The Elements *of* Code

A Guide To Writing Comprehensible Software

John Mark Wilkinson

The Elements of Code

Written by John Mark Wilkinson

Edited by Ian Hough

https://elementsofcode.io

DEDICATION

This book is dedicated to my Tutu, Beth Wilkinson, who encouraged me to write more but use the word “very” less.

Thank you.

Table of Contents

1. Introduction

> It is an old observation that the best writers sometimes disregard the rules of rhetoric.
> When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules.
> After he has learned, by their guidance, to write plain English adequate for everyday uses, let him look, for the secrets of style, to the study of the masters of literature.
>
> *Strunk and White, The Elements of Style*

Just as The Elements of Style gave millions of schoolchildren a reference for writing solid, acceptable English prose, this book aims to be a reference for programmers looking to write solid, maintainable code. It does this by providing a series of rules to follow, with examples related to their use. It starts with a focus on code construction and organization, followed by naming, documentation, and tests, and finishes with the importance of practice and refactoring.

The virtues outlined by The Elements of Style are also the virtues of good code:

- Brevity - make individual lines short (this is not about reducing the total number of lines).
- Clarity - use good names and constructs that can be quickly understood.
- Flow - minimize branching logic and the need to re-read sections of code.
- Simplicity - focus on composition to avoid unnecessary abstractions and duplication.
- Unity - make modules in the codebase that work well together, and keep them orthogonal– that is, modules should not overlap in their roles.

While reading the rules in this book, consider how they tie into those virtues.

In addition, we will introduce a more measurable, vital concept: Mean Time to Comprehension (MTC). MTC is the average amount of time it takes for a programmer familiar with the given language, syntax, libraries, tooling, and structure to understand a particular block of code. Though a precise quantification of MTC may be possible, in most circumstances it would be prohibitively expensive. The goal is not precision; it is to gain good instincts about tradeoffs. Often, understanding a block of code requires understanding additional code elsewhere. In the worst case, this includes knowledge of the entire codebase, all its dependencies, and other systems it calls into. To make even the smallest contribution to such a project, significant time and effort must first be expended to acquire the necessary knowledge.

> Programmers assume they can understand a block of code in isolation, and we should strive to make that assumption correct.

The more code a programmer must understand beyond what they are immediately interacting with, the greater the MTC cost.

Our goal as programmers is to make MTC as low as possible while adhering to the specification of the project.

Specifications all follow a similar pattern:

- The program must accept some form of inputs.
- The program must process those inputs under some set of resource constraints (CPU, memory, bandwidth, time, etc).
- The program must only do what is expected of it, and nothing more.
- The program must generate precise outputs.

Occasionally, adhering to the specification is in conflict with MTC reduction. Most often this is found in high-performance systems, which require magic numbers or specialized CPU instructions.

However, reducing MTC is usually necessary to write programs which adhere rigorously to their specification. We will keep this objective in mind as we review examples throughout this book.

The Elements of Code expects the reader to be familiar with basic programming concepts, but not an expert. Of course, even the most experienced programmer may find benefit in revisiting the fundamentals on occasion. This book is a practical guide to programming, focused on application rather than theory. It contains the rules for writing code, such that you can reference them, and eventually, know when to break them.

A Message in a Bottle

In 1976, Ann Druyan sat down and thought intently about falling in love. She also thought about the history of the Earth, problems that faced the planet, and other emotions. While she thought, her brainwaves were recorded, and then transcribed onto two records.

Those records were called the “Voyager Golden Records” and were launched into space as part of the Voyager mission. They contained more than just brainwaves; they also encoded hours of various languages, natural sounds and music, along with 116 images of humans, wildlife, and scientific diagrams. They were created on the off chance an alien species runs across a Voyager probe in deep space.

Ann was the creative director of the project, and she faced a difficult technical challenge: how do you communicate with a species without sharing a language, culture, or even common experience of the natural world?

The project found their answer in fundamental physics. They used the state transition period of a hydrogen atom as a single clock cycle. From that, they derived mappings to other concepts, including the mechanism for playback of the record and how to decode the images. It was a work of brilliance.

Despite the brilliance of the scientists, the Voyager Golden Records are a message in a bottle, sent to destinations uncharted and peoples unknown. We cannot say who, if anyone, will receive them, or how they will interpret them.

The Voyager team could only communicate as clearly and intentionally as possible, hoping for the best.

In software development, we face a similar, albeit easier, message in a bottle problem: we do not know who will need to read and understand what we were doing, or what our intentions were. We must therefore communicate as effectively and intentionally as possible. Our reader may be us in three weeks, or it may be someone we have never met - and will never meet - ten years from the day we wrote the code. If a programmer attempts to work with your code years after you wrote it, and you happen to be around to answer their questions, it is a wonderfully humbling experience to watch them struggle through your creation.

Working with software requires comprehension of that software, and this means writing it is an exercise in communicating with future readers. Effective communication is difficult, and often programmers mistakenly believe that because their program is successfully understood by the compiler or runtime, it can be successfully understood by other programmers. But code *viewed as instructions* can only ever communicate behavior, not intent, as it is prescriptive in nature. Our software encompasses much more than simply an instruction set - it is our model of reality, and it embodies our understanding of how to solve the problems at hand. To convey that understanding, we need more than code that merely compiles: we need to communicate complex ideas to future readers.

To reduce the burden of communication, we must also focus on minimizing the complexity of our software. In software engineering, there is “accidental complexity”, in opposition to “inherent complexity.” Inherent complexity is the fundamental difficulty present in solving the problem: some problems are hard. Accidental complexity is the difficulty not inherent in the problem; all the convolution we create in our pursuit of addressing the real concerns of the business.

The rules presented here are an antidote, in part, to accidental complexity.

Pillars of Communication

The four pillars of communication are the available mechanisms to communicate with future readers from within the codebase. They provide different levels of flexibility, which is dictated by the computer: some decisions must conform to strict syntax, some decisions have restricted choices, and other decisions have unbounded options.

Outside the codebase, there are many additional ways of communicating: discussions between the author(s) and those new to the code, recordings of use, diagrams placed in wikis, bug tracking threads, emails and messages, even interactive LLMs with knowledge of the code and supplemental materials. Because those exist outside of the codebase, we will not cover them in this book. Instead, we will focus our attention on the software project— its source code and supporting elements— created during the development process.

As you write software, consider what knowledge you currently possess that should be communicated to future programmers to keep that software stable, extensible, and easy to work with. Use these four pillars to convey that knowledge.

Structure

Structure, the first pillar of communication, is how the code is organized, and includes file system hierarchy, line placement, syntactic choices, polymorphic abstractions, and the like. It is the first level of flexibility: the programmer has many critical choices, but those choices must still be interpreted and executed by the computer. Often, programmers fail to realize the way they structure their code indicates expected ways of interaction; notably, structural patterns are copied when additions are made. Failing to communicate how to extend the code through good structural choices results in exponential complexity growth as the codebase increases in size. This is often seen in projects with conditionals duplicated throughout the code, or large sections of code copy-pasted to achieve some new functionality. Good structure leads to orthogonal additions, which have logarithmic complexity growth. The first several chapters of this book are primarily concerned with this element of communication.

Names

Names are the identifiers given to parts within the code. They are the second level of flexibility: the programmer can choose arbitrary names, but because those names will be used by the computer, they must adhere to some rules. Bad names cause the reader confusion and can lead to misuse, and through that misuse, introduce bugs. On the other hand, good names reduce MTC and allow readers to rapidly understand the intent behind even small sections of code. This pillar is covered in detail in Chapter 7, “Naming.”

Tests

Tests, specifically unit tests, communicate usage. They show precisely how the code can be composed to accomplish a given task. They act as a set of executable, verifiable examples that let readers know how to properly work with the codebase. They are the third level of flexibility: while they must be runnable by a computer, and thus have the same structural and nomenclative constraints as the primary source code, the programmer has freedom to write as many tests as necessary to communicate the appropriate usage. Good tests inspire confidence when making changes, as they guarantee the covered behaviors remain consistent. Additionally, they act as a reference manual, and can be used to rapidly understand specific behaviors without needing comprehensive knowledge of the entire system. In large projects with many developers, it may be difficult (or perhaps impossible) for any single individual to know every aspect of the system, so being able to understand and work with discrete parts is vital. Tests are covered in detail in Chapter 9, “Unit Testing.”

Documentation

Documentation, the last pillar of communication, is the most flexible. It grants the programmer the ability to communicate ideas and intentions in a completely out-of-band manner, without regards to any constraints imposed by the computer. Not all aspects of a system can be communicated through the other three pillars, so documentation acts as an “escape hatch,” a final option for ensuring our intent and knowledge is preserved. As we will see, this is simultaneously terrific and terrifying. This is covered in chapter 8, “Documentation.”

Wrong in Correctable Ways

Every software project carries with it “behavioral models.” These are representations in code of the various ways users can interact with the system, and ways for enhancing the system to enable new interactions. Behavioral models emerge over the course of a software project. Sometimes, these are carefully designed into a specification at the beginning of the project, and then implemented. Sometimes, they are developed as the project evolves and more is understood about the required behaviors. Often, it is a combination of both.

Over a long enough timescale, the behavioral models will always be wrong. This is important to understand and accept. The world is constantly changing and evolving, and as a result the behaviors of our software must also change. We will find bugs that must be fixed, additions that need to be made, and critically, ways in which our model was incomplete or inaccurate. Perhaps we made company management software that assumed a user could only be associated with a single manager, but the organization decided to try out dual reporting lines and wants that to be represented in the system. Perhaps we have a commerce platform where we must calculate total pricing and include taxes, and an additional tariff is levied on certain products but not others, so the system must now accommodate different pricing structures for different goods. Or perhaps it is as simple as the database we were using going end-of-life, necessitating a change to an entirely different storage mechanism.

Following the rules in this book does not guarantee that the behavioral models will be correct for all possible future modifications; no rules could make that guarantee. Instead, the rules provide a path to ensure that most of the time, most of the code will not need to be modified, and that the process to update the model is relatively painless.

> This book is not about being correct, it is about being wrong in correctable ways.

Those goals- minimizing the need for modification and making additions painless- are achieved by reducing MTC, effective communication with future readers, and presenting code in a structure that can be easily extended or rearranged. Each chapter in this book will bring us closer to this overall vision.

Chapter 2 is a high level overview of code structure.

Chapters 3, 4, 5, and 6 contain some key concepts in programming, along with tactical guidance for code construction.

Chapters 7, 8, and 9 are about communicating intent via names, documentation, and tests.

Chapter 10 is about putting together these concepts to improve existing code, or to create new code.

The final chapter is about the learning process, practice, and mastery.

The examples throughout the book mostly use Python, but include other languages. This is because the rules are applicable to a wide variety of contexts; readers are encouraged to think about the examples and how they apply to their own language and environment.

The Elements of Code has a strong focus on mechanisms and structure over higher-level decisions such as design and abstractions. As a result, we will go over how to replace bad patterns with better ones, and how to think about code, rather than concepts such as determining a project’s model, architecture, or module design.

---

2. Recipe

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> *Hal Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs*

Rule: Structure Code Sequentially

Programs should read like a recipe. Start at the top, and line by line work your way down. Error conditions should be indented to indicate they are obviously error cases.

Patterns and Instructions

Good computer programs follow the same pattern that recipes have used since the 18th century: list ingredients at the top, followed by step-by-step instructions on how to interact with those ingredients to prepare the final product. Sections for related steps are broken into paragraphs.

Recipes are organized this way to improve readability and comprehension. Before they were written according to a standard, recipes would often skip steps and intermix ingredient quantities with the usage of those ingredients. The lack of formalism meant recipes were rarely used except by professional chefs who already had the context necessary to interpret the steps.

This is similar to where we find ourselves as programmers. Many projects and code bases are structured inconsistently, with object construction embedded in application logic (see the Chapter 4, “New”), run-on functions, and complex conditional logic. These projects are understandable to their creators and those who have spent significant time acquiring the necessary context, but not to anyone else.

We can avoid this by following the lessons learned by recipe-makers who needed to create instructions that could be readily understood by everyone.

Bake An Apple Pie

The standard recipe format has two sections: ingredients, and directions.

Ingredients are analogous to the “building” (construction graph, see Chapter 4, “New”) part of a program. They are the dependencies, the prerequisites that must be met before the program can be executed.

Directions are the “doing” part of a program, and are responsible for the primary work that results in the important output and behavior.

To examine this approach for creating clear instructions, let’s take a look at a recipe to bake an apple pie:

```
Ingredients:

1 pastry for a 9 inch double crust pie
½ cup unsalted butter
3 tablespoons all-purpose flour
¼ cup water
½ cup white sugar
½ cup packed brown sugar

8 Granny Smith apples - peeled, cored and sliced

Directions:

Step 1
Preheat oven to 425 degrees F (220 degrees C).
Melt the butter in a saucepan. Stir in flour to form a paste.
Add water, white sugar and brown sugar, and bring to a boil.
Reduce temperature and let simmer.

Step 2
Place the bottom crust in your pan.
Fill with apples, mounded slightly.
Cover with a lattice work crust.
Gently pour the sugar and butter liquid over the crust.
Pour slowly so that it does not run off.

Step 3
Bake 15 minutes in the preheated oven.
Reduce the temperature to 350 degrees F (175 degrees C).
Continue baking for 35 to 45 minutes, until apples are soft.
```

Since a program is executed by computers, it must be specific and detailed, but the idea is essentially the same as a recipe. Writing the apple pie recipe as code would look something like this:

```
import appliances
import pantry
import recipes
import time
import utensils
```

```
def main():
    kitchen_range = appliances.Range()
    kitchen_range.oven.preheat(425)

    saucepan = utensils.Saucepan()

    ingredients = pantry.fetch_ingredients(
        "butter", "flour", "water",
        "white_sugar", "brown_sugar", "apples"
    )

    fridge = appliances.Refrigerator()
    rolling_pin = utensils.RollingPin()

    crust_dough = prepare_dough(
        recipes.Dough(
            ingredients['butter'],
            ingredients['flour'],
            ingredients['water'],
        ),
        fridge,
        rolling_pin
    )
    lattice_dough = prepare_dough(
        recipes.Dough(
            ingredients['butter'],
            ingredients['flour'],
            ingredients['water'],
        ),
        fridge,
        rolling_pin
    )

    cook_filling(
        saucepan,
        kitchen_range.front_right_burner,
        ingredients['butter'],
        ingredients['flour'],
        ingredients['water'],
        ingredients['white_sugar'],
        ingredients['brown_sugar']
    )

    pie_pan = utensils.PiePan()
```

```
    unbaked_pie = fill_pie(
        pie_pan
        crust_dough,
        lattice_dough,
        ingredients['apples'],
        saucepan
    )

    pie = bake(kitchen_range, pie_pan)
    return pie

def prepare_dough(dough, fridge, rolling_pin):
    fridge.chill(dough)
    rolling_pin.roll(dough)
    return dough

def cook_filling(
    saucepan, burner, butter, flour,
    water, white_sugar, brown_sugar
):
    burner.temperature("medium")

    saucepan.add(butter)
    saucepan.wait_until("melted")
    saucepan.add(flour)
    saucepan.stir_until("paste")
    saucepan.add(water, white_sugar, brown_sugar)

    burner.temperature("high")
    saucepan.wait_until("boil")

    burner.temperature("low")
    return saucepan


def fill_pie(pie_pan, crust, lattice, apples, saucepan):
    pie_pan.add(crust)
    pie_pan.add(apples)
    pie_pan.add(saucepan.pour())
    pie_pan.add(lattice)

    return pie_pan

def bake(oven, pie_pan):
    oven.bake(pie_pan, 15)

    oven.set_temp(350)

    oven.bake(pie_pan, 35)
```

```
    for i in range(10):
        if pie_pan.is_baked():
            return pie
        time.sleep(60)
    else:
        raise Exception("Pie not baked after 60m in oven!")
```

The code reads as a sequence of clear, linear steps that must be taken to accomplish the task. Rather than a single function, it uses multiple functions acting like paragraphs.

Note the use of whitespace and blank lines. Having blank lines between sections allows the reader room to focus on those sections, and indicates which behaviors are conceptually related.

Program Structure

First, programs should list the “ingredients” necessary to accomplish the task.

This is the “building” phase, which we will cover in chapter 4, “New”.

```
def main():
    a = A()
    b = B()
    c = C()
```

Next, a high-level function should be created that runs the primary steps. This is similar to the way directions are broken down into “Step 1”, “Step 2”, etc.

```
def run(a, b, c):
    step1(a)
    step2(a, b)
    step3(b, c)
```

Finally, each “step” function is similar to a “paragraph”. It is a short sequence of instructions necessary to accomplish a sub-goal within the overall task.

```
def step1(a):
    translate(a)
    rotate(a)
    a.format()
```

Often, those step functions need their own steps, and should be broken down further. A good rule of thumb is to split nested elements into their own function. In example below we have a few such elements:

```
def step1(a):
    for e in a:
        sub1(e)
        sub2(e)
        sub3(e)
        for i in e.items():
            sub1(i)
            sub4(i)
```

Applying our approach, this becomes:

```
def step1(a):
    for e in a:
        process_e(e)

def process_e(e):
    sub1(e)
    sub2(e)
    sub3(e)
    for i in e.items():
        sub1(i)
        sub4(i)
```

As code becomes more indented, it becomes more difficult to reason about the layers of indentation and their interactions, increasing the program's MTC.

Breaking those indented code blocks out into their own functions can clarify what is actually happening within the blocks, and what variables are used by which blocks. This is not about the length of the function, and there are no targets with regards to the length. This is about the impact that indentation has on MTC, and ways to keep that impact low. If putting indented blocks into their own functions results in many tiny functions, it may increases MTC. In such cases, it is better to leave those blocks in a single longer function.

Let's Get Technical

Indentation refers to all nested code blocks. In many languages, indentation can be avoided through the use of semicolons to end statements and odd formatting. This does not make the code any less confusing.

Nested Function Calls

There is a common pattern in software where some functionality must be chained together to accomplish a task. This can result in nested function calls:

```
def handle_user_name_update(user_id, name):
    publish(update_name(get_user_by_id(user_id), name))
```

In the above example, we retrieve a user, update the name property, and then publish the result, all on a single line.

We read from left to right, but in nested function calls, the leftmost call is the last one, not the first, which is confusing. Even separating which values are associated with which function is cumbersome. The result is code that is difficult to reason about, and increases MTC.

> Let's Get Technical
>
> Functional programming languages often have a "pipe" operator to chain function calls without needing to nest them. This maintains readability without creating additional variables.

Instead, split the calls into multiple lines:

```
def handle_user_name_update(user_id, name):
    user = get_user_by_id(user_id)
    result = update_name(user, name)
    publish(result)
```

By performing every operation on its own line, it takes less time to understand the order of function execution and how the parameters are associated. One level of nesting is usually acceptable, but it is situational. In general, avoid nested function calls.

Error Cases

A useful pattern, popularized by idiomatic GoLang, is to left-align the happy-path code, and indent the error path code.

Visually, this results in code where error cases are trivial to spot, and can be understood rapidly.

However, this is rarely our train of thought when programming. Often, we think about problems in terms of what we can do, and then our code follows those thought patterns. For example, without using the indented error principle:

```
def process(event):
    if is_valid(event):
        log.info(event)
        event.execute()
    else:
        raise InvalidEvent(event)
```

We initially check to see if we can do a thing, and then we proceed to do it if the check was successful. The error case is listed at the bottom of the function as an afterthought.

If we follow the indented error principle, we would first check the error condition at the top of the function, and thus ensure the error case is indented and the happy-path code is left-aligned.

```
def process(event):
    if not is_valid(event):
        raise InvalidEvent(event)

    log.info(event)
    event.execute()
```

This model scales wonderfully, particularly in cases where there are multiple error conditions throughout a function. A function can be rapidly scanned, and all the error conditions identified. If there is a known bug, it is often much simpler to spot it in functions that follow this pattern, since the misbehaving case can quickly be located.

Conclusion

These structural rules result in consistent code that can be easily understood and maintained.

In fact, because comprehension is enhanced to such a degree when following this structure, bugs can often be identified simply by reading the code, without any execution required by a computer. Given the general complexity of most programs these days, that in itself is an astounding benefit.

Keep in mind that the goal is readability. If splitting functions increases MTC and increases complexity, consider the following ways to make the code still read like a recipe:

1. Check for unnecessary nesting. Perhaps there are ways to rewrite the function to reduce the indentation, such as following the indented error principle.
2. See if the arguments are related enough that they could be bundled into clearer data structures.
3. Follow the guidance in the Chapter 6, “If”, to reconstruct the program with a more linear flow.

---

3. State

> The road to programming hell is paved with global variables
>
> *Steve McConnell, Code Complete*

Rule: Reduce State and Avoid Mutation

Avoid changing the state of variables. Limit the scope of variables as much as possible.

The state of a program consists of all the values within the program at a given point in time. A practical way to think of state is as all the variables (fields, arguments, properties, etc.) and their values. Programs only consist of only two things, state and behavior, so knowing how to manage state within a program is a prerequisite for writing clear, maintainable code. It cannot be said that a program is understood unless the program's state and how it changes over time is understood in minute detail. This chapter focuses on that knowledge: we will begin by covering essential terms and definitions, then consider which practices to avoid and which to embrace, and finally we will review the impacts that state can have on an application. MTC will remain a primary focus of our discussion. Additionally, we will walk through some example code and outline precisely which state changes are taking place.

First, we need to consider the two key concepts in understanding state: *scope*, which deals with where code is usable, and *mutability*, which deals with how variable values change.

Scope

Scope is the section of code, or *context*, in which a variable is usable, and is an important way to organize code. It allows us to reuse variable names and create modular, isolated pieces of code.

Let us look at an example in Python:

```
def log_hello(value):
    hello_msg = f"Hello {value}!"
    print(hello_msg)
```

In the above function, there are two variables, `value` and `hello_msg`, which are *scoped to the function*. This means if we tried to use those variables outside of that function, there would be an error.

Lexical and Dynamic

Python, and indeed most languages (Ruby, C, C#, Java, Javascript, etc), use *lexical*, also known as *static*, scope. Lexical scope means that a variable's value is determined by the value assigned in the closest containing block of code.

There is another type of scope: *dynamic* scope, where the variable's value is determined at runtime based on the most recently assigned value within an executing block.

Let's examine the difference between lexical and dynamic by seeing how the following Perl code would run in each scenario.

```
$x = "world!"; # creates a variable 'x'
sub hello {
   print "hello ".$x."\n"; # which 'x' does this refer to?
}

sub hello_universe {
   local $x = "universe!"; # creates another variable 'x'

   hello;
}

hello_universe;
```

In the above example, `x` is first defined as "world!", and then referred to in the `hello` function.

In a lexically scoped program, this prints "hello, world!", as that is the closest assignment in a containing block of code.

However, in a dynamically scoped program, scope is inherited from execution context. This means that when the program is called, a *new* scoped variable `x` is created. When the `hello_universe` function calls the `hello` function, `hello` is given the scope of its caller, which contains a different `x` value: "universe!". Then, when the `hello` function attempts to resolve `x`, it uses the closest one, in this case "universe!", resulting in the output "hello, universe!".

A useful way to think about nested state is as a map of values. As we work through the code, we'll examine the values within that map representing the current scope.

Lexical scope:

1. A variable `x` is created. Scope: `{x: "world!"}`
2. Functions are defined but not yet called. Scope: `{x: "world!", hello: {}, hello_universe: {}}`
3. `hello_universe` is called.
4. Another variable `x` is created. Scope:
   `{x: "world!", hello: {}, hello_universe: {x: "universe!"}}`
5. `hello` is called.
6. `hello` attempts to resolve `x`. It looks in its own scope, and then moves up one level where it finds `x` is set to "world!".

7. `hello` prints “hello, world!”

Dynamic scope:

1. A variable `x` is created. Scope: `{x: "world!"}`
2. Functions are defined but not yet called. As scope is determined when functions are *called* rather than defined, the scope does not change. Scope: `{x: "world!"}`
3. `hello_universe` is called. Scope: `{x: "world!", hello_universe: {}}`
4. Another variable `x` is created. Scope: `{x: "world!", hello_universe: {x: "universe!"}}`
5. `hello` is called. Scope: `{x: "world!", hello_universe: {x: "universe!", hello: {}}}`
6. `hello` attempts to resolve `x`. It looks in its own scope, and then moves up one level where it finds `x` is set to “universe!”.
7. `hello` prints “hello, universe!”

Dynamic scope is not as popular, since it is more difficult to look at the code and quickly understand which variable is being used. With lexical scope, it is often sufficient to look at the “blocks” surrounding a particular piece of code, and know which variables are within scope.

Global Variables

Variables that are accessible at any point in a program are said to be “global”; that is, they can be used regardless of the namespace, class, module, or function. They are defined in the program’s top-level scope.

Global variables are best avoided. This includes avoiding the “singleton” pattern, where a special, globally available constructor is used to ensure only a single object is ever created. There are three major reasons for this:

- When global variables are used, it is difficult to determine where or how they are defined. While modern IDEs make this less of a concern, the required tooling is not always available, or sufficient.
- If the variable needs to be changed, determining the impacts of that change can be quite costly and difficult to reason about, dramatically increasing MTC.
- As more code refers to the same global variable, the code becomes tightly coupled and fragile. Code cannot be re-arranged, added or removed, since doing so may impact completely unrelated code.

Take for example the following Python, in `main_module.py`:

```
a = "hello world!"

def log():
    print(a)
```

What does the above log function print? You may think, it clearly prints “hello world!”, but with further consideration you might get the feeling that this is a trap, since it appears to be using a global variable. You would be correct: it is a trap. In another module, far away inside the program, there is this insidious code:

```
import main_module

main_module.a = "unexpected value!"
```

This has reached inside our module and modified its value, causing us to print “unexpected value!”. In fact, Python is better than many languages at managing global scope, because a variable labeled as global is still restricted to the module level. Even so, the value can be modified in unexpected ways, leaving subsequent programmers confused and frustrated.

Proximal Variables

Understanding the perils of global variables leads us to use the opposite: proximal variables, sometimes imprecisely referred to as “local” variables. Proximal variables are created closest to the location where they are used, while local variables are created in a given scope, and are considered “local” to that scope (they cannot be accessed outside of it).

For example, here is a use of local, but non-proximal, variables in Python:

```
def main():
    msg_1 = "count: "
    msg_2 = "value: "
    msg_3 = "index: "

    for i in range(10):
        print(msg_3 + str(i))

    for i in range(100):
        print(msg_1 + str(i))

    for i in range(1000):
        print(msg_2 + str(i))
```

In the above example, the `msg` variables are defined at the top of the function, but they are not used until later in the code. While there is some benefit to grouping variables that are used together for reference, it tends to be clearer and reduces MTC to define them more *proximally*– close to where they are used. Note how in the first example it is easy to assume they will be used in the order they are declared and named: `msg_1`, then `msg_2`, then `msg_3`. However, defining them proximally to their use makes it obvious this is not the case:

> Let's Get Technical
>
> Often, this style is an artifact of languages like C, where variables needed to be declared at the beginning of the scope block (in this case, the top of the function). This was required in C89, but not in C99, and should be avoided.

```
def main():
    msg_3 = "index: "
    for i in range(10):
        print(msg_3 + str(i))

    msg_1 = "count: "
```

```
for i in range(100):
    print(msg_1 + str(i))

msg_2 = "value: "
for i in range(1000):
    print(msg_2 + str(i))
```

In general, try to define variables as *proximally* as possible.

Mutability

Variables whose values can be changed after creation are called *mutable variables*. Variables whose values cannot be changed over the course of program execution are called *immutable variables*. Immutable variables are a form of *invariants*: conditions that do not vary (or change) throughout execution. Whenever possible, we should prefer immutable variables over mutable ones. This is because invariants make our program simpler to understand, because we do not need to track changes over time. The following section focuses on immutability and how to write immutable code.

There are two general types of immutability: weak and strong.

Weak vs Strong Immutability

Weak immutability is when a reference cannot be changed (variables cannot be reassigned to new values), but data contained within the reference can be. The following is an example of weak immutability.

```
> const example = {value: 3};
undefined
> example = {id: 1}
Uncaught TypeError: Assignment to constant variable.
> example.value = 4
4
```

In this Javascript example, we create a `const` variable, and then attempt to reassign it, which results in an error.

Javascript will, however, allow us to mutate the internal data, such as when we set `example.value` to `4`, making its `const` a form of weak immutability.

Strong immutability prevents any change to the data. In fact, languages with strong immutability often lack the syntax to express such changes. Instead, modifications return a new reference to the updated value, which must be rebound to a variable. This is effectively the inverse of the weak immutability we saw in Javascript. Not all languages support both strong and weak immutability; often, they support either one or the other. Javascript, for example, does not natively support strong immutability. Elixir, on the other hand, supports strong immutability but allows “rebinding” (similar to variable reassignment in most languages). Here is an example of strong immutability in Elixir: we cannot update our existing reference, we can only rebind it to a new value.

```
iex> example = Map.new
%{}
iex> Map.put(example, :value, 1)
%{value: 1}
iex> example
%{}
iex> example = Map.put(example, :value, 1)
%{value: 1}
iex> example
%{value: 1}
```

In the above example, we perform the following steps:

1. We create a new, empty Map label (similar to a variable on other languages), which we name `example`. The REPL indicates to us the map is empty.
2. We attempt to put a value into `example`. Because Elixir is immutable, this returns a new map but does not modify `example`.
3. We verify that `example` remains empty; it has not been modified.
4. We perform the same operation again, but this time we rebind the label `example` to the return value.
5. We verify that `example` now contains the data we added.

Because Elixir does not allow us to modify the internal data of example, and instead we must rebind the label to the new data, Elixir provides strong immutability.

Impact of Mutation

Mutating state is one of the most common forms of bugs. It is difficult to keep track of when and where state is being changed, and by who.

This is why, even in languages that do not support immutability, it is best to write code that does not change variable state. Although this is not possible all the time, we can usually get close.

```
def build_request(hostname, duration, value):
    request = {}

    request['hostname'] = hostname
    request['duration'] = duration

    add_record(request, value)

    return
```

```
def add_record(request, value):

    record_id = get_next_id()

    request['record'] = {}
    request['record']['id'] = record_id
    request['record']['value'] = value
```

In the above Python example, we initially create a `dict`, and then proceed to add keys into it after creation. Interacting with the variable in this way makes it hard to reason about the structure of the data. It would be simpler to create it all at once, rather than mutate it, as doing so makes the structure obvious. For example:

```
def build_request(hostname, duration, value):
    return {
        "hostname": hostname,
        "duration": duration,
        "record": build_record(value)
    }

def build_record(value):
    return {
        "id": get_next_id(),
        "value": value
    }
```

We may even want to combine these two functions, to give even more clarity to the structure we are dealing with:

```
def build_request(hostname, duration, value):
    return {
        "hostname": hostname,
        "duration": duration,
        "record": {
            "id": get_next_id(),
            "value": value
        }
    }
```

Of course, decreasing MTC is not the only reason to program this way. Immutability has a large impact on concurrency, as well. Functions that do not mutate state can easily be run in parallel, as they do not need to lock access to their variables to prevent race conditions (i.e. non-deterministic modification of a value by concurrent operations) and the resulting data corruption.

> Let's Get Technical
>
> Immutability can have a cost, usually in the form of performance. Even languages that support immutability as a first-class concept tend to be slower than their lower-level counterparts. For most projects, this difference is small enough as to not be a consideration. However, if performance is critical, the impacts of mutable variables and shared memory should be meticulously considered prior to their implementation. In that case, it should still be possible to avoid mutating the vast majority of the variables.

Conclusion

State is a fundamental aspect of programs. If state management is done poorly, or is difficult to understand, it will be almost impossible to write bug-free programs. The unnecessary complexity and confusion created by poor state management inevitably results in the wrong value being updated, or missing which values should be updated, or, in the case of concurrent operations, race conditions and data corruption.

To properly manage state, take care to do two things:

1. Ensure you understand precisely how the state in your program behaves.
2. Write the program so that step one is simple.

The rules outlined throughout this chapter will help you accomplish this task.

---

4. New

> Most developers freely mix the "new" operator with the application logic. In order to have a testable code-base your application should have two kinds of classes. The factories, these are full of the "new" operators and are responsible for building the object graph of your application, but don't do anything. And the application logic classes which are devoid of the "new" operator and are responsible for doing work.
>
> *Miško Hevery, Writing Testable Code*

Rule: Separate Construction and Use

New objects or data structures must be constructed by objects or functions dedicated to that purpose, and not mixed into the objects or functions that perform the primary work of the application.

Most programming languages have a “new” keyword, which is responsible for indicating to the compiler that memory needs to be allocated and initialized based on a given template (in object-oriented languages, a “class”). Even if languages lack that specific keyword, they have syntax for creating new structures with associated state and behavior. In this chapter, we refer to that syntax as “new”, even if the particular word is not used.

- Builder: A builder is a class, structure, or function whose sole purpose is the construction and configuration of other objects or structures.
- Business Object: A business object is an object which performs functions intimately related to the core purpose of the application.
- Data Object: A data object is a simple object whose only purpose is to store data. These are commonly either maps or arrays, but most languages include additional data structures in their standard library.

One of the easiest mistakes to correct in software is improperly combining the “building” (construction graph) logic with the “business” (application) logic. Separating those within the program reduces the MTC.

The way to accomplish this is simple: in any business object, only data objects may be constructed. All other objects must be created by builders. Or to put it another way: don’t use the “new” keyword (or anything analogous) in any business object. This concept is closely connected to the ideas in Chapter 3, “State.” Keeping the building logic separate from the business logic is a practical way to manage scope, as the building logic becomes responsible for the available scope in any given piece of code.

All program entry points operate as “builders,” and are responsible for the startup procedure of the entire application. For example, the following are the primary builder functions in Java and Python, respectively:

```
// This is the entrypoint builder function in Java
public static void main(String args[]) {}
```

```
# This is the entrypoint builder function in Python
def main():
    pass

if __name__ == "__main__":
    main()
```

Often, initial builder functions are quite long. They construct the entire application, starting with the most foundational objects and long-running application services, before creating the higher-level, more complex objects. As the initial builder function grows, it is useful to split these functions into smaller parts. When and where that is appropriate is more a matter of taste than pragmatism. Since the initial application wiring requires a lot of context and references, splitting it up will sometimes result in silly functions which take a very high number of arguments (have high “arity”).

The way the internals of the application are connected, the way the application is “wired”, can be imagined as a hierarchy of references, where high-level objects refer to lower ones, which in turn refer to even lower ones, all the way down to the foundational objects. This hierarchy can be modeled as a graph, where the objects are nodes and the dependencies are edges, and is often referred to as the application’s “construction graph”. The “construction graph” is the fundamental responsibility of the builder logic.

Once the initial application construction is complete, program execution is handed over to business objects. A business object is one that performs calculations, interacts with other systems, or is generally concerned with the actual goals of the application and users.

```
# This is a business object
class SlowGrep:
    def __init__(self, path: str):
        self.path = path

    def find(pattern):
        """
        Look for the pattern in all the files
        """
```

The separation of builder logic and business logic is incredibly powerful. It keeps the “doing” code obvious, simple, and maintainable. It ensures the “building” code is transparent and easy to modify.

Mixed Building and Business Logic

Let’s look at a bad example of object construction in Python, where the “building” and “doing” logic have not been separated, and discover how such code is modified over time:

```
class HttpClient:
    def __init__(self, url):
        self.url = url

    def fetch_data(self):
        resp = requests.get(self.url)
        return resp.text

class Parser:
    def __init__(self, url):
        self.http_client = HttpClient(url)

    def parse(self):
        data = self.http_client.fetch_data()
        parsed = {}
        for line in data.splitlines():
            key, _, value = line.partition("=")
            parsed[key] = value
```

The problem is that the `Parser` is building the `HttpClient`. On the surface, this doesn't look too bad, but as soon as we add a few more requirements the problems become obvious.

Perhaps we need to optionally use a `RetryHttpClient` that handles retries:

```
class RetryHttpClient(HttpClient):
    def __init__(self, url, retries):
        super().__init__(url)
        self.retries = retries
        # Configure internal retry logic here

    def fetch_data(self, query):
        resp = requests.get(self.url, params=query)
        return resp.text

class Parser:
    def __init__(self, url, use_retries, retries=0):
        if use_retries:
            self.http_client = RetryHttpClient(url, retries)
        else:
            self.http_client = HttpClient(url)

    def parse(self, query):
        data = self.http_client.fetch_data(query)
        parsed = {}
        for line in data.splitlines():
            key, _, value = line.partition("=")
            parsed[key] = value
```

Well, this is going downhill fast. Now we have branching construction logic in the `Parser`. What happens if we need to use a more complex `RetryHttpClient`, like `ComplexRetryHttpClient` which takes a `retry_strategy` to determine how to do retries?

Let's look at just the `__init__` method, as this is getting quite long.

```
class Parser:
    def __init__(
        self, url, use_retries, retries=0, retry_strategy=None
    ):
        if use_retries:
            if retry_strategy is not None:
                self.http_client = ComplexRetryHttpClient(
                    url, retries, retry_strategy
                )
            else:
                self.http_client = RetryHttpClient(url, retries)
        else:
            self.http_client = HttpClient(url)
```

Wasn't the point of the parser just to parse the data? Now it is convoluted! But it is easy to see how we got here: we made `Parser` responsible for builder logic. So as the construction graph complexity goes up, so does the complexity of the `Parser`.

An additional downside is that since the logic to construct the http client is inside the `Parser`, the quickest way to construct an http client is now to simply make a new `Parser`. This can lead to confusing situations like the following:

```
class RemoteData:
    def __init__(self, url, use_retries, retries=0, retry_strategy=None):
        p = Parser(url, use_retries, retries=retries,
                   retry_strategy=retry_strategy)
        self.http_client = p.http_client
```

But that long parameter list is ugly, and this is Python, so we can hide it:

```
class RemoteData:
    def __init__(self, *args, **kwargs):
        p = Parser(*args, **kwargs)
        self.http_client = p.http_client
```

Now we have truly confused anyone trying to make sense of what this program is supposed to do. This is madness.

Separated Building and Business Logic

Let's fix all of this, starting with the first example.

```
import requests
```

```
class HttpClient:
    def __init__(self, url):
        self.url = url

    def fetch_data(self):
        resp = requests.get(self.url)
        return resp.text

class Parser:
    # Note: Parser really shouldn't use an HttpClient at all.
    # It would be much better to separate the
    # concerns of fetching the data
    # entirely from the concerns of parsing the data
    def __init__(self, http_client):
        self.http_client = http_client

    def parse(self):
        data = self.http_client.fetch_data()
        parsed = {}
        for line in data.splitlines():
            key, _, value = line.partition("=")
            parsed[key] = value

def main(args):
    # Assume we first parse out the url from argv

    client = HttpClient(url)

    parser = Parser(client)
```

We have moved the building to a builder function `main`. Now let's go back and add in all that additional retry code:

```
def main():
    # Assume we first parse out the parameters from argv

    if use_retries:
        if retry_strategy is not None:
            client = ComplexRetryHttpClient(url, retries, retry_strategy)
        else:
            client = RetryHttpClient(url, retries)
    else:
        client = HttpClient(url)

    parser = Parser(client)
```

Much better. The parser has not increased in complexity: instead, we have completely isolated the complexity within the builder function, where it can easily be understood, and updated or moved with little effort.

Finally, let's see what the `RemoteData` example looks like in the corrected scenario:

```
class RemoteData:
    def __init__(self, http_client):
        self.http_client = http_client

def main(args):
    # Construct the http client and parser as before

    remote_data = RemoteData(http_client)
```

This is straightforward and easy to understand.

How The Code Should Look

The mixing of building and doing logic is one of the most pervasive and harmful programming mistakes, but also one of the simplest to fix.

When your application first boots up, it should read something like this:

```
function bootstrap() {
    let log = new Logger();

    let foundationA = new FoundationA(log);
    let foundationB = new FoundationB(log);

    let serviceA = new ServiceA(foundationA, foundationB);

    let serviceB = new ServiceB(serviceA, foundationA);
    let serviceC = new ServiceC(serviceA, foundationB);

    let application = new Application(log, serviceB, serviceC);
}
```

With this structure, it is trivial to see exactly what the chain of dependencies is. The sequencing of dependencies in your program is its construction graph. If additional functionality needs to be added, it can be easily accomplished by just wiring it into that construction graph.

Injecting Dependent Behaviors

Part of the benefit of separating "building" from "business" is how it allows us to configure behavior through composition (discussed in chapter 5, "Polymorphism") and dependency injection. We can take advantage of this when we recognize that some business logic seems to do two unrelated tasks. We can then create two different constructs responsible for the two different tasks, and compose them together using the building logic.

Let us take another look at the `SlowGrep` Python example from the beginning of this chapter.

```
# This is a business object
class SlowGrep:
    def __init__(self, path: str):
        self.path = path

    def find(pattern):
        """
        Look for the pattern in all the files
        """
```

The `find` function takes a "path" argument, which indicates that the find function handles opening and searching all the files. However, the logic for matching should probably not be coupled to the logic for walking through the files, so let's pull that out.

```
import os


class SlowGrep:
    def __init__(self, matcher):
        self.matcher = matcher

    def find(self, path):
        matches = []

        for root, dirs, files in os.walk(path):
            for name in files:
                self.search_file(os.path.join(path, name))

            for name in dirs:
                matches.append(*self.find(os.path.join(path, name)))
        return matches

    def search_file(self, file_path):
        with open(file_path) as f:
            return self.matcher.match(f.read())


class SimpleMatcher:
    def __init__(self, pattern):
        self.pattern = pattern

    def match(self, contents):
        return self.pattern in contents
```

Note that `SimpleMatcher` is not constructed by `SlowGrep`. Instead, `SlowGrep` expects it to be injected into the class. This is powerful, since we can now create other types of matchers. For example, let's make one that uses regex:

```
import re


class RegexMatcher:
    def __init__(self, pattern):
        self.pattern = re.compile(pattern)

    def match(self, contents):
        return self.pattern.search(contents) is not None
```

This is more flexible. But from a construction graph perspective, there is still a problem: `SlowGrep` is creating new `File` objects, and `SlowGrep` appears to be a business object. We can extract the embedded notion of an iterator from `SlowGrep`, and put it in a dedicated object. Because we're doing this in Python, we'll take advantage of pythonic methods to enable iteration: `__iter__`, which must return the iterator to use, and `__next__`, which must return the next element in the sequence we are iterating.

```
import os


class FileContentsIterator:
    """
    FileIterator takes a string base path and
    iteratively returns the contents of all files,
    including subdirectories within that path
    """
    def __init__(self, path):
        self.path = path

    def __iter__(self):
        return self

    def __next__(self):
        return self._search_dir(self.path)

    def _search_dir(self, path):
        for root, dirs, files in os.walk(self.path):
            for name in files:
                yield from self._search_dir(os.path.join(path, name))

            for name in dirs:
                with open(os.path.join(path, name)) as f:
                    contents = f.read()
                    yield contents

class FileIteratorBuilder:
    def __call__(self, path):
        return FileContentsIterator(path)
```

```
class SlowGrep:
    def __init__(self, iterator: FileIterationBuilder, matcher):
        self.iterator = iterator
        self.matcher = matcher

    def find(self, path):
        for contents in self.iterator(path):
            return self.matcher.match(contents)
```

In the above example, `FileContentsIterator` acts as a short-lived object builder, responsible for creating `File` objects which do not persist for the entire execution of the application. These will be discussed in the following section.

> Let's Get Technical
>
> The primary drawback to this approach is performance. A general rule of thumb is: the more performant code must be, the more coupled it will also be. The inverse is therefore also true: the more decoupled code is, the more difficult it is to make it performant. A point of nuance to consider here, however: just because code is coupled does not mean it is performant.

Organizing code this way has tradeoffs. When used in simple systems, creating an abstraction like this increases MTC. However, because it decouples both the matching operation and the thing that finds us stuff to match, we could create an iterator that traverses a remote directory structure over SSH, or perhaps walks through website links, or reads SNMP values, or some other operation. Structuring code in this way involves creating a behavioral model. For more information, see “Choosing a Model” in Chapter 6, “If”.

A Note on Short-lived objects

You may be aware of the “Factory Pattern.” While most construction logic should take place when the application is starting up, at the “bootstrap” phase, this won’t always be sufficient. Occasionally shorter-lived objects (transient objects) will be necessary, based on external input and imbued with context (scope, discussed in Chapter 3, “State”) such that they can be passed around to long-lived service objects. Factories exist to create transient objects by combining *service objects* with *runtime input*.

The following is an example of how to create a factory (`TaskBuilder`), which can make transient objects (`Task`):

```
class Task:
    def __init__(self, log, api, id):
        self.log = log
        self.api = api
        self.id = id

    def cancel(self):
        self.log.debug(f"Cancelling task {self.id}")
        self.api.delete(self.id)
```

```
    async def results(self):
        r = await self.api.get(f"/results/{self.id}")
        self.log.debug(f"Results for task {self.id}\n" + str(r))


class TaskBuilder:
    def __init__(self, log, api):
        self.log = log
        self.api = api

    def build(self, id):
        return Task(self.log, self.api, id)


class TaskManager:
    def __init__(self, api, task_builder):
        self.api = api
        self.task_builder = task_builder

    async def run(self, spec):
        r = await self.api.post(f"/tasks", json=spec)
        return self.task_builder.build(r.id)
```

In this example, `Task` is an encapsulation of some executing value, and requires three inputs: `log`, `api`, and `id`. Two of these, `log` and `api`, are (probably) "long-lived" service objects, while `id` is only known at runtime. `TaskBuilder` acts as a way to combine the long-lived values with the temporary values to create the `Task`.

Instead of using `TaskBuilder`, it may seem reasonable to simply have `TaskManager` create the `Task` directly. This, however, would violate our rule, as `TaskManager` is a business object. Let's examine further the negative impacts of that rule violation.

- Currently, `Task` only requires two *service objects*, but it may need more in the future. `TaskBuilder` allows those service objects to remain decoupled from everything that needs a `Task`.
- The `log` and `api` objects may actually need to be transient objects themselves, with specialized configurations related to `Task`, such as logging to an additional location, or sending additional headers. Making `TaskManager` responsible for the creation of additional objects that have nothing to do with it increases its coupling and unnecessarily complicates it.
- It would be difficult to test `TaskManager`, since it would return a static `Task`.
- If another `Task` type was introduced, either `TaskManager` would need to have a conditional added to figure out which one to create, or another `TaskManager`, with duplicate logic, would be introduced. We will cover this in more detail in Chapter 6, "If".

Conclusion

Keeping the `new` operator separate from business logic feels odd at first, and it requires practice to get acquire the habit of immediately separating concerns. Often, going through the exercise of keeping builder logic and business logic apart seems unnecessary or a waste of time, or perhaps appears to add complexity. However, in your day-to-day programming you have likely noticed that as the codebase grows, somehow it also becomes more and more difficult to change, and more and more difficult to understand. Mixing these concerns of construction and business, failing to follow the `new` rule, is the culprit well over half the time.

If you encounter a codebase that is difficult to work with, and is in need of refactoring, separating construction and business logic is often the best place to start. Pull out `new` logic piece by piece, and put that logic in the bootstrap phase, as was done in example with the `HTTPClient` and `Parser` objects. The process of organizing and straightening “spaghetti code”, code whose logic is tangled and difficult to reason about, is the first step towards understandable software.

---

5. Polymorphism

> If I paint a fine shark upon this page, will you say, “Fine shark!”
> or will you complain that it is flat and does not eat you?
>
> *Kaimu, The Codeless Code, Case 175*

Rule: Use Polymorphism

Polymorphism is the most effective way to make composable pieces that build into a flexible system. Organize code to leverage the polymorphism of the language.

Polymorphism is one of those Computer Science words that gets bandied about frequently, but is rarely explained. The term originated from the fields of biology and genetics, and simply refers to how members of the same species can have different forms. For example, individual butterflies of the same species vary in the kinds of patterns and colors found on their wings. Computer Science re-purposed the term.

Here is a simple, straightforward definition: *Polymorphism is having multiple behaviors that adhere to a single interface.*

Polymorphism lets us create interchangeable pieces and modularity. This means that instead of needing to completely rewrite a program from the ground up when adding new functionality, we can add functions that implement the new behavior, and then call those functions when the new behavior is required. Polymorphism is the bedrock of creating *correctable code*, discussed in the Introduction as a core objective of this book. Understanding its use is essential to writing maintainable programs.

This is achieved through function dispatch, which has two high-level categories: static dispatch and dynamic dispatch. Dynamic dispatch has two subcategories: single dispatch and multiple dispatch.

This chapter covers function dispatch, as well as how polymorphism relates to the concepts of composition and inheritance. It has a stronger focus on programming theory than the rest of the book, because understanding that theory is essential to proper understanding of the topics covered later in Chapter 6, “If.” Additionally, although polymorphism’s application is generally oriented towards software design, we will mostly cover it in terms of code construction techniques.

Static dispatch

Static dispatch occurs when the function to be called is determined at compile time based on reference (e.g. object or struct associations). The following code is an example of this, with the association between Logger and Writer acting to determine the function dispatch.

```
package main

import "fmt"

type Writer interface {
        Write(p []byte) (n int, err error)
}

type HexWriter struct{}

func (w HexWriter) Write(p []byte) (int, error) {
        fmt.Printf("%x\n", p)
        return len(p), nil
}

type StringWriter struct{}

func (s StringWriter) Write(p []byte) (int, error) {
        fmt.Printf("%s\n", p)
        return len(p), nil
}

type Logger struct {
        w Writer
}

func (l *Logger) Log(s string) {
        l.w.Write([]byte(s))
}

func main() {
        l1 := Logger{
                &StringWriter{},
        }

        l2 := Logger{
                &HexWriter{},
        }

        l1.Log("hello, world!") // prints "hello, world!"
        l2.Log("hello, world!") // prints "68656c6c6f2c20776f726c6421"
}
```

In the above Golang example, the `HexWriter` and `StringWriter` have identical `Write` functions, and the code in `main` determines which one to call based on the compile-time evaluation.

Many programming languages support classes, which are simply a way of passing consistent scope to a set of related functions (see Chapter 3, “State”). The scope is referenced by a variable such as `this`, or `self`, or `receiver` (or whatever you want to call the structure associated with the function). That reference acts as an input to the function, and the function uses that scope to achieve polymorphism.

Dynamic Dispatch

Dynamic dispatch operates the similarly to static dispatch, except which function to call is determined at runtime rather than compile time, based on the type of arguments it can receive, or its arity (number of arguments). This allows for more sophisticated behavior, such as adding plugins to extend a program’s capability without rebuilding the system, but has the drawback that the compiler cannot easily detect errors.

The two forms of dynamic dispatch are single dispatch, and multiple dispatch. They differ in whether the function to be called is determined by one or multiple values.

Single Dispatch

Single dispatch is a form of dynamic dispatch where the selected function is based on a single value.

In some languages, this might be through a “special” object, such as `this` or `self`. In this example, the “special” object is `self`, and contains the related scope defined by the associated class the object was initialized from.

```
class Comparator:
    def compare(self, a, b):
        return self.bigger(a, b)

class IntComparator(Comparator):
    #  Returns True if a is bigger than b
    def bigger(self, a, b):
        return a > b

class StringComparator(Comparator):
    #  Returns True if a is longer than b
    def bigger(self, a, b):
        return len(a) > len(b)

def compare(a, b, comparator):
    return comparator.compare(a, b)

compare(1, 2, IntComparator())
compare("hi", "bye", StringComparator())
```

In other languages, function selection may be based on the type of a single argument passed into the function. In this example, taken from the Elixir programming language docs, the `Utility.type/1` function is defined for both `BitString` and `Integer` types. Depending on the argument types it is called with, a different implementation will be invoked.

```
defprotocol Utility do
  @spec type(t) :: String.t()
  def type(value)
end

defimpl Utility, for: BitString do
  def type(_value), do: "string"
end

defimpl Utility, for: Integer do
  def type(_value), do: "integer"
end



iex> Utility.type("foo")
"string"
iex> Utility.type(123)
"integer"
```

The benefit of single dispatch is its simplicity. Since a single value is used to determine which function to call, it is straightforward to understand how the code will execute.

Multiple Dispatch

Multiple dispatch is when functions have the same name, but differ in either the types of arguments they can receive, their return type, or their arity. These differences are then used to select the matching function for a caller at runtime. In this Elixir example, the function is selected based on the first matching function definition.

```
defmodule Utility do
  def type(a, true) when is_bitstring(a), do: "string|true"
  def type(a, false) when is_bitstring(a), do: "string|false"

  def type(a, true) when is_number(a), do: "number|true"
  def type(a, false) when is_number(a), do: "number|false"

  def type(a, true) when is_boolean(a), do: "boolean|true"
  def type(a, false) when is_boolean(a), do: "boolean|false"
end

IO.puts(Utility.type("hi", true))   # string|true
IO.puts(Utility.type("hi", false))  # string|false
IO.puts(Utility.type(1, true))      # number|true
IO.puts(Utility.type(true, false))  # boolean|false
```

The benefit of multiple dispatch is its expressiveness. The ability to reuse a common name and simply redefine the argument types allows the code to be extended easily, without any need to change existing behavior. This provides a lot of potential flexibility, often at the cost of some simplicity.

Using Function Dispatch

Most languages do not support all forms of function dispatch, so picking which option will work best for a given scenario is often dependent on the language you are working in.

In general, static dispatch is a good default choice because it is the simplest and has the best compiler support, so is the easiest to debug. In terms of complexity, static dispatch is the simplest, followed by single dispatch, then multiple dispatch. Prefer the lowest level of complexity that can solve the problem, without introducing additional problems. If you are unsure which type to choose, it is almost always easier to go from a lower complexity level into a higher one, than from a higher complexity level into a lower one.

The most important activity is to think critically about the problem being solved, and ensure that you have a comprehensive understanding of the behaviors you need to achieve.

Composition

Composition is a way to achieve polymorphism through explicit configuration and references, rather than a compiler or runtime inferring it.

See the following Python example where an object is *composed* with a function to achieve different behaviors:

```
#  Returns True if a is bigger than b
def compare_ints(a, b):
    return a > b

#  Returns True if a is longer than b
def compare_strs(a, b):
    return len(a) > len(b)

class Comparator:
    def __init__(self, compare):
        self.compare = compare

int_comp = Comparator(compare_ints)
int_comp.compare(3, 1)  # True
str_comp = Comparator(compare_strs)
str_comp.compare("Bigger", "Smaller")  # False
```

Note that there is still a function `compare` that will behave differently, but instead of composing at the functional level, we are composing at the object level, creating a reference to the appropriate function. The primary difference is that both `int_comp` and `str_comp` will behave differently for their entire lifecycle (excluding mutation).

We can achieve a high level of customization by using this style of compositional polymorphism, where properties are set and functionality is created through association.

This is how programmers create modularity, and we will explore its impacts further in Chapter 6, “If”.

> Let's Get Technical
>
> Some may note that this is redundant- we could simply pass function references to any object requiring them, as in the previous example. But that would fail to demonstrate how composition can be polymorphic at the object level.

Inheritance

Inheritance is when an object can derive properties and behavior from one or more other objects.

Inheritance is not intrinsically bad, but it lends itself to easy misuse.

Inheritance is good when you have a set of objects that, *by definition*, have the same set of functions, but some of those functions need to have different behavior.

Here is an impractical, but reasonable, use of inheritance:

```
class Car:
    def __init__(self):
        self._speed = 0

    @property
    def speed(self):
        return self.speed

    def accelerate(self):
        raise NotImplemented("accelerate is not implemented in Car")


class Buick(Car):
    @property
    def max_speed(self):
        return 150

    def accelerate(self):
        self._speed += 10


class Ferrari(Car):
    @property
    def max_speed(self):
        return 250

    def accelerate(self):
        self._speed += 20
```

We can see that both the `Buick` and the `Ferrari` inherit from `Car` and share the `speed` property, but they differ by `max_speed` (150 vs 250) and they have their own acceleration rates (10 vs 20). This means that the two `Car` classes provide the same functions, but will behave differently when those functions are called.

Here is another, *slightly* practical example:

```
class Iterator:
    def __init__(self, condition):
        """
        condition is some object with a `status` function,
        which will return "complete" once done.
        """
        self.condition = condition

    @property
    def is_complete(self):
        return self.condition.status() == "complete"

class NumericIterator(Iterator):
    def __init__(self, condition):
        super().__init__(condition)
        self.value = 0

    def next(self):
        if self.is_complete:
            raise StopIteration()
        ret_val = self.value
        self.value += 1
        return ret_val

class AlphabeticIterator(Iterator):
    def __init__(self, condition):
        super().__init__(condition)
        self.alphabet = string.ascii_lowercase

    def next(self):
        if self.is_complete or len(self.alphabet) == 0:
            raise StopIteration()
        return self.values.pop(0)
```

A key thing to note here is that the `NumericIterator` is bounded only by the `condition.status()` value, unlike the `AlphabeticIterator` which is also bounded by the number of letters in the alphabet. This means we could continue to call `next` on the `NumericIterator` and count as high as Python will let us, but as soon as the `AlphabeticIterator`'s `next` function returns "z", iteration will end, regardless of the `condition.status()` result.

Let's look at a bad example of inheritance:

```
class TipCalculator:
    def __init__(self, percentage):
        self.percentage = percentage

    def calculate(self, cost):
        return cost * (1 + self.percentage / 100)


class RestaurantGuest(TipCalculator):
    def __init__(self, percentage, preferences):
        super().__init__(percentage)
        self.preferences = preferences

    def order(self):
        return self.preferences.order()
```

Here, `RestaurantGuest` extends `TipCalculator`, not because of any behavioral relationship, but because we wanted to "reuse" the `calculate` method of `TipCalculator`.

This is a poor use of inheritance. The convenience of not needing to compose a single function into a class does not outweigh the burden of inheriting all the irrelevant functionality. This irrelevant functionality includes not only what currently exists, but all future additions as `TipCalculator` is updated. There is a high MTC cost to poor inheritance: code updates that should be straightforward will have unclear impacts, and so substantially more of the system must be understood before changes can be made.

Instead, we should recognize that the `TipCalculator` *functionality* is what we require, and pass a `TipCalculator` into the constructor of `RestaurantGuest` following the guidelines in the section "Composition". Doing so allows us to add new `TipCalculator`s in the future, and update which one `RestaurantGuest` uses, without ever modifying the `RestaurantGuest` class.

This is the fundamental danger of inheritance: it is easy in the moment to simply extend existing behavior to accomplish a goal, without realizing that doing so may have substantially increased the overall MTC. Because of this, every use of inheritance should be carefully scrutinized to ensure it remains consistent with the original intent of the code.

Conclusion

Polymorphism may be the most powerful tool in a programmer's toolbox, but perhaps also the most dangerous. It allows us to model any behavior, any complex system, in ways that can be orthogonally extended. It also allows us to create incredible complexity.

Think deeply about the problem space you are modeling, and the behaviors you are trying to achieve, before creating some polymorphic abstraction. Do not create unnecessary abstractions, which results in more code to understand and maintain within a system.

When used properly, polymorphism is the single most effective tool for making systems that stand the test of time.

---

6. If

> The computing scientist's main challenge is not to get confused by the complexities of his own making.
>
> *Edsger W. Dijkstra*

Rule: Be Suspicious of If

Conditional logic complicates programs, and is often a mistake. Use polymorphism and program structure to avoid the use of unnecessary if statements.

One of the first introductions most students have to the world of code is that of the `if` statement. It teaches how to think about logic and conditionals, to view a program through the lens of taking action based on state. It is also one of the primary causes of program complexity (as measured by the number of linearly independent paths through the code, or "cyclomatic complexity"), is mostly misused, and is a major source of confusion for those trying to understand the flow of the code.

Every `if` statement creates a branch in the execution path that the code could take. The problem with this is that each branch and its implications must be fully understood. Branching logic has a high cognitive load: it is difficult to reason about multiple things at the same time.

Imagine if this book was available in multiple languages, and in order to find the subsequent section in your language, you needed to read the prior section that would tell you which page to go to next. Then, on that page, you needed to follow a further delineation based on dialect, and perhaps an even further one based on phrasing.

Once you combined the phrasing, dialect, and language subsections together you could deduce the meaning, after which you would backtrack in order to determine which page has the next section, before following the entire process again. This would be slow, tedious, and frustrating, which is why books do not work that way.

Instead, books are constructed for a given language, and when they are translated, they are translated in whole and published for that language. To make something comprehensible, it must be as linear as possible.

Mapped Values

Often, `if` statements can be replaced with map values.

For example, using Python:

```
def calculate_discount(member_level, purchase_amount):
    if member_level == "Silver":
        discount = 0.95
    elif member_level == "Gold":
        discount = 0.90
    elif member_level == "Platinum":
        discount = 0.85
    else:
        discount = 1

    return purchase_amount * discount
```

A clearer way to write this is by using a `map` (or `dict`), where the conditional has been replaced with a simple key lookup in the map:

```
membership_levels = {
    "Silver": 0.95,
    "Gold": 0.9,
    "Platinum": 0.85
}

def calculate_discount(member_level, purchase_amount):
    discount = membership_levels.get(member_level, 1)
    return purchase_amount * discount
```

Using this pattern, it is trivial to add new levels if more are defined. Additionally, the reader can rapidly understand which membership levels are currently defined and their related discounts, ensuring MTC is kept low.

Types

To represent more complex values, we should use the polymorphic type system that almost all languages have (see Chapter 5, “Polymorphism”). In object-oriented languages, we can associate state with behaviors.

In the following bad example, we use a property `shape_type`, when instead we should be using the built-in type system:

```
class Shape:
    def __init__(self, shape_type, dimensions):
        self.shape_type = shape_type
        self.dimensions = dimensions

    def area(self):
        if self.shape_type == "Circle":
            radius = self.dimensions[0]
            area = 3.14 * radius * radius
        elif self.shape_type == "Rectangle":
            length = self.dimensions[0]
```

```
            width = self.dimensions[1]
            area = length * width
        else:
            area = 0.0

        return area
```

Having recognized that we can use the type system, we can replace the `if` logic with polymorphism. We eliminate `shape_type` as a property, and create classes with names corresponding to their `shape_type` value: `shape_type` of `circle` becomes `class Circle`, for example.

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

In functional languages, depending on the type system, we can still do something similar by using dynamic dispatch, again associating behaviors with types rather than using conditional logic to emulate a type system. In this example, `Utility.area` will execute the correct behavior based on the type of the argument, so there is no need for an `if` statement.

```
require Math

defmodule Circle do
  defstruct [:radius]
end

defmodule Rectangle do
  defstruct [:length, :width]
end
```

```
defprotocol Utility do
  @type shape :: map()

  @spec area(shape) :: integer()
  def area(shape)
end

defimpl Utility, for: Circle do
  def area(shape), do: Math.pi * shape.radius * shape.radius
end

defimpl Utility, for: Rectangle do
  def area(shape), do: shape.length * shape.width
end

defmodule Main do
  IO.puts Utility.area %Circle{radius: 5}
  IO.puts Utility.area %Rectangle{length: 5, width: 4}
end
```

This is excellent, as we are able to add behaviors orthogonally. For example, if we need behavior to calculate the area of a hexagon, we would simply add another function, and not modify any existing code. This allows us to follow an established pattern to introduce new behavior, it reduces the risk of introducing bugs, and it eliminates the need to understand all the existing area functions before working with the code.

Registries

Registries combine values and polymorphism in a practical pattern for creating composable behaviors.

For example, we may have the following embedded `if` statements (using Python):

```
class DateFormatter:
    def __init__(self):
        self.format_name = "Local Time"

    def format(self, d):
        if self.format_name == "Local Time":
            return d.strftime("%d/%m/%Y %H:%M:%S")
        elif self.format_name == "UTC Time":
            return (d.astimezone(ZoneInfo("UTC"))
                .format("%d/%m/%Y %H:%M:%S"))
        elif self.format_name == "ISO Format":
            return d.isoformat()
```

The content of those conditional statements is complex enough that we cannot create a simple value map for them, but we *can* map polymorphic functions using a registry.

```
def _format_local(d):
    return d.strftime("%d/%m/%Y %H:%M:%S")
```

```
def _format_utc(d):
    return d.astimezone(ZoneInfo("UTC")).format("%d/%m/%Y %H:%M:%S")

def _format_iso(d):
    return d.isoformat()


class DateFormatter:
    def __init__(self):
        self.formats = {
            "Local Time": _format_local,
            "UTC Time": _format_utc,
            "ISO Format": _format_iso
        }
        self._format_name = "Local Time"

    @property
    def format_name(self):
        return self._format_name

    @format_name.setter
    def format_name(self, value):
        if value not in self.formats:
            allowed = ', '.join(self.formats.keys())
            msg = (f"Invalid format_name {value}, "
                   f"must be one of: {allowed}")
            raise ValueError(msg)
        self._format_name = value

    def format(self, d):
        return self.formats[self.format_name](d)
```

You will notice this code is longer than the example with `if` statements. This is because we added validation when setting `format_name` using the registry approach; it would have required additional `if` statements with hardcoded values in the prior example. Often, these trivial examples going from some bad practice to an improved rewrite involve an increase in the lines of code. This is not a bad thing. Code is written once (rewrites, after all, are different code that is also written once), but read many times.

> Let's Get Technical
>
> There are challenges called “code golf” where the objective is to accomplish a coding task in as few characters as possible. The solutions to these challenges are often completely incomprehensible without spending extensive time to understand them, much longer than it would take to understand a solution with more lines.

More lines that are easier to understand is better than fewer lines that cause confusion.

In some cases, we must work with complex conditionals. In those instances, the set of conditionals can be modelled via an array of combined types, where those types have some construction function associated with them.

As an example, let us create a `Repeater` class, which can repeat until either a `count`, `while`, or `duration` value is met, and will repeat until any of those conditions, should they be set, are met. Here is a naive implementation, using `if` statements:

```
class Repeater:
    def __init__(self, options):
        if 'while' in self.options:
            try:
                self.evaluator = PredicateEvaluator(self.options['while'])
            except EvaluationParseException as e:
                raise RepeaterException(
                    f"Cannot parse predicate exception " +
                    self.options['while']
                ) from e
        self.options = options
        self.iteration = 0
        self.start_time = time.time()

    def is_done(self, current_state):
        if 'count' in self.options:
            if self.options['count'] == self.iterations:
                return True
            else:
                self.iteration += 1

        if 'while' in self.options:
            return self.evaluator.evaluate(current_state)

        if 'duration' in self.options \
            and time.time() - self.start_time > self.options['duration']:
            return True

        return False
```

Note that by keeping the different behaviors in the same class, we have coupled the behaviors of each.

- There is a check for `while` at the beginning in order to create the evaluator, which is not used by the other two repetition types.
- Similarly, `iteration` and `start_time` are only used by their associated repetition types.
- Current state is always passed in, even though it is only used by `while`.
- The `count` repetition needs to internally modify state, and thus nests `if` statements.

All of this complexity is multiplied if we want to create a good `__str__` (`toString`) function for representing both the current state and objective state, or if we want a function to calculate an estimate of progress, both of which are entirely reasonable additions to a repetition system.

Contrast that with using polymorphism and a registry to solve the problem. In the following code, we create a model of repetition using policies, which have a single function to indicate whether or not they are done. Each policy has a "builder" function attached to it, and those "builder" functions are added to a registry. As a result, the registry is able to

construct whichever set of policies is needed, reliably, at runtime.

```
class RepetitionRegistry:
    def __init__(self, system):
        # For brevity, we are defining the policies here,
        # but it would be better to inject them during the building phase,
        # as we would not need access to "system" in that case.
        # See Chapter 4, "New".
        self.policy_creators = [
            CountPolicy.make,
            WhilePolicy.from_system(system),
            DurationPolicy.make,
        ]

    def policies(self, options):
        policies = []
        for creator in self.policy_creators:
            possibility = creator(options)
            if possibility is None:
                continue
            policies.append(possibility)

class CountPolicy:
    def __init__(self, count):
        self.count = count
        self.iteration = 0

    @classmethod
    def make(cls, options):
        if 'count' not in options:
            return None
        return cls(options['count'])

    def is_done(self):
        self.iteration += 1
        return self.iteration > self.count


class WhilePolicy:
    def __init__(self, evaluator):
        self.evaluator = evaluator

    @classmethod
    def from_system(cls, system):
        # Let's say `system` has a `get_state`
        # function to retrieve the current system state
        def make(options):
            if 'while' not in options:
                return None
            return cls(Evaluator(system, options['while']))
        return make
```

```
    def is_done(self):
        return self.evaluator.evaluate()


class DurationPolicy:
    def __init__(self, duration):
        self.done_at = time.time() + duration

    @classmethod
    def make(cls, options):
        if 'duration' not in options:
            return None
        return cls(options['duration'])

    def is_done(self):
        return time.time() > self.done_at
```

An interesting property emerges from this structure: before, the `is_done` function returned `True` if any of the policies were considered done, so it always used the quickest policy, which was inflexible. Now, we can could organize the policies in whatever way we want. For example:

> Let's Get Technical
>
> Despite this chapter cautioning you about using `if`, we use it multiple times in this example. That is because these conditionals are being used for construction, rather than behavioral logic. See "Good Uses of If" later in the chapter.

```
# Same behavior as before, done if any policies are done:
any([policy.is_done() for policy in policies])

# Require all policies to be done:
all([policy.is_done() for policy in policies])

# Require at least two to be done:
sum([policy.is_done() for policy in policies]) > 2
```

Additionally, it would be straightforward to add functions to the policies themselves to represent how they are configured (time remaining, iterations remaining, or a representation of the predicate function), or even their progress.

The key value of this model is that it scales horizontally, and could be combined orthogonally with other policies and registries. For example, perhaps there are policies that dictate which actions must be executed on a given iteration, or the order of those actions; that functionality could be added without any modification to the functions for controlling the repetition behavior.

Writing code like this is what allows us to be wrong in correctable ways.

Switch Statements and Ternaries

Ultimately, the problem with `if` statements is not actually one associated with the keyword. Rather, it stems from the use of conditional logic to control program flow. The correct practice is instead to control flow using program structure and types.

Switches, in particular, can almost always be rewritten to use polymorphism and registries to great effect. By their nature, switch statements operate on singular value types, providing a strong signal to the developer that the problem would be better modeled with polymorphism.

Using switch statements generally results in functions that must do multiple things, and as a result have long parameter lists to accommodate all the potential things they could do.

```
function area(
    shape, side, length, width,
    radius, minor_axis, major_axis
) {
    let area = null;
    switch(shape) {
        case 'square':
            area = side * side;
            break;
        case 'rectangle':
            area = length * width;
            break;
        case 'circle':
            area = Math.PI * radius * radius;
            break;
        case 'oval':
            area = Math.PI * minor_axis/2 * major_axis/2;
            break;
    }
    return area;
}

area('rectangle', null, 3, 4, null, null, null);
```

Although the above Javascript example is extreme when it comes to unused parameters, this is a common side effect of using conditionals rather than polymorphism: functions require data that will only be used when a particular conditional path is hit. This decreases readability and increases MTC for maintainers.

Instead, we can separate out the functionality and again use a registry to allow the caller to choose the behavior. By doing this, we limit the parameters to only the ones that are necessary, and we ensure that adding new behaviors only requires an addition to the registry, without altering existing code.

```
class Square {
    constructor(side) {
        this.side = side;
    }

    area() {
        return this.side * this.side;
    }
}

class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    area() {
        return this.length * this.width;
    }
}

class Circle {
    constructor(radius) {
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius * this.radius;
    }
}

class Oval {
    constructor(minor_axis, major_axis) {
        this.minor_axis = minor_axis;
        this.major_axis = major_axis;
    }

    area() {
        return Math.PI * this.minor_axis/2 * this.major_axis/2;
    }
}

const shapes = {
    Square,
    Rectangle,
    Circle,
    Oval
}

(new shapes['rectangle'](length, width)).area();
```

As before, we can attach additional reporting or related functions to the types that we have created.

Choosing A Model

The process of going from `if` statements to polymorphism involves creating behavioral models, which is the representation in code of how users can interact with the system, and how to enhance the system to enable new interactions. Finding correct models can be challenging, as there may be multiple ways to pull apart the functionality. In such a case, the “Rule of Three” can be a helpful guide: often a behavior cannot be modeled correctly until there are at least three known sub-behaviors. In the same way that three points define a plane, or three locations are needed to triangulate a position, two are usually insufficient to establish a trend and determine the model.

In cases where the correct model is unclear, it may be preferable to keep a single conditional and an inaccurate but simple model, rather than attempting to create a polymorphic model with so little data. For example, it might not be clear how to create the `membership_levels` if there are only two levels initially: `premium` and `default`. However, when a third level is added, it becomes clear that the choice is not binary (few choices are), and in fact there is a set of options; therefore, the code must be structured accordingly.

When there are many unknowns, do not try to come up with the most comprehensive solution. Instead, recognize that good design emerges from deep understanding of the problem space, and that understanding may take time. Early in a project, prefer simplicity and rigorously reduce MTC. As the project progresses, pay attention to the ways in which the behaviors and code interactions change, and consider how to improve the underlying models.

If the rules in this book are followed, refactoring the code later should not be too difficult. What is difficult, however, is for the maintainer to know that the code should be refactored. I recommend leaving a comment for future programmers describing the intent to delay creating a behavioral model until more information is available.

Focus on keeping the code clear, and even if the initial design is not perfect, it can be corrected later.

Good Uses of If

Despite how commonly `if` and conditionals are misused, there are still many valid, important uses.

Input validation is a common one:

```
def validate(user_input):
    choices = ['red', 'blue', 'green']
    if user_input not in choices:
        raise InvalidInputError(
            f"Must select one of {', '.join(choices)}," +
            " not \"{user_input}\"")
```

This type of validation should happen as close to the user input as possible, ideally immediately after the input is received. Often, it should also result in the creation of a typed value, which can then be used by the rest of the program rather than the raw input.

Another good use of `if` is during the object construction phase, to determine how to build the required object, or whether it should be built at all. In the following example, we have `Sequence` classes we construct based on provided options.

```
# These classes only show the "make" function,
# to demonstrate good uses of "if"

class RandomSequence:
    @classmethod
    def make(cls, options):
        if options.get('sequence') != 'random':
            return None
        return cls()


class ReverseSequence:
    @classmethod
    def make(cls, options):
        if options.get('sequence') != 'reverse':
            return None
        return cls()
```

The `if` statements in those classes are used for building the object, not as part of the business logic, and are thus good uses of `if`.

The examples in the “Registries” section also demonstrate both the validation use case, and the construction use case.

Using `if` in algorithms is also entirely reasonable.

In Chapter 4, “New,” we created a class for matching against file contents. Let’s imagine we need another matcher, which checks if a given character is used more than some number of times.

```
class CharMinUsageMatcher:
    def __init__(self, character, threshold):
        self.character = character
        self.threshold = threshold

    def match(self, contents):
        count = 0
        for c in contents:
            if c == self.character:
                count += 1
            if count > self.threshold:
                return True
        return False
```

In the above example, we use `if` algorithmically to determine if we have matched the given character, and if we have crossed the given threshold.

Conclusion

While `if` statements and conditionals may not be inherently bad, their use should be carefully considered. Most often, they are being used instead of the languages provided type system.

The use of polymorphism, mapped values, and registries to accomplish the same goal results in code that is far more flexible, scalable, readable, and critically, easier to understand as the requirements become more complex. As projects grow, conditionals always result in significantly more complexity than polymorphism to accomplish the same task.

One of the most reliable ways to determine whether a section of code will have many bugs is by looking at how many `if` statements it has. Avoid `if` .

---

7. Naming

> Names are deeply meaningful to your brain, and misleading names add chaos to your code.
>
> *Andrew Hunt, The Pragmatic Programmer: From Journeyman to Master*

Rule: Think Carefully About Names

Names should aspire to create the correct semantic map in the reader's mind with as few characters as possible. Names must be as consistent as possible.

Names are jokingly said to be one of the hardest problems in programming. Like most jokes however, the statement isn't entirely spurious; choosing good names can immediately clarify a problem, as the problem becomes associated with the ideas behind the name. Choosing bad names can force programmers to think about a problem incorrectly, wasting time or even causing bugs.

Given the importance of names and how they can provide an immediate framework of ideas if executed well, they are one of the four pillars of communication (see Chapter 1, "Introduction").

Semantic Mappings

Let's look at two examples to demonstrate the power of names. Does this code make a network request?

```
user = user_manager.get(user_id)
```

What about this code?

```
user = user_manager.fetch(user_id)
```

Of the two, most people believe the latter one is more likely to make a network request. The term *get* is entirely generic, and simply means *to retrieve something* with no notion of proximity. However, *fetch* brings to mind the idea of a dog chasing a stick after it has been thrown, and bringing it back. It is the act of leaving to retrieve something, and then returning with the item. As network requests involve data leaving the system, and usually result in data being brought back in the form of a response, fetch leverages our *semantic mapping* of behavior.

Semantic mappings are the connections between our understanding of the meaning of language, and our understanding of some synthetic construct. When creating names in code, we want to leverage semantic mappings as much as possible, guiding readers to the most likely interpretation and purpose of the code. If we fail to do so, readers will often apply incorrect assumptions to what the code does. In the best case, this will cost them time and cause frustration; in the worst case, it will cause critical bugs as functions are misused. In the example with “fetch” vs “get”, the reader could reasonably assume “get” retrieves local data, and be surprised when there is unexpected latency or errors while calling the function. Similar issues arise with “set” vs. “store”, “subscribe” vs. “stream”, “create” vs. “open” vs. “connect”, and other similar words.

A Rule of Thumb

One of the best ways to choose a good name is to describe the purpose of the code to someone. During the description, some key words will be used. Those words can be used to create a good name.

For example, take the following description:

> *So, this function takes a path to some diagnostic data, and based on the directory and file names it parses out a bunch of metadata details about the diagnostic data. It constructs an object with that data and returns it.*

They key words are the descriptive ones: path, diagnostic, parses, metadata details, and constructs. We could call the function `ParseDiagnosticPathAndConstructMetadataDetails` but that is so long that it is difficult to read. Instead, reduce it to only the most critical information: it determines diagnostic metadata from a path.

Perhaps `diag_info_from_path`. Its still a bit long, and the abbreviation “diag” is used, but it is nicely descriptive. If the function was attached to a `DiagInfoBuilder` class, it could be simplified to `from_path`, as the `diag_info` part is implied by the classname.

The process of describing the thing in plain english and identifying the key words is the most useful tool in creating good names. If the words used to name something are not used in its description, consider updating the name.

Consistency

The most important rule for names is *be consistent*. A bad name used everywhere must only be learned once; a good name used everywhere except a few places must be learned multiple times. Inconsistent names create higher cognitive load and increase MTC.

Consider the following Python example where the same concept has multiple names.

```
def process_all(users):
    for person in users:
        person.process()

def main():
    admins = load_records()
    process_all(admins)
```

The inconsistency in the above example forces the reader to memorize what a variable is called and in what context. If instead the name is consistent, it is much simpler to follow what is happening.

```
def process_all(admins):
    for admin in admins:
        admin.process()

def main():
    admins = load_admins()
    process_all(admins)
```

Even if the name was bad and had no semantic mapping (for example, `x`), being consistent means once we learn what `x` is, we no longer need to continue learning all the new names it might have.

Additionally, names should match the style that surrounds them. Consistency means that if a convention used in one specific function is different from the rest of the project, it is *more important* to match the style of that function than the style of the project. The priority is: function, class, module, package, project. It can be helpful to update code to make style conventions consistent across a project, but it should not be done as part of a behavioral change, as that makes it difficult for readers looking at the modifications to determine what is behavioral and what is stylistic. Instead, when updating style, create separate commits (if working in Git) that only contain stylistic changes. Additionally, recognize that there is a tradeoff: non-behavioral changes will impact the project history when examining it through version control.

Names and Access

The Law of Demeter (LoD, discussed in Chapter 10, “Refactoring”) states that code must ask for all of its dependencies directly, instead of requiring a container and then reaching inside it to pull out the actual dependencies.

Part of this has to do with the value of names and access. As LoD forces us to choose names when creating a function signature, we enhance communication: a function signature with `logger, auth_service, user_report` communicates much more than a function that simply requires `dependencies`.

```
def execute(dependencies):
    pass


def execute(logger, auth_service, user_report):
    pass
```

The first example communicates nothing: all arguments are already dependencies; no reader has a clue what is actually required to run the function.

The second example uses meaningful names. If those names are consistent throughout the application, the requirements for the function are much more obvious. In typed languages, requirements are even more explicit, since the reader knows precisely the type of each argument.

Affixes

Affixes are either prefixes or suffixes, and are used in names to help communicate purpose. Two common affix types are Hungarian notation, and namespaces. Let’s consider each of these in turn.

The Role of Type Safety

Types are one of the best ways programmers have to communicate information. However, the support for types varies wildly between languages.

To help convey important information in the absence of type support, there are two prefix conventions: Systems Hungarian, and Apps Hungarian.

Systems Hungarian

In Systems Hungarian, variable names are prefixed with physical type indicators, such as `str_name` or `int_count`, related to how the data is stored physically in memory. In languages with weak type safety, this can be very helpful.

For example, we can use it to communicate the type of argument a particular function needs in code:

```
def process(int_timeout):
    """
    Process, but timeout if it takes too long
    """
```

In large codebases, it can be useful to have the physical type information associated with the variable. However, Systems Hungarian should not be used in languages with strong type safety, where the physical type information is already associated with the variable.

For example, in Golang:

```
func Process(intTimeout int) {}
```

In such cases, its use is redundant and distracting.

Apps Hungarian

Apps Hungarian affixes variable names with their logical type, or semantic information, meant to indicate the variable’s purpose or intended use.

For example:

```
def process(timeout_ms: int):
    """
    Process, but timeout if it takes too long
    """
```

In the above example, we use Apps Hungarian to denote the timeout value is in milliseconds (logical information), and use Python type hinting to indicate it is an int (physical information).

Apps Hungarian is helpful in languages that do not support logical type aliasing. Communicating the intended purpose of a variable through its name helps the reader determine whether it is being properly used, though the programmer must take care to be consistent with the Apps Hungarian prefix that is used.

Some languages (such as Haskell) allow the aliasing of physical types such that new, logical types can be created. Apps Hungarian is unnecessary in those languages as an appropriate logical type can be created instead. This has the benefit of the compiler or runtime being able to enforce consistency. However, even in languages with sufficient type support, creating a new type for every distinct permutation of purpose can lead to code bloat and confusion, and should thus be used judiciously.

Let’s examine the difference between aliasing and Apps Hungarian in the following examples.

In this Python example, we denote whether the string is sanitized or not by including it in the variable name using Apps Hungarian.

```
raw_input = input("What is your username?")
sanitized_sql_input = sql_sanitize(raw_input)
```

In the following Go example, we create a new type to indicate if it is sanitized, and then our function parameters can specify whether they expect raw strings, or sanitized strings.

```
package main

import "fmt"

type RawInput string
type SanitizedSQL string

func Sanitize(input RawInput) SanitizedSQL {
    return SanitizedSQL(input)
}

func main() {
    var input RawInput

    fmt.Scanln(&input)

    username := Sanitize(input)

    fmt.Println(username)
}
```

Both of these examples are reasonable ways to convey necessary information, and which approach to use is dependent on the problem being solved and the capabilities available.

The primary consideration in using Hungarian notation is MTC: does the affix increase or decrease the time it takes for a reader to understand the intent of the code? The answer will not always be the same, so it is important to consider the MTC impact before applying the notation to your code. Having covered Hungarian notation, let’s move on to consider the other affix type: namespaces.

Let's Get Technical

Go also has type aliases, in the form:

```
// the "=" indicates an alias
declaration
type A = T
```

Had we used a type alias instead of a type definition, the Go compiler would have allowed all matching types.

In that case, the type alias is purely to improve communication to the reader.

Namespaces

Packages, libraries, modules, classes, and functions are all forms of namespaces, which use *lexical scope* (refer to the Chapter 3, “State,” discussed previously). Namespaces allow us to structure code hierarchically, ensuring that using a name in one location does not conflict with using that same name in another location.

Code within a namespace should generally not be prefixed with the name of that namespace.

Let's Get Technical

Some programming languages have explicit constructs called "Namespaces". These are often general-purpose ways of isolating variable names. While they may use the term "namespace", it applies more generally to any construct which isolates named variables from each other.

Take, for example, the Python module `pathlib`:

```
import pathlib

pathlib_path = pathlib.PathlibPath(".")
```

- This is repetitive, forcing the programmer to type more and the program to be longer than necessary, without decreasing MTC.
- If the namespace is renamed, all its members must correspondingly be updated.
- Naming like this distracts from the underlying purpose of the code. When we repeat ourselves, it adds noise, and what the reader cares about becomes lost.

Fortunately, this is not the API of `pathlib`. In fact, very few standard libraries have this sort of redundancy in them.

```
import pathlib

path = pathlib.Path(".")
```

Avoiding repetitive names does not just apply at the module level. Repetition may be present in any namespace, even within a function, and should be corrected. The following is an extreme example of this bad practice:

```
# service.py

class ServiceAPI:
    def service_api_get(self, request):
        service_api_get_response = self._fetch_response(request)
        return service_api_get_response

# main.py
import service

response = service.ServiceAPI().service_api_get(request)
```

Attempting to read the preceding code aloud helps drive home the point: does it sound silly to repeat the words “service” and “API” so many times? Yes - and therefore it is probably equally silly to write the code this way.

A corrected version may look like this:

```
# service.py

class API:
    def get(self, request):
        return self._fetch_response(request)

# main.py
import service

response = service.API().get(request)
```

Note that we are using the term `get` rather than `fetch`, despite `fetch` having the stronger semantic mapping. This is because the context of the term is important. There is an HTTP method `GET`, and if we are constructing an API client that uses HTTP, we likely want to communicate that we are literally using the HTTP `GET` method by calling our API method `get`.

Like with all rules we’ve discussed, on occasion it may make sense to break this one. Sometimes many elements within different modules will share the same general names. In this case, to help avoid confusion across the project, it can make sense to prefix those elements with the related namespace purpose, so that searching across the project can yield more specific results. However, if you find yourself in this situation, it is worthwhile to think about how the code is constructed and whether there are clearer, less general names within the namespaces that could be used.

Conclusion

Coming up with good names can be really difficult, but is always important. Good names can dramatically reduce MTC, and bad names will invariably increase it.

Time spent thinking of a good name is almost always worth it, and often the process is easier than we believe if we approach it systematically. Leverage semantic mappings to communicate intent, and naming will become much easier.

Above all, be consistent. A reader having to learn a bad name is already bad; having to learn a bad name, and then having that name change into an okay name (much to the reader's inevitable confusion), is even worse.

---

8. Documentation

> Incorrect documentation is often worse than no documentation.
>
> *Bertrand Meyer*

Rule: Document Usage and Oddities

Documentation in code should focus on how to use the code correctly, or why odd code exists, but avoid detailing exactly what the code is doing.

Documentation embedded in source code is referred to as “comments”. Though comments may not be recognized by the compiler or language runtime, they are an integral part of the code. They are the fourth pillar of communication and are the only free-form communication mechanism available to programmers. Because of that flexibility, they can be both incredibly valuable, and incredibly dangerous.

Doc Blocks

Doc blocks are comments attached to functions, classes, modules, namespaces, or other multi-statement constructs, and are generally multi-line, creating a “block” of text, as seen in the following Python example.

```
def hello(value):
    """
    Prints the given value, prefixed by "hello, "
    """
    print(f"hello, {value}")
```

One of the dangers of doc blocks, or any type of comment for that matter, is the lack of enforced accuracy. By its very nature, code is an accurate representation of what it does. It is possible for names to be misleading, but reading the code will result in understanding precisely, specifically, what its execution will do.

Comments do not behave like this. Nothing ensures the comments are truthful, and in fact time often creates “accuracy drift”, as code is updated while comments are not:

```
def hello(value):
    """
    Prints the given value, prefixed by "hello, "
    """
    print(f"Hello, {value}")
```

Now, if we read the comment but not the code, we expect the output of `hello("world!")` to be “hello, world!”, when in fact it would be “Hello, world!”. A subtle difference, but in critical systems accuracy drift creates bugs, confusion, and miscommunication.

Fortunately, there is quite a bit of tooling to help us when it comes to doc blocks. This tooling parses the signature of the identifier (often a function or class), and generates default documentation which is both human readable and partially machine readable. This allows IDEs to highlight when our documentation is suffering from identifiable forms of accuracy drift.

Here is an example:

```
def add_two(num):
    """
    Adds two to the given number.

    Args:
        num (int): The number to increment by two.

    Returns:
        int: The incremented value.
    """
    return num + 2
```

In this case, the documented argument and return names, their existence, and their type information can all be verified. IDE tooling can then highlight areas that are out of sync as a gentle reminder to correct the docs.

For Python, Elixir, Lisp, Haskell, Clojure, and some other languages, doc block notation like this is included with the code as a “doc string”. It is attached as a property somewhere, and is then interrogable by the code itself.

For example, if we were to run `help(add_two)` within a Python shell, we would see the output of the doc string we attached to the function.

```
>>> help(add_two)
Help on function add_two in module __main__:

add_two(num)
    Adds two to the given number.
```

```
    Args:
        num (int): The number to increment by two.

    Returns:
        int: The incremented value.
```

Many languages also have support for “doctests”, where examples in their doc blocks can be introspected, executed, and validated. This further helps prevent accuracy drift.

Inline Comments

Inline comments are comments occurring just before or on the same line as a statement of code, with the intention of explaining that line, and perhaps several of the following lines as well.

In most instances, inline comments should be avoided. Generally, they indicate the code is overly confusing and complicated, and perhaps not well thought-out. Additionally, as there is very little accessible tooling around inline comments, they are even more prone to accuracy drift than doc blocks.

A bad use of inline comments may look something like the following:

```
def inscrutable(a, b, c):
    # Takes the list of lists from a, filtering out the false-y values,
    # and then multiplying them by b to assign them
    # as the key in a dict for value c
    return {(v * b): c for x in filter(lambda a: a, a) for v in x}
```

Clearly, the MTC of this code is very high, and the comment is being used to try and decrease it.

Instead, we should *use the code itself to reduce MTC*, by refactoring it. In this case, that means adding more lines of code, where each line prioritizes *brevity*.

> Let's Get Technical
>
> Some readers may note that we are mutating the state of the result, despite being advised against this in Chapter 3, "State". This is true, and an important observation. In the tradeoff between mutation and MTC, assuming the mutation is carefully isolated, we should prioritize the reduction of MTC.

```
def scrutable(matrix, key_multiple, value):
    result = {}
    for sublist in matrix:
        for v in sublist:
            key = v * key_multiple
            result[key] = value
    return value
```

The following is another improper use of inline comments, as it describes what the code *does* rather than *why it does it*.

```
def apply_standard_percentage(b):
    return b * 0.2  # Gets the standard 20%
```

In such cases, doc blocks should certainly be used instead. It is downright horrific when those comments start to suffer from accuracy drift, and the programmer reading them doesn't realize it:

```
def apply_standard_percentage(b):
    return b + 0.3  # Gets the standard 20%
```

When this happens, it may take hours or even days for the programmer to realize the program is not behaving as expected due to inaccurate comments.

Not all inline comments are bad. They are helpful and necessary when there is critical information out of the programmer's control that must be communicated. For example, documenting why a particular decision was made when another one seems more reasonable:

```
def process(api, task):
    prepared_task = prepare_task(task)
    # There is an active bug in the API lib tracked here:
    # <link>
    # The workaround is to manually initialize and submit the task.
    initialized_task = api._initialize_task(prepared_task)
    api._submit_task(initialized_task)
```

This comment is not about *what* we are doing, but about *why* we are doing it. That is where the value of comments shines: when the decision-making reasoning must be expressed. Code (assuming it has been written properly) already tells us "what", more precisely than we could ever do with human language. However, it lacks information on the motivation and reasoning behind the decisions. That information is "out-of-band"; it exists in the programmer's brain (hopefully), and can only be expressed through normal human language channels.

When bugs, odd APIs, or unexpected values force us to write surprising code, documenting that fact is crucial, and inline comments are an excellent way to do so.

Another reason to add inline comments is when some piece of code contains *inherent complexity*, complexity that necessarily exists within the problem, and we cannot write our code differently and make it disappear.

```
void complex(int a, int b) {
    // Assume more code here...

    // Swap the values of a and b without using a tmp variable,
    // as we have limited memory
    a = a ^ b;
```

```
    b = a ^ b;
    a = a ^ b;

    // And more code here...
}
```

Explaining the reasoning for the use of esoteric syntax is incredibly helpful for decreasing MTC.

Conclusion

Documentation is the most powerful out-of-band communication method we have, but can easily be abused. Be meticulous with your docs, and future programmers will thank you. Be frivolous and they will curse you.

If you spend five minutes writing documentation to describe your intent and save a future programmer an hour, it is time well spent. If you spend two minutes updating the documentation in a section of code to ensure it remains accurate, and save a future programmer a day, it is definitely worth it.

If you have ever read code and thought, “What were they thinking?”, well, documentation is your opportunity to tell a future reader the answer.

---

9. Unit Testing

> Write tests until fear is transformed into boredom.
>
> *Kent Beck, Test-Driven Development: By Example*

Rule: Unit Test Complexity or Contracts

Unit tests should focus exclusively either on ensuring complex code operates correctly, or that high-level contracts (APIs) are adhered to. Avoid directly testing simple internal functions whose implementation may change.

Unit testing consists of tests that can execute within the same process space as the application. In particular, they are responsible for instantiating parts of the application, then executing those parts and evaluating the results. The ability to do this is powerful: it lets us exercise the internal functionality of the application without relying on the application's external dependencies. Removing the need for databases, file systems, external application programming interfaces (APIs), and other dependencies greatly simplifies the setup required to verify the program, and allows that verification to run quickly.

Other forms of testing tend to treat the application as a black box: they start the application and then interact with it externally, evaluating its output, but never look at what is happening inside. Unit testing allows us to peek inside the program and ensure all the parts are operating as expected.

There is a caveat, however: no amount of unit testing can replace running an application in a real environment and performing external validation against its features.

As effective as unit testing is, additional forms of testing is still required.

Testable Code

Many of the structural rules we have covered will result in creating testable code: separating builder and business objects, using polymorphism, avoiding state mutation, avoiding `if` statements, etc.

When writing new tests, the value of these practices becomes apparent.

Imagine the scenario where we fail to follow these rules:

```
class Config:
    def __init__(self, url):
        res = requests.get(url)
        self.data = res.json()['config']


class Difficult:
    def __init__(self):
        self.config = Config()
```

When we try to test our `Difficult` class, how do we instantiate it?

```
difficult = Difficult()  # We made an HTTP request here!
```

Without intending to, we have made an HTTP request just by trying to construct an object we want to evaluate.

Instead, we should follow the principles from Chapter 4, “New,” and inject our dependencies:

```
class Config:
    def __init__(self, data):
        self.data = data

    @classmethod
    def from_url(cls, url):
        res = requests.get(url)
        return cls(res.json()['config'])


class Easy:
    def __init__(self, config):
        self.config = config
```

Constructing the object to test is now trivial:

```
easy = Easy(Config({"value": 123}))
```

Test Doubles

Test doubles are simulated components within the application, which let the application function without potential side effects or dependencies on external services or processes. Because this decoupling of the application is required for unit testing, test doubles are a necessary element of writing unit tests.

Test doubles should be used carefully, as they replace *real coupling* with *simulated coupling*. This means that code using test doubles will no longer be able to detect if their dependencies have changed in a breaking way. This is exacerbated if the code being tested violates the LoD (Law of Demeter, see chapter 10, “Refactoring”), and is tightly coupled to the internal structure of its dependencies.

There are four types of test doubles: dummies, stubs, mocks, and fakes. We will examine them using the following snippet of Python code, by injecting various test doubles as input for the `api` parameter:

```
def process(task, api):
    try:
        values = task['values']
    except KeyError:
        raise MissingValuesException(task)
    api.send(task)
    return api.fetch(task['id'])
```

Dummies

Dummies are values required by the underlying code (generally through function signature or class property), but not used by it. They are the simplest of the test doubles, and often a `null` / `nil` / `None` value is sufficient to allow the test to execute without error.

```
class Test(unittest.TestCase):
    def test_process_exc(self):
        task = {}
        with self.assertRaises(MissingValuesException) as ctx:
            process(task, None)
```

Here, we pass `None` as our dummy value for `api` since we should not even reach the call to `api.send` in the code. Using `None` communicates to the reader that it is an unnecessary value for the case we are testing.

> Let's Get Technical
>
> Passing a null value is not possible in every language; particularly strongly typed languages such as Typescript, where in order to pass a null value the function signature must allow it. In such cases, do not modify the function signature only to accommodate the tests. Instead, construct a stub instance (discussed next), and document within the test that the stub is not used.

Stubs

Stubs are dummy objects that implement the same methods as “real” objects, but those methods do nothing other than adhere to the required interface.

Assuming we expect `api.fetch` to return a list of numbers, we create a `StubAPI` with a `fetch` function, which adheres to the interface by returning an empty list:

```
class StubAPI:
    def send(self, task):
        return None
```

```
    def fetch(self, id):
        return []


class Test(unittest.TestCase):
    def test_process(self):
        task = {"values": [1,2,3], id: 5}
        ret = process(task, StubAPI())
        self.assertEqual(ret, [])
```

In this case, the stub adheres to the contract: a list is required, so it returns an empty list, which is the simplest return value possible. Subs are helpful when the code we are testing requires data that strictly adheres to the contract, but we do not need any introspection beyond that. Often, stubs are created for use in a single unit test, or perhaps a small subset of tests concerned with validating a specific feature.

Mocks

Mocks go one step further than stubs: they additionally record internally what has happened, such that they can be interrogated later by the test to ensure that everything proceeded as expected.

They are most commonly used for database, filesystem, and protocol-based API doubling. They may be used by smaller subsets of unit tests, or across all tests, depending on the generality of the behavior being mocked. In this example, the `MockAPI` records data about how it was used, such that the test can check that it was used as expected.

```
class MockAPI:
    def __init__(self):
        self.task_stack = []
        self.id_stack = []

    def send(self, task):
        self.task_stack.append(task)

    def fetch(self, id):
        self.id_stack.append(id)
        return []


class Test(unittest.TestCase):
    def test_process(self):
        mock = MockAPI()

        task = {"values": [1,2,3]}
        ret = process(task, mock)
        self.assertEqual(ret, [])
        self.assertDictEqual(mock.task_stack[0], task)
        self.assertEqual(mock.id_stack[0], 5)
```

Notice how, although we can evaluate precisely what was called, and in what order, there is still a disconnect: the returned values will always be empty, regardless of what we pass in.

Fakes

Fakes are the most sophisticated of the test doubles. They simulate the doubled component entirely, such that interacting with them feels identical to interacting with the real thing. Data may be cached locally and then retrieved, updated properly, and internal representations maintained to create the illusion of the actual component.

In this example we create a `FakeAPI` that internally caches the data it receives, and can then respond with that same data:

```
class FakeAPI:
    def __init__(self):
        self.values = {}

    def send(self, task):
        self.values[task['id']] = task['values']

    def fetch(self, id):
        return self.values[id]


class Test(unittest.TestCase):
    def test_process(self):
        mock = FakeAPI()

        task = {"values": [1,2,3]}
        ret = process(task, mock)
        self.assertEqual(ret, [1,2,3])
```

While dummies, stubs, and even occasionally mocks are created for use within a particular unit test, or maybe a small subset of unit tests, fakes are generally created for use in many different unit tests across the application. If you have critical interactions with external services (e.g. databases) that must be rigorously tested, fakes may be the best approach. They have a high initial cost to develop, though, as they must mimic the service precisely, and to remain valid they must also maintain behavioral consistency with the service they are faking.

Unit Test Placement

Unit tests should be placed as close to the code they are testing as possible, while still residing in a separate file (in the case of the language Smalltalk, within the same package). For many languages, this means suffixing the source code file name with either `.test` or `_test`, and placing it in the same directory as the source being tested. For example, if we have `service/api.py`, we may have `service/api_test.py`. There are multiple benefits to this.

- It is easy to find the related source code.
- The import structure for the source code into the test code tends to be simple.
- It is obvious which files have corresponding test files and which do not.

- For some languages, even the process of loading code from the same directory results in exercising the structure of the project, so there is validation that happens for “free.”

Test doubles should be placed within the module or package most closely related to the thing they are doubling. If they are so generic as to be application-wide, they should be placed in a common location accessible by all the tests.

Test Objectives: Temporary vs Permanent

Unit tests have the delightful ability to exercise code in a controlled, targeted manner. This allows programmers to use unit testing somewhat like a REPL (read-evaluate-print-loop); perhaps we could call this a TERL (test-execute-refactor-loop). We can write a test to exercise the behavior of a low-level part of our system, and observe that it behaves as expected.

Additionally, we can use these tests to ensure the behavior of the refactored code remains constant (see chapter 10, “Refactoring”).

The tests that we use to interrogate the low-level functions and behaviors of our application, are *temporary unit tests*. They assist in the development and refactoring process, and must either be removed or converted to satisfy one of the permanent unit test objectives.

Permanent unit tests have only two potential objectives:

1. Verify that abnormally complex logic behaves properly.
2. Verify the code adheres to its externally-facing contracts.

Unless the unit tests fulfill one of these two objectives, they are extraneous. This means any tests that exclusively execute against internal contracts (private APIs which are not visible, accessible, or meant to be used by the consumers of the code) should not make it into the published, production version of the project. If they do, they will hamper development, increase the likelihood that the tests fail despite the application functioning properly, and cause testing to be considered unreliable.

After all, those sorts of tests will fail if we change the internal structure of the code, *even if we still honor the external contract*. This increases the work to refactor and improve code, since not only must the code be altered, but the corresponding tests must be updated as well. It also muddies the waters: readers may be unclear on which tests are vital and validate real, important functionality, and which are simple, rote tests that validate nothing related to the purpose of the application.

Here is an example of a Python test with an improper objective:

```
class Processor:
    def __init__(self, logger):
        self.logger = logger

    def process(self, task):
        self._log(task)
        # Continue on with processing

    def _log(self, task):
        self.logger.info(f"Processing task {task.name}")
```

```
class ProcessorTest(unittest.TestCase):
    def test_log(self):
        l = MockLogger()
        p = Processor(l)

        p._log(DummyTask("updateRecords"))

        self.assertEqual(
            l.get_message(0),
            "Processing task updateRecords"
        )
```

This test may have started out as a temporary test for internal development purposes, but was not removed or converted, and is now a liability.

Imagine we have realized we need to change the internal API of the processor, and decide to offload the logging to another class `TaskLogger`. This change has no impact on the publicly available `process` function, yet our tests will break unexpectedly, because we are validating code that is purely internal. It is then unclear which tests have discovered a legitimate issue, and which have broken because they are unnecessary and testing the internals. Such situations can lead to "failure fatigue," where test results are ignored because even small changes that are bug-free result in failures.

There is an additional downside to superfluous tests: most unit test frameworks have related coverage tools that indicate which lines of code that have been executed and which have not. While not a perfect metric, it is useful to determine a rough level of confidence in the code, since you know how much was able to run without crashing and seemingly returns correct results.

However, writing bad unit tests skews the perception around coverage. They can inflate the coverage value by targeting internal functions and never validating the path that the code takes to actually use those internal functions. This can hide "dead" code, i.e. code that should be removed as it is not actually possible to execute when running the application normally. Alternatively, bad tests can give false confidence that the application has been rigorously tested, when in fact the connections and calling mechanisms are faulty.

The exception to the rule about testing internal contracts is Complexity Tests.

Complexity Tests

Complexity tests are used to verify a particularly complicated bit of internal logic that is difficult to reason about. This difficult logic should not be the result of the programmer introducing accidental complexity, but rather complexity that is inherent to the problem space. Most often, if a temporary test should become a permanent one, it will be converted to a complexity test.

A good rule of thumb to determine if a complexity test is required is to look at the data inputs and outputs. If the function is transforming data in ways that are not simple to reason about, it likely needs a complexity test.

A common example of this is protocols: the code may be implementing a particular protocol specification, and many tests are needed to validate that the protocol is being implemented properly. Parsing is another common example: often raw data needs to be consumed, and there is complicated logic to properly transform it into an internal representation.

```
from dataclasses import dataclass
from typing import Dict

@dataclass
class User:
    id: int
    name: str
    email: str
    labels: Dict[str, str]

    @classmethod
    def from_string(cls, raw):
        user_id, _, remainder = raw.partition(":")
        attributes = remainder.split(",")
        labels = {}
        for attribute in attributes:
            key, _, value = attribute.partition("=")
            labels[key] = value
        name = labels.pop("name")
        email = labels.pop("email")
        return cls(id=user_id, name=name, email=email, labels=labels)
```

The above code is not highly complex, but is complicated enough to justify the creation of a unit test. To ensure that the code for corner cases, it would be prudent to call it with a variety of inputs, both valid and invalid, and check that it does the correct thing.

```
class UserTest(unittest.TestCase):
    def test_from_string(self):
        raw = ("123:name=John,email=john@site.org," +
               "job=programmer,education=stanford")

        user = User.from_string(raw)

        self.assertEqual(user.id, 123)
        self.assertEqual(user.name, "john")
        self.assertEqual(user.email, "john@site.org")
        self.assertDictEqual(user.labels, {
            "job": "programmer",
            "education": "stanford"
        })
```

If possible, real-world inputs should be captured, and then incorporated as part of the tests.

Contract Testing

Contract testing is specifically about testing the *externally facing* contracts provided by the package or application.

To determine whether the code is externally facing, see if it is an entry point that will be used by the consumer of the codebase. Those entry points are external API endpoints, user-facing features, or functionality exposed by a package.

We will cover packages and libraries, as well as testing applications intended for machine use, and applications intended for human use.

> Let's Get Technical
>
> Sometimes, when an application is large enough, it is developed with modules that behave as internal packages. These are high-level subsystems within the application that expose their own contracts, and are often owned by different groups. Despite those sections of code being embedded in the application, they should be treated as packages, and the rules for contract testing are applicable.

Packages and Libraries

Packages and libraries are code projects that may be embedded within an application, and are responsible for some particular behavior. For example, we may provide a package with an API for interacting with our automated email service:

```
class Mailer:
    def __init__(self, server_url: str, sender: str):
        self.server_url = server_url
        self.sender = sender

    def send(to, email, cc=None, bcc=None):
        """
        Sends the specified email to the given addresses

        Args:
            to (:obj:`list` of :obj:`str`):
                List of email addresses
            email (:obj:`Email`):
                Email to send
            cc (:obj:`list` of :obj:`str`, optional):
                List of CC email addresses
            bcc (:obj:`list` of :obj:`str`, optional):
                List of BCC email addresses
        """
        # Code here...
```

Internally, `send` likely calls other functions, perhaps formatting the email object or validating the address structures, before actually making the SMTP calls to send the email. In this case, `send` is the contract we are providing. It is the function we expose to the consumers of our codebase, and is therefore the function we should be writing a test for. We should be able to reach any internal functions it calls by varying its input.

Of course, the example of a package included by another codebase may seem trivial: we have obvious entry points in the form of functions and classes that belong to the documented API. Applications, on the other hand, are more nuanced. There are two types of applications, each of which require different approaches: those intended for human input, and those intended for machine input.

Machine Applications

Applications intended for computer/automated input, or “machine applications”, often require input that is difficult or even impossible to understand, and produce similarly inscrutable output.

However, they tend to be straightforward to test. Like packages, we can target their provided contracts, at the highest reasonable level.

Say we expose an API as JSON over HTTP. Some test frameworks allow us to start the application process, but hook it into internally faked TCP sockets, over which we can send our preconstructed data and evaluate the results. We can gain a lot of confidence and coverage by using this approach. If it is not available, or too cumbersome to implement, we should test the entry point functions we are exposing through the protocol.

For example, most frameworks let us expose endpoints by binding them to entry point functions through routes:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

In this case, the entry point functions are where we should test:

```
import app

class Test(unittest.TestCase):
    def test_hello_world(self):
        response = app.hello_world()
        self.assertEqual(response, "<p>Hello, World!</p>")
```

We are assuming that `Flask` is functioning properly, and binding our route to the correct location. This might be a poor assumption, as frameworks do not always behave as we expect. As stated at the beginning of the chapter, unit tests by themselves are insufficient. Functional tests and scenario tests are necessary to catch issues that escape unit tests.

Even so, this will allow us to validate the contracts provided by our own codebase.

Human Applications

Human applications are, as the name suggests, intended to process input from humans. Often, they take the form of Graphical User Interfaces (GUIs). For these applications, validation often must happen through frameworks that help with emulating human-like interaction: pressing buttons, clicking the mouse, tapping on the screen, and the like.

Unit testing in these contexts will be highly specific to the framework being used. Consider the state of the application as the user interacts with it:

- What should be visible?
- What values should be updated?

- When can an element be interacted with?
- When are the inputs valid, and how is their invalidity reported?
- Are these interactions meant to be delayed? What amount of delay is acceptable?
- How responsive is this application?

> Let's Get Technical
>
> Command Line Interfaces (CLIs) cross the boundary between machine and human applications, as they are often used by both. They can be tested similarly to machine APIs: by finding the highest entry points, and then invoking those in the unit tests and validating the output.

When creating these tests, think about them from the perspective of the user. Ask yourself what sort of experience are they having when they use your application, and how can you ensure through the tests that it functions as they expect.

Conclusion

A project with a comprehensive, stable, and correctly-written suite of unit tests is the most relaxing environment in which to work. Every change can be quickly validated, and its impacts seen. Development for such a project is rapid, and can be picked up by even the inexperienced, as they have tests to guide them.

A project with inaccurate, buggy, incorrectly-applied tests is terrible. It is a soul-sucking experience to work in such an environment, as every change results in some unknown test breaking for some unrelated reason, in a way that is never impactful. In those cases, remove the bad tests, write better tests, and move on. Do not let bad tests prevent you from striving for the first scenario.

---

10. Refactoring

> In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts.
>
> *Martin Fowler, Refactoring: Improving the Design of Existing Code*

Rule: Refactor Continuously

Use refactoring as a tool to learn about a codebase, and to slowly improve that codebase. Do not be afraid to begin refactoring, and do not be afraid to abandon a refactor.

Refactoring is the process of improving code without altering behavior. For instance, refactoring a function would not alter its inputs, outputs, or side effects, but would alter the code internal to the function. On the other hand, refactoring a feature would not alter the behavior of a feature, but may significantly alter various functions, including adding new ones and removing unnecessary ones.

Refactoring can be used to either improve code that has been published and is active, or to improve code during development before it goes into production.

Refactoring code during development is important. The development process starts with the design phase. This involves using research and experimentation to explore the problem space, and come up with a potential solution. Time spent during the design phase is subject to diminishing returns; if everything is completely correct with the initial software implementation, too much time was spent on design.

Instead, it is more efficient to get the initial design correct with regards to major decisions, and then discover additional minor problems during the first implementation. It is important not to stop after the first implementation; instead, as problems are identified, the code should be iteratively refactored and improved, until both the code and the design are robust. This decreases the time to develop an application or feature, and the software produced tends to be more reliable.

Once code has been published, requirements will change, and the quality of the codebase may drift. To help prevent that drift and maintain low MTC, the code will periodically need to be refactored. We have discussed being “wrong in correctable ways” in previous chapters, and if the code was written according to the rules in this book, the refactoring process should be relatively painless. However, it is important to recognize we may not have the luxury of code that is

simple to understand and easy to change. Often, we are working with software written many years ago, and it feels like it has been cobbled together with little regard for the sanity of the maintainers. In such cases, we have received a message in a bottle, but it has degraded during its journey.

This chapter will cover how to apply the rules in this book to code that does not adhere to those rules. It will go over the proper process to reduce the chance of introducing bugs, what problems to look for, and how to recognize if a refactoring attempt should be abandoned. Refactoring is a deep topic, and this chapter is only meant as an introduction to it. I encourage the reader to explore refactoring in greater detail both by reading books dedicated to it and through practice.

Before You Begin

There are two common reasons programmers do not refactor code, even when small improvements could have large, positive impacts: time constraints, and fear. Time constraints require frank discussions with management about the relative value add and tradeoffs involved. It is not uncommon to find yourself in a situation where the organization values feature development over foundational improvements. Navigating those circumstances is challenging. I advise approaching all such interactions with honesty, humility, and copious data.

Fear is different. It tends to arise because the process of changing code involves breaking it, and breaking code results in compiler errors, failing tests, and potentially introducing bugs. A healthy dose of fear is not a bad thing; as long as it does not paralyze us, it can often result in making fewer mistakes as we approach improving code.

When refactoring, it is key to adhere to the following guidelines:

- Use version control.
- Edit code in a development environment, never change code directly in production.
- Use unit tests to ensure existing behavior remains consistent.
- Avoid automation that transfers code from development to production without human review.
- Copy production data (or generate new data) for use in your development environment.
- Ensure all external dependencies are configured for development. That is, connect to a development database, use external development APIs, etc. While this is not always necessary, err on the side of caution, and be mindful of using any production system during development.

If you have a fear of refactoring, you are not alone. The best way to break that fear is to break the code, demonstrating the worst that can happen is not so bad (assuming that you follow the above practices). Modern tooling allows us to easily revert changes, so there is no permanent damage if it does not work out. Additionally, keep refactors targeted and focus on small, incremental changes to achieve an overall improvement. This way, if a misstep is taken, the impact is minimal.

Know the Tools First

Every language is different. They come with different package management systems, different formatting conventions, different standard libraries. Within the definition of Mean Time to Comprehension (MTC) provided in the book's Introduction, there is an qualification: "a programmer familiar with the given … *libraries, tooling*." - i.e., we assume that whoever is reading our code is at least familiar with all of the essentials around that code. That familiarity is critical: without it, you cannot expect to understand the code you are working on.

> Stepping into a refactor is a wonderful way to learn about the particularities of *that specific project*. It is not a good way to learn a language or the tooling of that language.

Before refactoring, make sure you understand the language, including its standard library and all the major libraries or frameworks the project depends on. Read about them. While going through the process of refactoring, if you do not understand some facet of the language, stop and look it up. Perhaps implement a tiny program to familiarize yourself with how that element is used.

Ensuring you know the tools can forestall awkward refactor attempts that are the result of unfamiliarity with software using that particular language and that particular framework.

Identifying Problems

The actual process of refactoring code will depend on the language and structure, but there are some common approaches that can help. The first step in refactoring is identifying problems with the current code. The rules covered in this book provide guidance to do this:

- Nested control blocks, such as `for` loops, `if` statements, `switch` statements, `try|catch` blocks, etc. (chapter 2, “Recipe”)
- Nested function calls, like: `A(B(C(D(foo))))` (chapter 2, “Recipe”)
- Unnecessary variable mutation (chapter 3, “State”)
- Global variables (chapter 3, “State”)
- Mixing building logic with business logic (chapter 4, “New”)
- Conditional logic that could be modeled using the type system (chapter 6, “If”)
- Shorthand or meaningless variables names: `a` , `m1` , `myVar` , etc (chapter 7, “Naming”)
- Variables that seem to be re-used for multiple purposes (chapter 7, “Naming)
- Inconsistent style or naming (chapter 7, “Naming)
- Excessive in-line comments, explaining what a line of code does, instead of why the line is necessary (chapter 8, “Documentation”)
- Lack of documentation about expected inputs and outputs (chapter 8, “Documentation”)

Simple Improvements

Let us examine the following function, which calculates the standard deviation of an array of numbers:

```
# We're going to avoid most built-in math functions,
# to make the code more interesting
import math
```

```
def StdDev(array):
    Sum = 0
    Total = 0
    n = len(array)
    for i in range(len(array)):
        n = array[i]
        Sum = Sum + n
    if n < 2:
        raise ValueError("Must have at least 2 numbers in array")
    mean = Sum / n
    for i in range(len(array)):
        Sum += (mean - array[i]) * (mean - array[i])
    Total = Sum / n
    Total = math.sqrt(Total)
    return Total
```

Most programmers will have the instinct that something is amiss with this code. Using the guidance of the rules outlined above, we can concretely identify the issues. I encourage you to take a moment and identify any problems you see.

Recall it is important to create unit tests to ensure behavior remains consistent across refactors (see chapter 9, “Unit Testing”).

This simple unit test helps us ensure consistent behavior:

```
import unittest
from . import math_utils

class TestModule(unittest.TestCase):

    def test_stddev_invalid(self):
        self.assertRaises(ValueError, math_utils.StdDev, [])
        self.assertRaises(ValueError, math_utils.StdDev, [0])

    def test_stddev(self):
        self.assertEqual(math_utils.StdDev([1, 2]), 0.5)
        self.assertEqual(math_utils.StdDev([0,1,2,3,4,5,6,7,8,9]), 2.872281323269)
        self.assertEqual(math_utils.StdDev([1, 4, 5, 7]), 2.165063509461097)
        self.assertEqual(math_utils.StdDev([344, 1002, 45,222]), 361.66515936706)
```

We call the function we want to verify, and check the results against our expectations. If the behavior of the function changes, the test will alert us to this fact by failing. There is always some risk here: in a system with enough complexity, ensuring all behavior remains consistent can be quite difficult. This is why refactoring should be done in as many small, incremental doses as possible, since doing so allows us to more easily validate that the changes did not impact behavior.

Let us take a look at a rewritten version. Try to identify some of the changes we made, and whether they were the ones you expected:

```
# We're going to avoid most built-in math functions,
# to make the code more interesting
import math

def stddev(numbers):
    """
    Given a sequence of numbers, calculates their standard deviation.

    Args:
        numbers (Sequence[Union[int, float]]): A sequence of numbers.

    Returns:
        float: The standard deviation of the sequence.
    """
    if len(numbers) < 2:
        raise ValueError("Must have at least 2 numbers in array")

    num_sum = 0
    for number in numbers:
        num_sum += number

    mean = num_sum / len(numbers)

    deviation_sum = 0
    for number in numbers:
        deviation = mean - number
        deviation_sum += deviation * deviation

    variance = deviation_sum / len(numbers)
    return math.sqrt(variance)

StdDev = stddev
```

In the above example, we made several improvements to reduce MTC:

- There were intermixed variable naming conventions. We updated the code to adhere to PEP8, the formatting standard for the Python language. This is not by itself enough to make code readable, but consistency is a step in the right direction (see chapter 7, “Naming”). Because we are altering the behavior of all code in this function, the style changes do not have to be done separately.
- We changed the variable names to be more meaningful. Instead of using generic names, we use the mathematical terms for the individual elements used in the calculation (see chapter 7, “Naming”).
- The original code was not taking advantage of Python’s for loop to easily to iterate over a list of items, instead accessing using the index. We could have used reduce or some other construct, but in Python a simple for loop is faster for most programmers to understand (Know The Tools First).
- We eliminated some unnecessary variables and statements, but also added another variable (deviation) to help clarify things and improve flow (see chapter 2, “Recipe”).

- We moved the value check to the beginning of the function, for clarity and to prevent unnecessary execution (see chapter 2, “Recipe”).
- We added a doc block to clarify the function’s purpose and usage (see chapter 8, “Documentation”).

Note that we aliased the old function name at the bottom of the example, despite its non-compliance with PEP8 coding standards. This is because removing it entirely would break the existing API of the function, and would potentially propagate changes to the rest of the codebase, or even other software systems if this function is a public part of a package meant for other applications. In general, when we want to change a function name, we must evaluate if we should alias the name to keep backwards compatibility (if the function is used outside this codebase, this would be the correct approach) or simply update the name and all the references to it (if the function is entirely internal, we could do this).

Law of Demeter

The Law of Demeter (LoD), also known as the Principle of Least Knowledge, states that code should only access its immediate dependencies, and should avoid reaching into those dependencies to obtain access to additional references. If those references are needed, they should be added as dependencies in the form of arguments or fields. Often the process of refactoring to follow the LoD eliminates unnecessary code, sometimes even entire functions or classes, which only existed in the first place to cover up the fact that the LoD was being broken. Following LoD improves the clarity of the code, and as a result, reduces MTC.

Here is an example of breaking the LoD:

```
def build(obj):
    obj_id = obj.value.id_generator.generate_id()

    return BuiltObject(obj_id, obj.defaults, obj.prop.overrides)
```

This violation of the LoD causes several problems:

- If the internal structure of the passed `obj` changes, this code will break, indicating it is fragile. Coupling has been accidentally introduced, and the maintainer of the underlying `id_generator` class (or the underlying `value` or `props` class) may have no idea that the `build` function relies on them.
- Interacting with this function is onerous, as an entire `obj` must be created somehow in order to call `build`.
- This function lacks clarity, because the real requirements are hidden within the function’s code, where properties from `obj` are accessed.

Rather than reaching deeply into the `obj` variable, `build` should request what it is using directly, as arguments. We therefore refactor the code as such:

```
def build(id_generator, defaults, overrides):
    return BuiltObj(id_generator.generate_id(), defaults, overrides)
```

We can see that `obj` as a dependency was eliminated entirely here. Instead, `build` only requires the exact values it needs in order to accomplish its task.

Following The Specification

A common difficulty when refactoring is determining whether or not the existing code is actually correct. This is because correctness of code cannot be evaluated simply by looking at its behavior in isolation. We must know the specification for the code, which dictates what “correct” means. Specifications are often poorly documented, or perhaps not documented at all. It is our job as programmers to research and understand the real requirements.

The following program builds a dictionary of keys and related values based on an input string containing those values.

Here is the string value: `time=1998-10-10,distance=5554,color=blue`

```
import re

def parse_values(values):
    result = {}
    result["distance"] = re.search(r"distance=([\w\-]+)", values).group(0)
    result["time"] = re.search(r"time=([\w\-]+)", values).group(0)
    result["color"] = re.search(r"color=([\w\-]+)", values).group(0)
    return result
```

One positive of this example is that the formatting already adheres to PEP8. However, there are still some problems with it.

A few things jump out immediately:

- Every time this function is called, we recompile the same regular expressions.
- The pattern of searching for a regex seems to be repeated.
- What happens if this fails to find a value? Currently, it will fail with some esoteric error message. Let us assume the spec says the value should not be set in the returned dictionary.

We can rewrite it to improve these things (for this example, we will assume there are unit tests in place):

```
import re

value_re = {
    "time": re.compile(r"time=([\w\-]+)"),
    "distance": re.compile(r"distance=([\w\-]+)"),
    "color": re.compile(r"color=([\w\-]+)"),
}

def parse_values(values: str):
    result = {}
    for k, r in value_re.items():
        v = r.search(values).group(0)
        if v:
            result[k] = v

    return result
```

This is better. By using a map (see chapter 6, “If”), we can easily add more values, we handle errors properly, we are not recompiling on every function call.

We may look at the code now and believe it to be correct: it behaves as before, is more performant, and has less duplication. However, recall that correctness is determined by the specification. To know with certainty that we have refactored the code correctly, we must refer to that specification.

> Let's Get Technical
>
> MTC likely decreased with this change. It is often the case that for trivial examples, generalizing and robustifying them initially makes them slightly more difficult to comprehend. However, this changes as trivial cases become more complex. Generalization ensures the code remains robust and the MTC remains constant.

In the case of the above example, let us imagine the specification for this data format is an arbitrary number of comma-delimited properties, where the keys and values are separated by an `=` sign. Perhaps when the code was originally written, the only properties were time, distance, and color, but that was coincidental. That coincidence, however, caused the original programmer to believe the code was correct, as it parsed all the available data.

Because this code does not adhere to the actual specification, the original behavioral model was wrong. This is a normal part of programming, and when we encounter these situations, we should be aware that the code likely provided value for some time, and that is important. However, when refactoring, we should make sure we know the actual specification, and refactor code to reflect that knowledge.

Start by updating the unit tests (refer to Contract Testing in chapter 9, “Unit Testing”).

```
import unittest
# This is the module with the parse_values function
from . import value_parsing

class TestModule(unittest.TestCase):

    def test_parse_values(self):
        self.assertDictEqual(
            "time=1998-10-10,distance=5554,color=blue",
            {"time": "1998-10-10", "distance": "5554", "color": "blue"}
        )
        # Here we add a unit test to reflect
        # our updated understanding of the spec
        self.assertDictEqual(
            "key1=value 1,key 2=value 2,KEY3=VALUE3",
            {"key1": "value 1", "key 2": "value 2", "KEY3": "VALUE3"}
        )
```

Then we can refactor the code to reflect our understanding of the spec, and pass the updated unit test.

```
def parse_values(values):
    """
    Given a string of key=value pairs, parse into a dictionary.

    Args:
        values (str): A string of key value pairs, in the form
            "<key1>=<value1>,<key2>=<value2>,etc"

    Returns:
        dict: A dictionary with the parsed keys and values.
    """

    result = {}
    kv_pairs = values.split(',')
    for pair in kv_pairs:
        k, v = pair.split('=', 1)
        result[k] = v
    return result
```

Or, even simpler and taking advantage of the available Python syntax:

```
# The downside of this approach is that, while it
# does the same thing as the previous example,
# it can be confusing to those less familiar with python
def parse_values(values):
    # (The doc block is left out of this example for space reasons)
    return { pair.split('=', 1) for pair in values.split(',') }
```

No regex, new properties do not need to be handled individually, and the code is concise. It appears to properly adhere to the specification.

There is a downside: if any of the keys or values use the reserved tokens `,` or `=` with escape characters introduced, a full tokenizer and parser will need to be used (note that most languages have some lexing functionality built in). Refactoring to address this issue is left as an exercise for the reader.

Abandoning a Refactor

> There exists in such a case a certain institution or law; let us say, for the sake of simplicity, a fence or gate erected across a road. The more modern type of reformer goes gaily up to it and says, “I don’t see the use of this; let us clear it away.” To which the more intelligent type of reformer will do well to answer: “If you don’t see the use of it, I certainly won’t let you clear it away. Go away and think. Then, when you can come back and tell me that you do see the use of it, I may allow you to destroy it.”
>
> *G. K. Chesterton, Chesterton's Fence*

Refactoring code is a wonderful way to familiarize ourselves with a codebase. Sometimes, we may start refactoring a project only to realize that it was implemented a certain way for a reason, and changing it is more complicated than we initially thought.

When this happens, remember that the time spent was not a waste. We gained a deeper understanding of both the code and the problems that it solves.

The knowledge gained from the experience is valuable, and will help us navigate the project moving forward. At some point in the future, we can return and improve the codebase in other ways that still respect the project's requirements and constraints.

Do not view abandoning a refactor as a failure. Doing so can introduce a fear of failure that will make it more difficult to begin refactoring again in the future, and make it more difficult to stop refactoring when it becomes clear the code will be worse due to the changes. The knowledge gained from the process is always a success.

Ask yourself the questions: what documentation could have been present in the code that would have allowed me to understand why it is written the way it is? How can I communicate to someone in the future what I have learned during this process so that future refactor attempts are more productive?

Record the answers to those questions within the codebase, as either doc blocks or comments, and outside of the codebase in supporting documentation. This will ensure the hard-won knowledge you gained is shared with future maintainers and will make their job that much easier.

Conclusion

These examples really just scratch the surface. In large, complex projects, the process of unraveling a codebase may take weeks, months, or even years. As much as possible, the refactoring process should be approached in incremental steps, where each step brings the code to a more cohesive, simpler state.

The ability to refactor may be the most vital skill a programmer can have. It improves the ability to write new code, it improves existing code, it reduces the time others have to spend on understanding or working within a codebase, it decreases bugs, it deepens understanding of the codebase as a whole, and it frequently results in improvements to the end user experience.

---

11. Conclusion

> In most professional domains, ... it takes a relatively long time for students to acquire the relevant knowledge and skills required for the profession.
>
> *K Anders Ericcson, Deliberate Practice*

New violin students often develop bad technique that limits their potential progress; this effect is exacerbated if they lack formal training. Perhaps they can learn some of the simpler pieces, but as soon as they need to play very quickly their bow orientation or positioning makes it impossible. Perhaps they can play some slower pieces, but holding the violin improperly leaves them unable to use vibrato to create a rich and vibrant tone.

This is not exclusive to violin students, or even musicians. Anyone who has practiced a sport knows the same lesson: bad habits put a limit on performance.

Golf, an individual sport with a single goal and one metric, is one of the best examples of this. Every golfer knows that beginners can get lucky, and anyone with enough persistence will eventually put the ball in the hole. However, eventually putting the ball into the hole does not make one good at golf. Rather, practice and technique reduce the number of strokes it takes to accomplish the goal, and perhaps more importantly, help forestall injury while playing the game.

Improving technique involves undoing bad habits, and that process is difficult, tedious, and requires dedication. It requires the learner to commit to the endeavor and to think critically about not only their goal, but more importantly *how* they accomplish the goal. This often involves “relearning” the activity to undo bad habits. During the process of relearning, the learner will temporarily become worse at the activity.

The violinist training their bow technique will find they must play much slower. The golfer focused on their swing and stance will find the ball goes to unexpected places, at first. And the programmer learning to apply the rules in this book will initially take longer to write the same piece of code.

It is vital to recognize this is a transition phase. Output will decrease while new technique is learned, and coding may feel more difficult or cumbersome. But it is temporary; as technique improves, so do instincts, and eventually problems that once may have taken days or even weeks become trivial. Of course, it is not a panacea: there will always be difficult problems that plague programmers— no amount of technique can make Paganini’s 24th Caprice easy, and certainly there are at least as difficult problems that face us in software. However, eliminating problems of our own making allows us to focus on the challenge inherent in the specification.

Deliberate Practice

In 2004, Dr. Ericcson published a paper titled “Deliberate Practice and the Acquisition and Maintenance of Expert Performance in Medicine and Related Domains”. Specificity was clearly a virtue he admired.

In it, he covered the body of evidence that shows that continued, deliberate practice is the necessary ingredient for becoming an expert in a given field. Most people who learn a skill will stop as soon as they are merely proficient; they are able to accomplish the goal, so there is no incentive to get better. However, if one wishes to truly master the skill, they must have the drive and desire to continuously improve. The evidence suggests this drive is more important than any innate talent. The process of improvement is often a slow one. It requires “execution, monitoring, planning, and analyses of performance”.

The paper covers many disciplines, including that of computer science. In all cases, unless there is a drive to become better, not only will practitioners quickly hit the limit of “proficient amateur,” they will in fact become worse at the skill over time.

I have worked as a programmer for almost two decades. To my chagrin, even after I learned the rules outlined in this book, I continued to break them with scant justification. As a result, I have written bugs, inscrutable code, and overly complex logic. I have made working in projects more difficult not only for myself, but other programmers who must maintain the bad code I wrote.

The unfortunate truth is that we are constantly fighting problems of our own creation.

Software is similar to a building undergoing constant maintenance and additions, and yet much of it is treated as though the initial plan is all that will ever be needed. What happens when another story is requested? What about another three stories? Can the foundation support the load? Can additional structural support be added, or was the original construction such that it precludes any addition?

Following the rules we’ve covered in this book does not guarantee that the codebase will be correct for all possible future modifications, without changing existing elements. Instead, it provides a path to ensure that most of the time, most elements will not need to be modified, and if they must be, the process to do so is relatively painless. Critically, it creates software focused on understandability.

Of course, understanding how to write code properly is only one part of being a capable programmer. There is much more to learn about designing good software: from high-level system architecture to low-level project layout, from user interface design to language interface design; it all matters. Writing code that will stand the test of time, while critical, is only the beginning. As always, the most important element is to nurture the desire to learn and improve.

> Whereas proficiency in everyday skill is attained rapidly, professional development ... is completed only after years or even decades of experience.
>
> *K Anders Ericcson, Deliberate Practice*

And this is a wonderful thing, because surely we would quickly become bored otherwise.

The Road to Mastery

> Unless he is certain of doing as well [as the masters], he will probably do best to follow the rules.
>
> *Strunk and White, The Elements of Style*

Over the course of this book, we have covered ten rules that we can carry with us into future projects:

1. Make your code read like a recipe.
2. Avoid state mutations, and create variables close to where they are used.
3. Avoid constructing new objects outside of bootstrapping and factories.
4. Use polymorphism to make the program dynamic.
5. When possible, use maps, registries, and polymorphism rather than conditionals.
6. Names should reflect semantic mappings.
7. Document why something is done, not what is being done.
8. Write unit tests focused on contracts and complexity, not merely coverage.
9. Learn and improve through refactoring.
10. Practice.

The rules outlined in this book are foundational. They are the fundamental code construction skills necessary to build reliable, understandable, extensible software.

Eventually, following the rules will become instinctual. That is an important step, because it leaves you free to focus on the inherent complexity of problems, without introducing accidental complexity as you compose the solution.

---

12. Cheat Sheet

Write code like it is a recipe (because it is)

Start by constructing the long-lived service objects, and wiring them all together. Primary execution happens after that. Functions behave like paragraphs: the longer they run on, the easier it is to become lost.

Remember to separate “new”

Often, going through the exercise of keeping construction logic and business logic apart seems unnecessary or a waste of time, or perhaps seems to add complexity. However, as the codebase grows, somehow it becomes more and more difficult to change, more and more difficult to understand, as a result of coupled construction and business logic.

Most “if” statements (conditional logic) can be replaced with polymorphism

Every if statement creates a branch in the execution path that the code could take, and each branch and its implications must be fully understood. Branching logic has a high cognitive load: it is difficult to reason about multiple things at the same time.

Documentation is vital

- Document the intent behind a decision (i.e. the “why”), not what the code does
- Prefer doc blocks to inline comments
- Ensure documentation is updated to remain correct when updating code

Unit tests must be useful

- Contract tests and complexity tests should be permanent, other unit tests should be temporary
- There are four types of test doubles: dummies, stubs, mocks, and fakes
- Adhering to the rules described in this book should make unit testing straightforward

Common problems when refactoring

- Nested control blocks, such as for loops, if statements, switch statements, try|catch blocks, etc.
- Nested function calls, like: A(B(C(D(foo))))
- Shorthand or meaningless variables names: a, m1 , myVar, etc
- Unnecessary variable mutation (changing the value of a variable)
- Global variables
- Excessive in-line comments, explaining what a line of code does instead of why it does it
- Lack of documentation about expected inputs and outputs
- Variables that seem to be re-used for multiple purposes Inconsistent style or naming

Acknowledgements

Nothing is created in a vacuum, and this book is certainly no exception. Without the support and encouragement of those around me, I doubt I would have ever managed to finish writing it.

Thank you to my editor, Ian Hough, for his help in taking this from a rough manuscript to a real book.

I received feedback and suggestions throughout this process, and those contributions were essential in making this book a reality. The following individuals all helped with drafts at various stages, and I am indebted to them for their contributions: my wife Katie, Nathan Cutler, Raymond Krajci, Michael Sweeney, and David Van Maren. I appreciate all the time and effort you spent helping me.

I have numerous indirect inspirations, whose work I used as a basis for everything in this book. This includes: Miško Hevery, Rich Hickey, Dave Thomas, Micheal Feathers, and many others.

Finally, I am grateful to you, the reader, for choosing to spend your time with The Elements of Code.

Thank you.