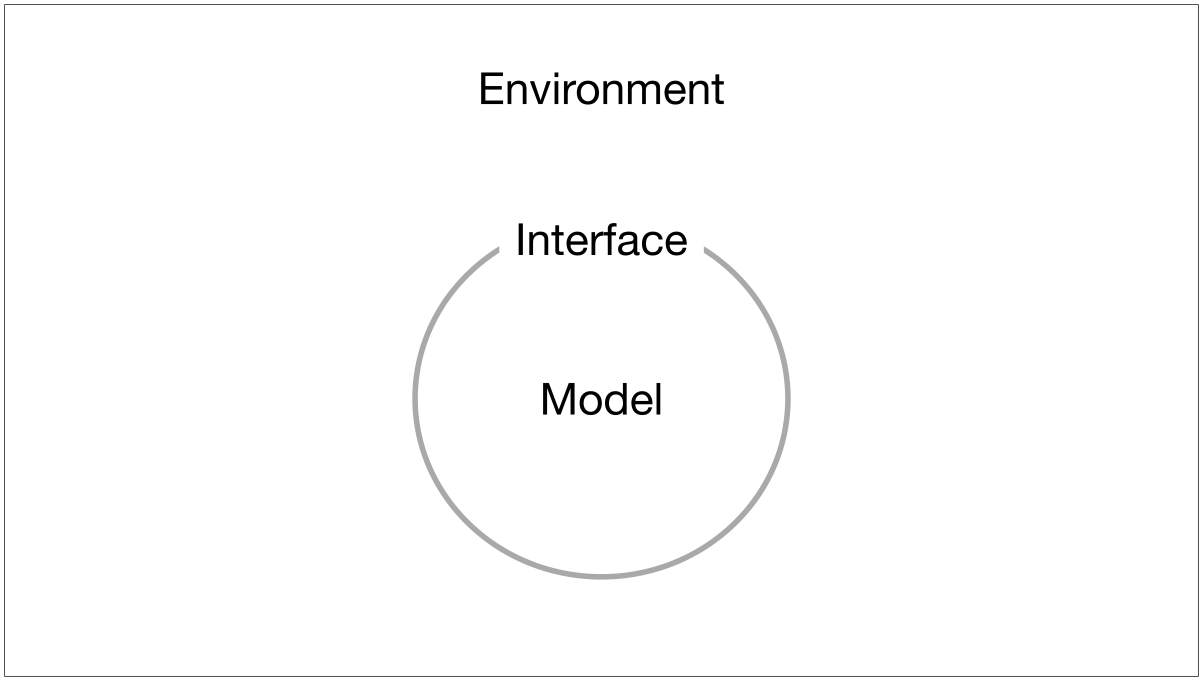
Hoare's abstraction function, like the Church numeral, is a mathematical abstraction. It describes a model which has provable qualities and is often described as "correct," which means it is self-consistent. By omission, it deems the context unimportant.

In software, we don't have the luxury of ignoring context. We need our models to be self-consistent, but we also need them to be *useful* within our given environment. There are countless resources for creating self-consistent abstractions, but few for creating useful abstractions. This is in part because self-consistency is an objective property, while utility is hopelessly subjective.[17]

Self-consistency is necessary, but not sufficient. When evaluating software, we cannot ignore the broader context. We must take advantage of every available perspective. We must be dissatisfied with easy answers. We must be curious.

[17]There is an unfortunate tendency to treat proof that software is "correct" as proof that it is useful. The term "self-consistent" is preferable because it clearly suggests that we are ignoring a broader context.

A Model for Modules



Most software abstractions take the form of a **module**, which consists of a model, an interface, and an environment.

The **model** is a collection of data and functions. The **interface** is the means by which the model and environment interact. The **environment** is everything else: other software components, the users, and the world they exist in.

Models reflect specific facets of their environment. They narrow our attention, giving us the ability to reason about something that is endlessly complex and to change it in predictable ways. Everything the model does not reflect represents an **assumption** that these missing facets are either fixed or irrelevant. If a model can represent invalid states, it must enforce **invariants** that preclude those states.

In the context of our mechanical clock, the model is the inner clockwork, which enforces an invariant relationship between the passage of time and the turning of its gears. The

interface is the clock's hands and dial, as well as the winding mechanism. The clock assumes that it will be regularly wound and that the clockwork won't be subjected to any forces that prevent it from doing its job. If those assumptions are false in our particular context, then the clock is no longer a useful tool for telling time.

Models

When we discuss models, we often turn to physics. The transitions from Ptolemy's epicycles to Kepler's ellipses, from Aristotelian to Newtonian mechanics, and from the hodgepodge of laws about electricity and magnetism to Maxwell's equations, are shining examples of how better models can be transformative. They sweep away something crumbling under the weight of exceptions and bolted-on fixes, replacing it with something simple and clean.

Physics, like all natural sciences, aspires to reason deductively about the world. Deduction maps the environment into the model and then uses the model to infer new facts about the environment. In physics, we observe the world in order to predict what comes next.

In formal deductive reasoning, our conclusions are **necessary**; if they're wrong, it's only because our initial assumptions were wrong. By this measure, physics falls short. Newton's mechanics can't predict the orbit of Mercury, and it's not because we're using the wrong gravitational constant. The mechanics themselves are flawed.

Physics is not, in the strictest sense, deductive. It may never be. But physicists will never stop trying.[18]

Many early computer scientists were trained as physicists, and it shows. When building systems that interacted with the world, they leaned heavily on this deductive approach. The "General Problem Solver" was a software agent created in 1959 that tried to solve

[18]Many flawed theories, like Newton's mechanics, can still be useful in an engineering context. However, this is just a pleasant side effect of physics research, not its purpose.

every problem via "means-end analysis." It would observe the current state, compare it to the desired outcome, and search for a path between the two by simulating intermediate actions. Having found a path, it would then act.

The General Problem Solver was not successful, but similar approaches were tried on successively more powerful machines until the late 1980s, when the AI Winter more or less snuffed out that line of research. Since then, practical use of software has exploded, and deductive models have given way to inductive ones.

Inductive reasoning is, in effect, reasoning by analogy. If two objects occupy the same point within our model, we can observe one to draw conclusions about the other. If rocks and phones look the same within our model, and a rock falls to the ground when it's dropped, we can reasonably assume our phone will too.

The conclusions we draw from inductive reasoning are **contingent**; they're allowed to be wrong. Inductive models are more resilient than their deductive counterparts. Consider a simplistic deductive approach to our dropped phone:

- Our phone is an object.
- All dropped objects fall.
- Therefore, if we drop our phone, it will fall.

This takes a narrow observation, applies rules, and yields a conclusion. The conclusion is reasonable unless we happen to be in an accelerating vehicle, in free-fall, or in any other situation where gravity is not the dominant force. To handle those cases, we'd need to increase the breadth of our observations and the complexity of our rules.

Contrast this with an inductive approach: our phone will do whatever a rock will do. This is both simpler and more robust, but it comes at a cost: we must continuously observe the behavior of our rock. We can never retreat into our own minds.

Empirically, this is a price worth paying; induction is used by every living organism. Consider how the world is seen by a tick.

A tick's decision to try to latch onto a passing animal is based on the presence of heat and butyric acid, found in sweat. If there are sufficient quantities of both, it reflexively makes an attempt.

The tick's model of its environment cannot guarantee a consistent outcome. For any given values of heat and sweat, there are scenarios that lead to success and scenarios that lead to failure. Nevertheless, the tick survives; its simplistic model **satisfices**.[19] It may not be optimal, but it works well enough in practice.

Where the Solver tried to *predict*, the tick only *compares*. The tick is not a brain in a jar; we can only understand it within its environment. The same is true of every living organism, including humans. Our internal models may be more sophisticated than a tick's, but they're still far from deductive.[20]

Likewise, real-world software models don't attempt to predict, only compare. A login process does not verify a person's identity; it only compares the username and password (or some hash thereof) to what's stored within the model. Two people who know that username and password are equivalent within our model; they are effectively the same person. This is a reasonable assumption in some situations, and dangerous in others. In order to judge any model, we must first define its environment.

It's easy to create a deductive model: simply reduce everything to arithmetic or first-order logic. What's difficult is to create a deductive model that is useful. Fred Brook's famous observation that "nine women can't make a baby in one month" is a refutation of a model that is simple, deductive, and wrong. Distrust any abstraction that touches the real world and touts its own logical simplicity.

[19]This term comes from Herbert Simon's book *Sciences of the Artificial*, which is a seminal book on design. Curiously, Simon was also one of the creators of the General Problem Solver. When reading the book, and similar works from the dawn of computing, one must take care to separate the logical positivism that birthed the Solver from the pragmatism that led to *satisfice* being coined.

[20]Philip Agre's *Computation and Human Experience* provides a more detailed critique of the assumptions underlying the General Problem Solver and their prevalence in the AI community over the following decades.

This is not to say that deductive models are worthless; rules engines and Prolog-like mechanisms can be used to good effect in certain domains. However, these models do not belong at the periphery of our systems, and we must not mistake their self-consistency for more than it is.

The models of mathematics don't acknowledge their environment. We can use them to judge whether our software is self-consistent, but not whether it is useful. The models of physics are built atop deductive mechanisms, and aspire towards perfection. The models of software are built atop inductive analogies and aspire only to satisfice.

Invariants

Invariants are required when a model can represent invalid values. Typically, this term describes enforcement of the self-consistency of our models. In our earlier example, the integer set had to ensure that its array is always in sorted order or it wouldn't work properly.

But there is a broader sense to this term, which concerns how the model relates to its environment. We only want our model to represent values that can be found in the environment. For instance, if we store a user's email address as a bare string, countless possible values are not valid addresses: they might be empty, lack an `@` symbol, or contain the collected works of Shakespeare.

Adding regex validation narrows the possible values in our model, but only to strings that superficially resemble email addresses. To validate the address we have to reach outside our process, which makes things considerably harder. We can enforce restrictions on our own model, but we can't enforce a fixed relationship between our model and environment.

We can add a confirmation step to our user registration process, where users are required to acknowledge receipt of a message before we accept their email address as valid. But even then, the email server may go out of service, the account might be stolen, or the user

might forget their password. No one will tell us when the environment drifts away from our model.

At the risk of annoying our users, we can periodically revalidate our model by asking, "is this the best way to contact you?" This only mitigates the problem, but mitigation is the best we can hope for. We must understand how drift between our model and environment can affect us and make sure we avoid the worst of it.

Assumptions

Everything a model omits represents an assumption about the environment: these unrepresented facets are either fixed or irrelevant. A mechanical clock may assume that it will sit on a flat surface, give or take a few degrees. All clocks assume, by omission, that the motion of Jupiter's moons won't affect them.

It's easy to imagine a failure of the former assumption; the clock might be knocked over or placed on a crooked shelf. It's difficult, however, to imagine how the latter assumption would ever fail. Practical models are small and therefore come with an enormous number of assumptions, most of which are completely sound.

The trick, then, is to know which assumptions are worth our attention. We care about assumptions that are invalid or are likely to become so. Judging whether a model is useful in a given environment requires understanding how that environment can change.

We can reduce the burden of fragile assumptions by layering modules with complementary invariants and assumptions. For instance, our mechanical clock assumes a flat surface, and a gyroscopic platform provides a flat surface in all environments. If we put the clock atop that platform, the assumption is no longer our concern.

Broad assumptions mean smaller models, which means simpler code. If we keep modules with similar assumptions grouped together, we can wrap them in a single layer that enforces those assumptions. Abstractions that fail together should stay together.

In some cases this is easy: if a collection isn't thread-safe, we can wrap all access to that collection in a mutex. Likewise, input validation can be performed once at the outermost layers of our software. However, almost all software assumes that memory allocations won't fail, and the JVM won't let us enforce that assumption.

Conventions

If an assumption isn't hidden away by an abstraction layer, it becomes our responsibility to enforce it. Our software's assumption that it can always allocate memory is false unless we make the heap large enough. A mutable data structure's assumption that it has a single owner is false unless we structure our code to ensure it.

These flawed assumptions at the edges of our software are what make abstractions "leaky." Often, the flaw is that we expect too much of our users. The C++ language assumes that programmers can consistently free allocated memory once they're done with it, which hasn't proven true. We can hide away this flawed assumption by adding garbage collection to our runtime, but that comes at a cost some are not willing to pay. Instead, C++ relies on a **convention** known as "resource allocation is initialization" (RAII) that makes it much harder to write code that leaks memory. Likewise, the second chapter of this book describes conventions that help us satisfy the assumptions Clojure drops in our laps.

Even where conventions are required, grouping modules by assumption can be useful. It allows us to more easily remember what conventions are required where.

Conventions are fallible. We sometimes use them because an additional layer of abstraction is too expensive, either to build or to execute. More often, though, we use them because our assumptions are flawed, and we don't know how to automatically enforce them. Conventions are a useful tool, but they're not a solution. We should always aspire for something better.

Interfaces

The **interface** is the means by which the model and environment interact. This encompasses the formal interfaces, such as those defined with `defprotocol`, but also any effects or shared state. Uses of `stdout` and `stdin`, log files, shared atoms, and network requests are all ways that a model can change, or be changed by, its environment.

The interface describes the sense of a module; it encompasses what the model is and also what we expect it to become. As a result, interfaces change much more slowly than models and almost always grow over time. It's far easier to add a method than to take it away.

We should keep our interfaces, like our names, narrow. They should reflect the fundamental qualities of our model and hide away everything else.

Environments

The **environment** is everything that is not the model and interface. Our model does not attempt to define further boundaries. It does not draw distinctions between software and users, or users and the moons of Jupiter. The environment is just a large, homogenous "everything else."

If we applied our model to itself, we'd have to conclude that it assumes these distinctions are irrelevant. This is clearly a bad assumption; for any given model, small facets of the environment are critically important, and the rest are not. To use this model to analyze whether a module is useful, we must draw a line around the things in the environment that are worth paying attention to. This must be done on a case-by-case basis by someone with domain expertise.

That same underdefined quality, however, is what makes this model useful for inductive reasoning. Few modules care about the same environmental facets, but all modules have a model, interface, and environment. This is true for software, architecture, biology, urban planning, and a host of other fields.

Computer science has been grappling with the problem of abstraction for half a century. Other academic traditions have been grappling with it for millennia. This model allows us to map their insights into our domain.

Consequences of our Model

Our goal is to write better modules. At the very least, this means we have to be able to judge whether a given module is useful. If a module isn't useful, we have to decide what steps to take.

While our definition of "module" doesn't directly address either of these, it does lead to some obvious conclusions.

To abstract is to treat things which are different as equivalent

If a tree loses a leaf, we consider it the same tree. This is because our mental model of that tree does not enumerate every leaf; the world may have changed, but our model remains the same. Our understanding of the tree remains valid. If we find a second tree that looks identical within our model, we can apply our understanding of that first tree to the second. By ignoring parts of the world, we can use our existing knowledge in novel situations. This is the essence of inductive reasoning.

In his short story *Funes the Memorious*, Jorge Luis Borges describes a man whose memory is so lossless that he can only recall the past by reliving it, moment by moment. Seeing a dog from two different angles, he cannot find any connection; they are both just collections of endless details, different in every way. The narrator claims that Funes is not really thinking, only remembering:

> To think is to forget a difference, to generalize, to abstract. In the overly replete world of Funes there were nothing but details, almost contiguous details.

By including a facet in our model, we are saying a change to that facet invalidates our past understanding. If we're selling tickets to sporting events, our database will focus on the details of the venue, but largely ignore the players themselves. If we're trying to predict

the outcome of a game, on the other hand, even a single player being injured will force us to reconsider everything; it has become a different team.

Inductive reasoning is about deciding which differences we choose to acknowledge. If plausible changes to some facet will alter our model's identity, we must include it. If not, we should spend our finite mental resources elsewhere.

Models reflect our perception of their environment

There is no objective measure of the importance of a given facet. Our choice to include one facet, and exclude another, reflects the subjective importance we ascribe to each. The model cannot help but reflect its environment; it has no other source of information.[21] By curating what it can and cannot reflect, however, we can distort the environment into something unrecognizable.

A module is useful only if its assumptions are sound

By ignoring, we assume. If our module constructs SQL statements using bare strings from its environment, it ignores the possibility of malicious strings and thus implicitly assumes they won't occur. If it is wrapped in another module which escapes or otherwise validates the strings, this assumption is enforced. If not, then we must hope that our users are, by convention, virtuous. For an internal tool, this might be a sound assumption, but in most cases it is not.

When judging our assumptions, we must consider not only what our environment is, but what it is likely to become. Most disagreements about software are, at their root, disagreements about its present and future environments. When we say software is "over-engineered," we mean that it has too few assumptions; the same effect could have been accomplished with less effort. This means we believe our present environment is

[21] This is only true if we carefully interpret each facet's meaning. If we ask our users if they can fly, we cannot take their response at face value. We might call the field `can-fly?`, but it really only reflects their willingness to claim they can.

narrower than the one assumed by the software and will remain so in the near future. Over-engineering is not a property of our software, but of how we intend to use it.

To judge whether our module is useful, we must first describe the environment as it is and as it will be. Every conversation about software can be made more productive by describing, up front, our subjective understanding of its environment.

To know a module's assumptions, we must know its model

Our model and assumptions are duals of each other; knowing one allows us to infer the other. To understand a module is to know its assumptions. If we cannot understand a module, we cannot know when its assumptions are false. We are forced to use it timidly, confining ourselves to well-worn use cases. Exploration represents an unknowable risk.

This can be an argument against adopting new technology; knowing your software will fail in a given context is better than blindly hoping it won't. Even so, we are constantly drawn to software we don't know well enough to dislike. We see its capabilities, free of any obvious shortcomings, and wish to possess them.

This fixation on possession is centuries old. Arthur alone is able to possess the sword in the stone, and thus is "rightwise king born of all England." Arthur, we're told, holds the sword because he has the qualities of a king: he is wise and regal and from the proper bloodline. But these qualities are largely subjective, and possession is objective. In effect, holding the sword *confers* these qualities onto Arthur.[22]

A similar pattern can be seen in the myth of the philosopher's stone. Representing the highest achievement for any alchemist, it had the power to transmute base metals into gold, heal any illness, and extend life. While for some it was just a metaphor for mastery of alchemy, for most it was a physical object. It was reportedly red and heavier than gold. It could be created and passed from master to pupil. It could be lost and found by some random passer-by. It could be possessed, and used, without being understood.

[22]These sorts of symbolic inversions are further explored in Jean Baudrillard's *Simulacra and Simulation*.

Confidence requires understanding. If we cannot understand our software, it becomes oracular; we may trust or distrust it, but in either case, we do so blindly. We wrap oracles around anything deemed too complex to explain. Oracles deliver our search results and news feeds. Modern machine learning techniques generate oracles.[23] We can possess oracles, but we can never understand them. They turn us all into perpetual novices.

A module cannot prevent itself from being misused

By putting our clock on a gyroscopic platform, we only trade one failure mode for another. The only way we can avoid failure altogether is to know a module's assumptions, and anticipate when they might become a problem. We cannot solve this through further abstraction, and so we must adhere to a convention: only use a module when it is useful.

Unfortunately, it can be very difficult to anticipate failure. Understanding the implementation of our software isn't enough; we have to know what its environment might throw at it.[24] Often, this is learned through experience; the easiest way to know that a failure mode exists is to see it happen. Our job is not simply to understand how software is implemented but to understand the consequences of that implementation.

If a model ignores too much, we can grow our model, replace our model, or narrow its intended use

If we cannot safely ignore a facet of our environment, the most obvious solution is to stop ignoring it. We can do this by either adding that facet to our existing model or by recreating the model from scratch.

[23] Techniques for supervised machine learning are a means of automatically generating inductive models. We describe which inputs should coincide within the model, and the algorithm determines what the model needs to ignore to accomplish this. These models are typically opaque and thus oracular. We can describe their past behavior but not their assumptions or the resulting failure modes.

[24] The "seniority" of an engineer derives more from their ability to predict adverse environments than from mastery of any particular technology.

Alternatively, we can continue to ignore that facet and shift the blame onto the user: our module was never meant to be used in that sort of environment. Our clock was never meant to be put on a ship. Our teletype emulator was never meant to display emoji characters. Our application was never meant to be used by people with more than 32 letters in their surname. In the language of the lean startup, this is known as "firing your customer."

Models are useful because they're small

Over time, models grow. Sometimes we call this adding a feature, other times fixing a bug. In either case, the effect is the same: the model reflects more and more of its environment. This reduces its assumptions, making it more robust, but also makes it harder to understand.

In his story *On Exactitude in Science*, Borges describes a guild of cartographers that creates a 1:1 scale map, which is simply draped over the kingdom:

> The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast map was Useless, and not without some Pitilessness was it, that they delivered it up to the Inclemencies of Sun and Winters.

If we can't fit a model in our head, it has little value. This limit is more a property of our understanding of a model than of the model itself; as we internalize the model, individual facets coalesce into larger, more manageable concepts.[25] For individuals, or even small teams, a growing model doesn't present a problem as long as their understanding grows along with it. It is the model's rate of growth that must be managed, rather than its absolute size.

[25]This process, called "chunking", is described further in George A. Miller's seminal paper *The Magical Number Seven, Plus or Minus Two*.

If we want to teach our model to someone else, however, we must consider its absolute size. Knowing that a model can be comprehended by experts tells us nothing; every model ever created seemed tractable to someone. What matters is whether it can be comprehended by anyone else. If not, we should consider throwing it away.

Starting from scratch is costly

When solving a problem with software, few people begin by designing their own silicon, operating system, and programming language. We avoid reinventing the accreted layers of hardware and software even though they are, for our purposes, over-engineered; they make fewer assumptions than necessary.

These legacy solutions have grown by accretion, making them a reflection of everything they've been used to accomplish, even the things that are no longer relevant. For any specific problem they are too general and too distracted by the past.

And yet, in almost every case we continue to use them. This is because the inefficiencies and unnecessary complexity are a reasonable price to pay for not having to build it ourselves. And unless we are very certain our problem will never change, the generality of these underlying layers makes our software more robust to change.

When replacing software, we should only cut away what we can no longer use.

If a module makes unrealistic assumptions, users can wrap it, create conventions around its use, or discard it

If our clock must be kept level, our users can put it on a gyroscopic platform, never move it from their mantlepiece, or look for another clock. Each of these may be reasonable reactions, but neither the platform nor the mantlepiece are universal solutions. We can make a clock with fragile assumptions because there are many clocks out there, with assumptions and failure modes that are complementary to ours. If we don't solve a user's problem, someone else will.

If a module cannot be discarded, it may destroy what it doesn't reflect

Sometimes an abstraction cannot be easily replaced.

In rural France, before the rule of Napoleon, land ownership was a complex affair. Common pastures were shared within a village, or between villages, according to need. The fruit on a tree belonged to whichever family planted it, regardless of who owned the land. Fruit fallen from the tree belonged to whoever gathered it. If a tree was felled, the trunk belonged to the family, the branches to their neighbors, and the leaves and twigs to whoever gathered them. Where boundaries existed, they would be regularly adjusted in response to changing circumstances. Where rules were fixed, their exact nature would vary from village to village.

While perfectly clear to each villager, this situation was a cacophonous mess for the government officials. They could only tax what they could measure, and the rural model of ownership defied easy measurement. Their solution was simple: they sent in the surveyors, who drew maps assigning a single owner to each parcel of land, and a year later the tax collectors came calling.

As a result, ownership within these villages became more rigid. If the map said a family owned a parcel of land, they paid taxes on it, which made sharing the land an expensive proposition. By basing taxation on a simplistic model, the French government forced their citizens to conform to that model.[26]

Mandatory abstractions are coercive; if the environment doesn't fit their assumptions, we're forced to create an environment that does. These assumptions shape the lives of the people who use them, and in time they begin to feel obvious, just a natural reflection of how things are meant to be. Slowly, they fade from view.

Any software chosen for us is coercive. Enterprise software is sold to one person and used by many others. If the software makes unrealistic assumptions, the users cannot easily

[26] Many other examples of this phenomenon can be found in James L. Scott's *Seeing Like a State*.

replace it. They are forced to shape themselves to its needs. As creators of software, we cannot afford to ignore the impact our models have on their environment.

Software would be easy if things never changed

Software must change with its environment. If it doesn't, it will eventually become useless. This can only be avoided by choosing an environment that is naturally stable or by simulating stability by wrapping our software in further abstractions and conventions.

Libraries of mathematical routines, often written in FORTRAN, have survived with minimal changes for decades. This is because math doesn't change very quickly. The underlying hardware, however, does. The LINPACK library, written in the 1970s, was targeted at vector supercomputers like those made by Cray. In the early 1990s, LINPACK was supplanted by LAPACK, which targets modern cache-based architectures. Even so, these libraries are among the most intrinsically stable software ever written.

Everywhere else, software survives unchanged only because of our continued efforts. Batch data-processing software written in COBOL still runs on mainframes because the company has built layers of software and institutional conventions around it. The original UNIX tools are useful for data analysis only where we have avoided the use of binary formats or nested data representations like JSON or EDN. Depending on your perspective, these sorts of efforts may be pragmatic or decades-long examples of the sunk cost fallacy.

In either case, it is clear that vanishingly little software retains value without continuous, sometimes drastic, change. Given this, we should stop drawing comparisons between software and civil engineering. A bridge is a solution to a largely static problem; it may undergo maintenance, but change is almost always accomplished by building a new

bridge. Software, on the other hand, is a solution to an ever-changing problem.[27]

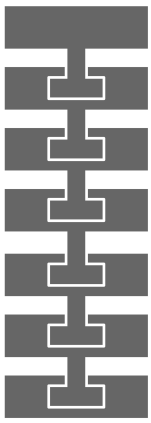
To create software of lasting value, then, we must minimize the effects of change wherever possible. The environment for any software component includes the users, the problem domain, and other software. We cannot control our users, and we often have limited control over how our problem domain evolves. We can, however, control how changing one software component affects the others.

[27] All metaphors relating to physical construction are fundamentally flawed. For one, software has no spatial constraints; no matter how closely we've packed our software, we can always add something in between. But if we must choose a metaphor, city planning is far better than bridges. A city is in constant flux, too large to understand in its entirety, and its residents will always demand more than they have. Anyone wishing to pursue this line of inquiry should read Jane Jacob's *Death and Life of Great American Cities* and Kevin Lynch's *Image of the City*.

Systems of Modules

Whenever we change a piece of code, we risk invalidating assumptions made elsewhere in our codebase. Our modules succeed by minimizing how often this occurs. Even small systems are too large to understand top to bottom, so we cannot simply consider all the other code whenever we make a change. We want to limit the unintended effects of our changes, even if we don't fully understand the environment.

There are two fundamental strategies for accomplishing this. We can build a **principled** system, which has predictable relationships between its modules. Alternately, we can build an **adaptable** system, which has sparse and flexible relationships between its modules.



A principled tower

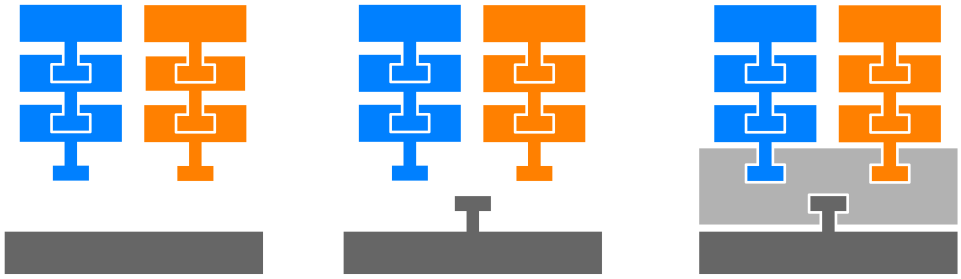
A principled system minimizes internal indirection and is usually structured as a hierarchy. The implementation of each component is guided by the central design principles. These principles, applied from the top down, allow each component to make broader assumptions. This makes each component smaller and often faster, since there are consistent methods used throughout.

This gives the code a minimal quality, even a certain elegance. In the words of Saint-Exupéry, there is nothing more to take away. Hierarchies can also be learned gradually; we can decompose from the top down, beginning at the root and peeling away layers to reveal the underlying implementation.

These systems, however, are highly interdependent. If work is split between children, each implicitly assumes the existence of all the others. They lean upon each other like a house of cards. If even a single one is out of place, the entire system can come crashing

down.

Each piece of a principled system serves a single purpose. If two such systems share a component, then neither can shape it. The shared component must strike a balance, guided by its own design principles. Without a coherent principle throughout, indirection forms. An interface is born.



As components on both sides of the interface come and go, the interface remains. The permanence of an interface enables the surrounding code to change. It allows our systems to adapt.

An adaptable system has a high degree of internal indirection and is usually structured as a graph.[28]. Each component is purposefully blind to the internals of adjacent components, which leads to redundancies. This makes each component larger, and often less efficient.

A graph is a much more flexible model, but it resists incremental decomposition. It has no clear root, leaves, or layers. Without a predictable structure, exhaustive exploration is the only way to discover where or how something is accomplished. There are no organizing principles, no commanding heights from which we can perceive and control the system.

[28]The interplay between the hierarchy and graph is a recurring theme in the works of the post-modern critical theorists Gilles Deleuze and Felix Guattari, notably in their book *A Thousand Plateaus*. This is a challenging book, not least because of the lack of shared vocabulary with computer science; they refer to tree-like structures as "arborescent" and graph-like structures as "rhizomatic". A determined reader, however, will find it rich and rewarding.

These approaches are contrasted in Christopher Alexander's *Notes on the Synthesis of Form*. In it, he discusses different traditions around building homes, drawing a distinction between what he calls selfconscious and unselfconscious cultures.

An **unselfconscious** culture, he says, has no word for "architect"; each person builds their own home. The design is refined over generations, and construction is taught using direct demonstration. While simple and sometimes crude, they reflect the constraints and variation of their environment.

In a **selfconscious** culture, the design and construction of homes is a specialized task. It is taught in schools using abstract principles. These designs are often complex and ornate, and they reflect the vision of the architect.

The structures of an unselfconscious culture are adaptable; they reflect the present needs of the inhabitants. If an igloo grows too warm, someone can poke a hole in the wall. When it grows too cold, the hole can be filled in. There is a constant awareness of the environment and constant adaptation as it changes. Such structures tend to be only large enough to hold a single family.

The structures of a selfconscious culture are principled; they not meant to change. They may reflect their environment by building atop solid earth or by orienting the windows north/south in warmer climates. Alternatively, they may simply pour a concrete foundation and install air conditioning. If the environment changes, the structure is hardened against the change rather than adapting to it. Some principled structures, like skyscrapers or stadiums, can hold thousands of people.

Clojure is a mostly unprincipled language. Its few principles, such as immutability, promote the creation of adaptable software. Rich Hickey's own definition of 'simplicity,' from his talk *Simple Made Easy*, describes an adaptable system: software components that are not entwined.

This suggests that our systems should be built from independent components. We can create an ecosystem of functions and modules, all separated by strong indirection, and combine them as needed.

An adaptable chain

But when an interface only serves a single purpose, indirection is hard to maintain. An interface pulled in many directions is intrinsically stable, but an interface pulled in a single direction tends to shift. Over time, the code on either side of the interface will grow interdependent. The interface itself will become vestigial, serving only to mislead future developers.

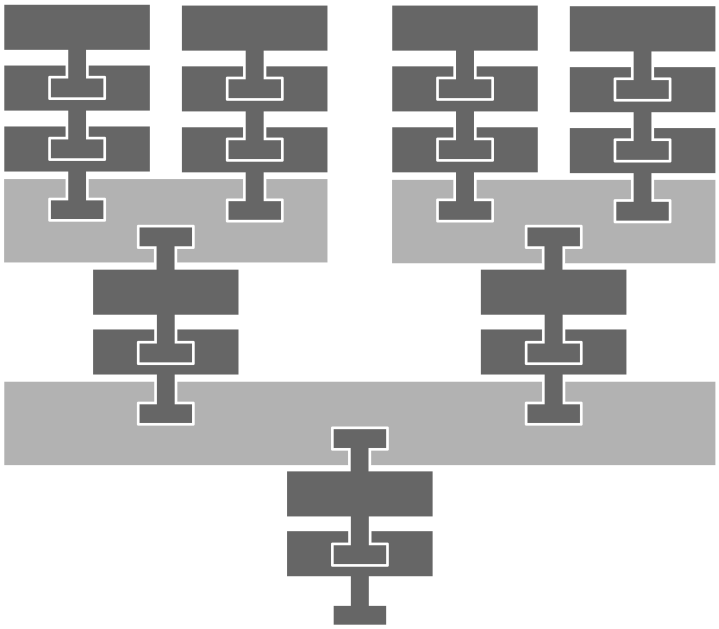
This is demonstrated by the mitochondrion, known to students everywhere as "the powerhouse of the cell." A few billion years ago the mitochondrion was an independent organism, but within the stable environment of the cell, that independence was an inefficiency. Today, it is unable to exist outside the cell and produces many times more energy than it needs itself. It has become just another interdependent part of a principled whole.

Even if we could maintain this indirection, it's not clear we'd want to. Crossing an interface puts us in a new context, forcing us to relearn our surroundings. If both sides of the interface share a single purpose, this is an enormous cost with no obvious benefit.

Principled components allow us to explore within a uniform context without any need to reorient ourselves. This uniformity, however, makes them fragile. They are constructed towards a fixed purpose and cannot be easily reoriented. We cannot shift the Arc de Triomphe without rebuilding the streets and buildings that radiate outward from it. The core assumptions are foundational, and even a small change invalidates everything.

And so we do not want a system that is wholly principled. We want a collection of principled components, built to be discarded, separated by interfaces that are built to

last.



Short-lived principled components separated by long-lived interfaces

These systems, which are able to replace every part of themselves but still retain their fundamental character, are often referred to as "complex adaptive systems." They are present everywhere in the world, from the human cell to global markets.[29]

Consider the Monarch and Viceroy butterflies. The Monarch butterfly is toxic and avoided by predators. The Viceroy butterfly is not toxic, but closely resembles the Monarch. Clearly, the Viceroy assumes the existence of the Monarch, which makes its existence precarious. If the Monarch were to disappear, the Viceroy wouldn't change its appearance to some other inedible butterfly – it would be eaten out of existence.

It is the ecosystem, not the organism, that adapts to change. If an organism makes an invalid assumption, it disappears and its niche is filled by something else. These roles

[29]So far, researchers have been able to draw analogies between a wide variety of real-world phenomena but unable to describe a common generative mechanism that connects them. As such, the literature around complexity is interesting but only indirectly applicable to software. Anyone wishing to go deeper on this subject should start with *Complex Adaptive Systems* by Miller and Page.

are fungible because the organisms consume and emit the same resources; they share a common interface.

For an interface to disappear, every participant on one side must disappear. A symbiotic relationship between two species is relatively fragile; the extinction of either species will render the relationship moot. The carbon-based building blocks of life, however, are foundational to millions of species and have persisted for billions of years. Likewise, the REST protocol for our web application will vanish without customers, but TCP and POSIX will likely last for centuries.

Where possible, we should avoid creating foundational interfaces. If an interface only touches our own software, we can learn from our design mistakes and move past them. Interfaces calcify when they are exposed to the world, which can allow our mistakes to outlive us. If we must create a foundational interface, we should first allow it to mature within our own code.

We should build our software from principled components wherever possible, separated by interfaces where necessary. Modules that share common assumptions should live in the same component, and modules with dissimilar assumptions should be kept separate. This keeps the effects of changes small and predictable.

If our problem domain is stable and uniform, it has little need for indirection. Any domain, when poorly understood, seems to fit this description. As an industry, we are biased towards simple solutions born from incuriosity. What we don't know, we fill in with blind optimism.

Given this, Clojure's opposing bias towards the adaptable approach seems reasonable; it saves us from ourselves. But we cannot find the right balance without a deep understanding of our software, the environment in which it exists, and what they both may become.

Composition

Composition is the combination of separate abstractions to create a new abstraction. These abstractions, once combined, begin to define each other's environment. Through composition, we create the context by which each individual piece can be judged. It allows us to discuss specific trade-offs, rather than the concept of a trade-off. Composition is applied abstraction.

In the classic mathematical sense, composition is the combination of functions, and most composition in Clojure does involve functions. The ultimate goal of software composition, however, is not simply to define new functions but to define **processes** that **pull** data from their environment, **transform** that data, and **push** the result back into their environment.

A process has execution isolation (*when* it runs), has data isolation (*where* it runs), and is sequential. It corresponds to, and is named for, a process in early operating systems that lacked support for threads. However, it can also describe threads in a modern operating system, a chain of asynchronous callbacks, or any other mechanism which shares these properties.[30]

The process, as defined here, is the smallest unit of standalone computation. If software does not perform all steps (pull, transform, and push) at least once, it can only be useful when combined with other software. Consider the UNIX `yes` command, reproduced here:

[30]Such mechanisms include Erlang's processes, Carl Hewitt's actors, and Smalltalk-72's objects, all of which communicate via asynchronous message passing.

```
(defn yes []
  (loop []
    (println "y")
    (recur)))
```

This utility neither pulls nor transforms data. It will simply push an endlessly repeating stream of `"y\n"` to `stdout` until the process is killed. This is useful when placed upstream of another process with interactive prompts but useless by itself.

Similarly, consider a function that reproduces the functionality of `/dev/null`:

```
(defn dev-null []
  (loop []
    (read-line)
    (recur)))
```

This utility endlessly pulls from `stdin` and then drops the data on the floor. This is only helpful when an upstream process has useful effects beyond what it pushes to `stdout`.

The `cat` command pulls the contents of one or more files and pushes them to `stdout` without any intermediate transformation:

```
(require '[clojure.java.io :as io])

(defn cat [& filenames]
  (doseq [f filenames]
    (doseq [l (->> f io/reader line-seq)]
      (println l))))
```

If we place this upstream of a process that only pulls from `stdin`, that process can now indirectly access the filesystem. Moving data is part of any nontrivial computation, but

not very useful on its own. We can use `cat` to view the contents of a file, but only when it's fed into a teletype emulator.

A process that doesn't produce data is obviously of limited use. A process that doesn't transform data is only useful where it makes data available to other processes. But what about a process that doesn't consume data?

```
(defn yes [expletive]
  (loop []
    (println expletive)
    (recur)))
```

This version of `yes` is parameterized and could be seen as transforming `expletive` into an infinite recurrence of itself. However, this infinite stream is not particularly interesting; once we've seen one line, we've seen them all. In general, the output of a process is only as interesting as its inputs.[31] Where the data pulled in by a process is *eventually* available, parameters must be *immediately* available. When using our process in isolation, the parameters are limited to the information literally at our fingertips. If we are the only source of information, our software can only tell us variations on what we already know.

Software applications comprise one or more processes. The building blocks for our software may not constitute a full process; they're often more useful when they don't. But to create software that is useful on its own, we must construct whole processes, and we often combine those processes into a larger system.

[31] There are a few exceptions to this rule, largely in the domain of mathematics. A program that calculates the digits of pi takes no inputs but can emit an endless stream of data. Likewise, pseudorandom number generators or visualizations of fractal geometry can generate large amounts of data given a small input. This data, however, is only useful when fed into other processes.

A Unit of Computation

Processes are a ubiquitous concept dating back to the earliest days of computing. It is the smallest piece of code which can be understood on its own, in part because it does something useful on its own, but also because it provides strong indirection between itself and its environment. If we cannot understand the system as a whole, we can at least understand it one process at a time.

Processes provide (some) data isolation

A process can only access data that is globally visible or passed in as a parameter. Once initialized, its universe is bounded and fixed, consisting of both immutable values and mutable references. Communication between processes is only possible via shared references, and any change to such a reference is called an **effect**.[32] These references are often hidden behind an interface, which provides structure around how and when effects occur.

For a reference to be safely shared, we must be careful in our interactions. Without coordination, simultaneous access can lead to data inconsistencies or even permanent corruption. But even if our shared data structure is thread-safe, updating the internal reference is rarely enough on its own. For our update to trigger an action in another process, we must signal that an update has occurred. This is why inter-process communication often uses queues, which provide both thread-safety and signaling.

Mutability may be necessary at the edges of our processes, but it should be avoided elsewhere. If we pass a mutable data structure into a function, we cannot prevent that function from sharing it with another process, creating a new edge in our system. Even if we know a particular function doesn't touch another process, that may change in the

[32] A more common term for this is 'side effect.' That term implies that our effects are incidental and avoidable, but truly they are a necessary component of our software.

future. This implicit process composition can make it near-impossible to reason about the system as a whole. As a rule, we should only use internal mutable data structures in principled components, which have predictable behavior and limited capacity for change.

Processes provide (some) execution isolation

Processes run sequentially; each operation is executed in order, one at a time. When reasoning about a process, we rely heavily on the operations having a deterministic order. When we invoke a function, we place its operations ahead of all the others. Within our process, function invocation has an immediate effect.

The same cannot be said of inter-process communication. We can update a shared reference, and even signal that the reference was updated, but that is the extent of our power. We cannot control when or how another process will react to that new information.

But while processes cannot control one another, they are still interdependent. To pull the contents of a file into our process, we must request it from the operating system and then wait for the signal that it's available. A process cannot force us to react to its effects, but by delaying the effects it *can* force us to wait for them.

Like function invocation, by waiting for data we place its arrival ahead of other operations. Unlike function invocation, our process is a passive participant. When a process is active, we can reason about it in isolation. When it's paused, we must consider the surrounding processes that will allow it to continue.

To read a file, we must traverse a cache, an I/O scheduler which prioritizes and deduplicates pending reads, a controller on the physical disk, and finally the storage medium itself. These are directly responsible for fetching data, but other processes using the file system may be indirectly responsible for delays. Holding the entire system in our head can be difficult at best.

This is only necessary, however, when our process waits *too long*. If we never wait longer than we're willing to, there's no need to consider the system as a whole. The simplest way to achieve this is to have low expectations. Modern hardware is far more capable than modern software would suggest, but most of us choose to be content with the performance we have. This can be a helpful strategy for coping with the complexity of modern systems.

A complementary strategy is to define timeouts, which prevent a process from ever waiting too long. If a timeout elapses, we don't try to understand why; we just try to recover. As long as timeouts don't occur too often, it's much simpler to let a system sometimes fail than to chase down every unexplained pause.

These strategies allow us to consider each process in isolation, so long as performance isn't a primary concern and some failures are acceptable. In more stringent domains we must fight to keep the system as simple as possible, so that it can fit in our head. Any system that exceeds our understanding will inevitably grow a bit slow and flaky, no matter what the design specification says.

Example: a REPL

```
(defn repl []
  (loop []
    (-> (read)
      eval
      print)
    (recur)))
```

The REPL is a simple, but complete, process: it pulls in expressions via `read`, transforms them via `eval`, and pushes the result out via `print`. Normally, it will spend most of its time waiting on the result of `read`; the computer can perform the eval-print-loop steps much

faster than we can type in new expressions. However, it's easy to craft an expression which takes more time to `eval` than it did to type:

```
(print (eval '(reduce + (range 1e9))))
```

Likewise, we can turn `print` into our bottleneck:

```
(print (eval '(range 1e9)))
```

Since `range` returns a lazy sequence, `eval` will return immediately, leaving `print` to push a billion numbers to whatever process is responsible for displaying the result. Since `print` can likely send numbers faster than they can be displayed, `print` will spend most of its time waiting for a signal that the downstream process can accept more data.

While it's convenient to talk about "pulling" and "pushing" data, both operations tend to require bidirectional communication. To pull data, a process must send information about what data it wishes to receive, or at least signal that it is ready to receive more information over a pre-existing channel. To push data, a process must confirm that downstream processes have the capacity to process this new data. There are exceptions where the data is of a predetermined type and bounded in size, but such guarantees are rare in practice.

Example: a web service

```
(defn handler [request]
  (-> request
    request->query
    query-db!
    result->response))
```

This function implements the API specified by the Ring specification, transforming data representing an HTTP request into data representing an HTTP response. Our `handler` is a parameter passed into a larger process, which is responsible for pulling in the encoded request and pushing out an encoded response. But these are not the only edges in our process; `handler` also pulls data from an external database. Our process follows these steps in sequence:

- **pull** in an encoded request from the client
- **transform** the encoded request into a Ring request
- the handler is invoked
- **transform** the Ring request into a database query
- **pull** the result of that query from the database
- **transform** the database result into a Ring response
- the handler returns
- **transform** the Ring response into an encoded response
- **push** the encoded response to the client

A Ring webserver is a **framework**; it invokes our code rather than being invoked by it. This frees us from having to consider the complexity of effects when writing our code, but it also makes it more difficult to understand what our software is doing in production. To reason about the operational properties of our code, we must understand the process that surrounds it.

Example: a frontend application

```
(on-click refresh-button
  (fn []
    (query-service
      (fn [data]
        (update-dom data)))))
```

This code registers a callback on a "refresh" button. Each click will fire off a request, and the response will be used to update the data shown in the browser. If we click the button multiple times, it will fire off multiple requests, which may execute concurrently. This does not represent a single process, but rather a mechanism that **spawns** processes. Each time the callback is triggered a process starts, executes once, and exits.

Since there's no real value in allowing concurrent refresh operations, we might decide to preclude them:

```
(on-click refresh-button
  (fn []
    (disable! refresh-button)
    (query-service
      (fn [data]
        (update-dom data)
        (enable! refresh-button)))))
```

This *is* a process because a second click cannot occur until the first has been fully handled. In a process, we have to decide what to do when our environment demands more than we can provide. Here, we've signaled to the environment (our user) that we will ignore any clicks while a request is still in flight.

To create a fully robust process, we must also decide what to do when the backend service is unavailable. We might retry the request, either indefinitely or up to a maximum

number of retries. We might perform our retries at fixed intervals, or increase the intervals using exponential backoff. Alternatively, we might simply display a message to the user that the refresh failed.

These strategies, describing what our process will do when its environment provides too much or too little, are called an **execution model**. A process with a well-defined execution model can be safely considered in isolation.

Queues by themselves do not provide isolation. Queues couple the execution of processes and, by default, allow one process to block the other indefinitely. If neither places limits on how long it will wait to push or pull data, they cannot be understood separately. Such processes share a single execution model.

In some cases this is unavoidable. Limits are tied to the specifics of our application and do not generalize. Since the components of our file system sit beneath many such applications, they cannot define their own timeouts. All they can do is wait indefinitely and rely on someone else to make the hard choices. Those choices, however, must be made somewhere, and if we allow an execution model to span too many processes, it will quickly exceed our understanding.

Building a Process

A process is composed of pull, push, and transform **phases**. These phases should be kept separate until the last possible moment. Consider this parameterized REPL:

```
(defn repl [read eval print]
  (loop []
    (->> (read)
      eval
      print)
  (recur)))
```

The source from which we `read`, the sink to which we `print`, and our evaluation strategy can all be understood in isolation. There would be no purpose to asking for a `read-eval` or `eval-print` parameter. Their composition is only useful at the apex of our process definition.

These phases are separable because they serve different roles. The push and pull phases are **operational**: they deal with code in motion and define the limits of our process. The transform phase is **functional**: it deals with code at rest and defines the purpose of our process. The push and pull phases enforce invariants that can only be designed and judged given a specific context. The transform phase, wrapped in those invariants, can safely ignore much of that context.

Consider Clojure's `sort` function. According to its documentation, it returns the elements of the input collection in sorted order, but this is only partially true. If we try to run `(sort (range 1e12))`, it will throw an `OutOfMemoryException`. Since `sort` only works if certain conventions are followed, we must consider the context in which it's used.

Contrast this with the GNU `sort` utility. It explicitly protects against this failure mode by pulling in chunks of data, sorting each chunk, and spilling the result to disk. Once

the input is exhausted, it will merge-sort all the chunks together, pushing the result downstream.[33]

Both examples use the same sorting mechanism, but only GNU's `sort` is explicit about what surrounds it. If we fail to do the same when using Clojure's `sort`, we create a leaky abstraction; our implicit assumptions become everyone's concern.

Leaky abstractions are fine, so long as they sit within a principled component that shares their assumptions. If our code is meant to load a configuration file, for instance, we may `slurp` it into memory rather than read the file incrementally. Once we've done that, there's no real harm in calling Clojure's `sort`; our code is already fragile in the face of oversized inputs.

But a principled component cannot span multiple processes; principled components rely on weak internal indirection, and process boundaries provide strong indirection. Leaky abstractions may be fine in the right context, but leaky processes are always dangerous. Our processes may be larger than GNU's `sort`, but at the edges we must always enforce the assumptions within.

Pulling Data

The pull phase acquires data from outside the process and verifies that it is properly shaped and sized. It also defines what happens when the data is invalid or unavailable.

All too often, however, we focus on acquiring data and ignore the rest. Consider this function, which yields a lazy sequence over the lines of a text file:

[33] Of course, this still leaves open the possibility that we might run out of disk space, but that resource is at least several orders of magnitude less scarce.

```
(require '[clojure.java.io :as io])

(defn file-line-seq [filename]
  (->> filename
    io/reader
    line-seq))
```

By simply passing this along to our transform phase, we ignore a number of failure modes:

- The size of each line is unbounded, except by the fact that a `String` cannot contain more than two billion characters. If we don't have four gigabytes of free memory each time we call `next` on our `seq`, we risk an `OutOfMemoryException`.
- If the text file contains encoded data such as JSON or EDN, the encoding may be malformed.
- Any time we touch the file, an `IOException` may be thrown.

By ignoring these scenarios, we make them fatal. Our process will simply end, possibly logging an error, forcing the surrounding processes to pick up the pieces.

Sometimes this is fine. If a configuration file is excessively large, malformed, or unavailable, all we can do is fail and wait for someone to debug the issue. But configuration files are unique inputs in many ways:

- Changes to the configuration data and the code that consumes it are often reviewed by the same people.
- As ancillary data, a configuration format that is fundamentally limited in both size and shape doesn't also limit the usefulness of a process.
- Configuration data is read during a deployment process, when there are people on hand to detect and respond to failures.

Most input data is not inherently limited in its origin, shape, and size. Most new input data does not have its effects carefully monitored by a trained engineer. In most cases, we must make our processes intrinsically robust.[34]

In a robust process, the pull phase should invoke the transform phase. This gives us greater flexibility in how we respond to errors; different scenarios may call for different kinds of transforms. If our pull phase simply yields a `lazy-seq`, this relationship is inverted, and our control flow is greatly constrained.

More generally, by consuming a `lazy-seq` that performs effects, we're forced to make operational decisions. When an error occurs inside `next`, we have three choices: we can retry, truncate the `seq`, or allow the exception to leak out. Since we can't retry forever, this can only be a supplementary solution. Truncation, at best, allows us to infer that *something* failed, and the conflation of all possible error modes is an operational decision in itself. And if we allow exceptions to bubble out, we must confront them directly.

Our transform phase is, by definition, code free from operational concerns. If we simply compose over lazy effects, we allow our pull phase to encompass the core of our process. In such a process, nothing can be considered in isolation.

Transforming Data

There are only three things we can do with data. We can **accrete** data by adding it to an existing collection, **reduce** data by discarding information from an existing collection, or **reshape** data by placing it in a different kind of collection.

We accrete data when we don't know enough to do anything else or when we want to work with a larger batch of data.

[34]Alternatively, we can make our systems intrinsically robust to the loss of processes, using something like Erlang's OTP framework. This approach, however, requires significant support at the language or runtime level to ensure errors are captured and propagated. Since no such mechanism currently exists in the Clojure ecosystem, we will not spend any time exploring this approach.

We reduce data when inputs that yield the same output are interchangeable. When we compute a sum, we imply that `[3 3]`, `[1 2 3]`, and `[6]` look the same to us. When we look up `:callisto` in a map, we imply that `{:callisto 1, :io 3}` and `{:callisto 1, :europa 5}` aren't meaningfully different. This is abstraction; we are treating different values as equivalent.

All the lessons of the previous chapter apply here. Data analysts tend to avoid simplistic metrics like mean and variance because datasets that intersect on these metrics can have important differences. The reduction of data requires more care than the rest of our software; it contributes most of the value and most of the risk.

Lastly, we reshape data when we want to make it easier to accrete and reduce. When we store large amounts of data, we prefer a database to a collection of flat files, even though both allow the same fundamental operations. This is because the database gives us random access to the data, which in most cases matters a great deal. Reshaping is *not* abstraction, because it is motivated by differences that *do* matter.

The study of data structures and algorithms is the study of the strengths and weaknesses of different data shapes. The importance of this is often overstated; in most cases, it suffices to understand the tradeoffs of Clojure's core data structures. It is not, however, something that can be ignored.

Data should never be implicitly reshaped. If our function needs a set, it should demand a set. This allows others to judge whether our code is a good fit for their problem and prevents pathological situations where a collection is reshaped repeatedly rather than once.

We should also try to keep our accretions and reductions separate. Sometimes this is impossible; adding values to a set both accretes elements and loses information about ordering and duplicate values. But if they are separable, we should expose them as pieces to be composed, rather than as an indivisible unit.

Pushing Data

The output of the transform phase is not just data, but rather a **descriptor** of the effects that the process should perform. Most often, this is a description of what data should be pushed to other processes and how.[35] The push phase acts upon that descriptor.

In the simplest case, the descriptor is only the data that should be pushed. When we pass `"hello world!"` to `println`, we are giving a literal description of what `println` should write to `stdout`. Even for more complex effects, this is still true:

```
{:url    "http://example.com"
 :method :post
 :body   "hello world!"}
```

This is not a literal description of an encoded HTTP request, but there is a direct correspondence between each part of this map and the encoded request. However, unlike `println`, a typical HTTP client library allows us to also specify *how* the request is made. This tells the client that any 3XX redirect responses should be automatically followed, so long as the chain of redirects isn't too long:

```
{:url              "http://example.com"
 :method           :post
 :body             "hello world!"
 :follow-redirects? true
 :max-redirects    99}
```

The meaning of each field in this descriptor may seem self-explanatory, but we should not fool ourselves into thinking our descriptor has well-defined semantics. Data is just

[35] A descriptor may also describe a pull effect, such as an HTTP `GET` request, but the design considerations for both are largely the same.

data; it doesn't have intrinsic meaning. The semantics of our data are defined by the effects it produces when passed into our functions. These effects should be predictable whenever possible, but data cannot prevent itself from being interpreted in surprising ways.

The push phase begins wherever our functions perform effects. Put another way, it begins wherever the meaning of our data is defined. Sometimes this is a straightforward execution of a descriptor:

```
(println "hello world!")
```

It's sometimes useful, however, to close over the descriptor and return a function which can be invoked at our leisure:

```
(defn printer [s]
  (fn []
    (println s)))
```

We cannot introspect on a `printer`; to know what it does, we have to invoke it and see what happens. We cannot alter what's printed, only what comes before or after:

```
(defn prepend [printer prefix]
  (fn []
    (println prefix)
    (printer)))

(defn append [printer suffix]
  (fn []
    (printer)
    (println suffix)))
```

A function cannot be reduced or reshaped; it can only accrete. By exchanging a string for a `printer`, we gain semantics but lose almost everything else. We should seek to delay this as long as possible.[36]

A process is composed of operational phases at the edges and a functional phase in the middle. The operational phases guard against the pathological behaviors found in production; they interact with the environment and enforce invariants on how it can affect the process. Unless we are very good at predicting and simulating these pathologies, the only true test is to deploy them and see what happens.

But if they're well crafted, these operational phases allow us to understand and test the functional phase in isolation. This plays to the strengths of Clojure and its REPL-driven development process. We should try to keep most of our code in the middle and as little as possible at the edges.

[36] The effects of some functions can be altered using dynamic state; the `println` function, for instance, can be redirected by binding a new `Writer` to `*out*`. This is only necessary, however, when we've discarded our data prematurely. Wherever possible, we should alter our effects by altering their descriptor.

Unfortunately, it can be easy to lose track of where we are. We have no way to know if a function or `lazy-seq` performs effects, which makes it easy for them to leak into the functional core of a process. It can be helpful to keep the operational and functional phases in separate namespaces. By only allowing functional namespaces to reference other functional namespaces, we guard against the inward creep of operational concerns.

Composing Processes

For many applications, a single process does not suffice. The application may require separate concurrently operating parts, or it may be easier to understand when structured that way. To accomplish this, we compose processes into a larger **system**. The graph describing the set of active processes and the relationships between them is the **topology** of our system.

A process is a mostly opaque thing. We cannot directly alter it or see inside it. All we can do is share data and wait for it to do the same. This means that, unlike most things in Clojure, processes are not values. We can only refer to them through a layer of indirection, using a process **identifier**, or communicate with them via a **channel**.

In a static process topology, channels usually suffice. Consider a simple `bash` pipeline:

```
cat moons | grep 'callisto' | less
```

This pushes the contents of the `moons` file into `grep`, which filters out the lines containing 'callisto' and pushes them along to `less`, which pushes them to the terminal. Communication in this pipeline is anonymous and externally configured. No one knows who their neighbors are; they just interact with the `stdin` and `stdout` channels they were initialized with.

This approach, however, is only possible if our system never changes. If we want to create new channels, we'll need some way to identify processes in our system. Unique identifiers, however, are only possible if our processes never die. More often, our identifiers are connected to processes through a layer of indirection. This identifier is then mapped onto a specific process via **resolution**.

The most familiar example of this is DNS resolution, which maps domain names onto IP addresses. DNS provides a one-to-many mapping between domain names and IP

addresses, allowing for basic load balancing and failover behavior. This is true of most resolution mechanisms designed for larger systems. In practice, there is little difference between resolution and **discovery**, which fetches a list of processes able to provide a particular service.

It's often useful to create a **router**, which provides indirection by exposing a single channel and distributing the data on that channel across multiple processes. This pattern is ubiquitous at every level of real-world systems. Even a thread pool is a router; functions are placed on a shared queue and distributed to threads which execute them.

Beyond these basic concepts, it's difficult to describe process composition in the abstract. Processes may share a common structure, but system topologies vary widely depending on the application. A taxonomy of system design patterns is beyond the scope of this book.

But if we examine the messages between processes, one last pattern emerges: most data pushed to another process is a command describing an effect they must perform. Often this command is passed along a chain of processes in a variety of forms until it arrives at a process where the effect can realized.

When we call `(println "hello world!")`, the encoded bytes of our string are pushed from process to process until they arrive in a desktop application. In that application, the string is rendered into a graphical representation of itself. That representation is then pushed to the graphics driver, which passes it along to the graphics hardware, which finally renders it to our display.

Likewise, when we make an HTTP `POST` request, our goal is not merely to send the bytes of our request to another machine; we want our request to be interpreted by that machine and acted upon. Often this requires that our request, or some variation of it, be forwarded to other machines we cannot directly access.

The proximate goal of any push is communication, but the ultimate goal is the completion of a **task**. A task begins when a command enters our system. It might be started

by a keystroke, a packet from the network, or the creation of the system. A task ends when the consequences of an effect are uncertain. We can display text on a screen, but we can't dictate how the user will respond; we can write a value to the database, but we can't dictate if it will ever be read.

It is useful and often necessary for the completion of a task to be acknowledged back up the chain of processes that propagated the command. This is because processes and the channels that connect them can be unreliable. Incomplete tasks must be remembered somewhere within a system, so that if no acknowledgement is forthcoming, they can be retried or reported as failed. This state should exist in a single place at the edge of the system, so that other processes can safely forget a command once they've passed it along.

The formal mechanism by which we accomplish and acknowledge tasks is called a **protocol**. Communication protocols like TCP and HTTP dictate the mechanism and failure modes for communication across a single edge of our system topology. When designing a system, we must do the same for performing tasks across our entire system. The design of these system-level protocols are, again, beyond the scope of this book.[37]

A task ends where the consequences of an effect become uncertain, but that is also where the value of a system begins. Displaying text or persisting data are necessary elements of building a useful system, but they are not sufficient. Systems are not intrinsically useful; like all abstractions they are judged with respect to their environment, which is in constant flux.

This book cannot tell you if your software is useful. There are no formulas to reduce our software down to an objective measure of worth. It can, however, help you judge for yourself. The ideas in this book are meant to provide a framework into which you must contribute your own understanding of your software and its environment.

[37] Anyone wishing to learn more about the inherent complexities of distributed systems should read Nancy Lynch's *Distributed Algorithms*, which is an extensive, if fairly academic, survey of the field.

We cannot solve software design, but we can reduce it to its essential questions. Happy reduction.