7

Clean Functions

In the earliest days of computing, programmers composed their systems of a sequence of routines that performed individual tasks. They quickly discovered that some of those routines were generally useful, and they wrote them in their notebooks and called them *subroutines*. When needed, they would transcribe those subroutines from their notebooks directly into the program they were creating. There were no calls—those subroutines were simply encoded inline within the program.

Later, in the early ’50s, instructions were added to the machines that would allow subroutines to be called. At first, programmers were reluctant to use such call instructions because they were expensive in time. But as machines grew in capability, the call instructions became less expensive, and programmers began to use them more and more frequently.

Then, in the era of FORTRAN and PL/1, we started composing our systems primarily of subroutines, and we introduced the special case of *functions* that take arguments and return values. Nowadays, only the function survives. Functions are the first line of organization in any modern program. Writing them well is the topic of this chapter.

Small!

Over the last six decades, I have written functions of all different sizes. In my early years, I wrote several nasty 3,000-line abominations. Later I wrote scads of functions in the 100-to-300-line range. As I matured, I wrote functions that were 20 to 30 lines long. And in the last two decades, I've shortened that to a dozen lines or less. What those decades have taught me, through long trial and error, is that functions should be very small.

Of course I am not the first person to say this. The idea of keeping functions small is an old one. In the eighties, it was common to hear that a function should be no bigger than a screenful. Of course, this was said at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. This effectively put the limit at around twenty lines.

In 1999, I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together so that he could show me the TDD[1] discipline. I was a skeptic about TDD and wanted to see it firsthand.

---

1. Back in those days, it was called *test-first programming*.

One failing test at a time, we cobbled together a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie *Cinderella*. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. As we wrote the code together, making one test pass after another, I was struck by how small the functions were. I was accustomed to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each had a descriptive name. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. It was that experience that drove home to me the importance of keeping my functions very small.

Well-Written Prose

Keeping functions very small implies that the blocks within `if` statements, `else` statements, `while` statements, and so on will be only one or two lines long. Often that block will contain a call to another function. Not only does this keep the enclosing function small, but it also adds documentary value because the functions called within the block will have nicely descriptive names. Such `if` / `else` / `while` statements read nicely because the function names flow well with the keyword. They will read like well-written prose.

```
if(employee.shouldBePaidToday())
  employee.pay();
```

This also implies that small functions will not be large enough to hold deeply nested structures. The indent level of a function will seldom be greater than one or two. This, of course, makes the functions easier to read and understand.

This is not a hard-and-fast rule, and it can be overdone. There are certain functions that read better if they are not decomposed into smaller four-line functions. But these are exceptional cases. In general, keeping functions small is the best strategy.

One Level of Abstraction per Function

In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our functions are all at the same level of abstraction. Look at the following bit of Golang code.[2]

2. This is from the Video Store example that we'll be studying much more thoroughly in Chapter 10, "One Thing."

```go
func (customer *Customer)
statement() string {
  totalAmount := 0.0
  frequentRenterPoints := 0
  result := "Rental Record for " + customer.name + "\
  for _, rental := range customer.rentals {
    thisAmount := 0.0
    switch rental.movie.movieType {
    case NewRelease:
      thisAmount += float64(rental.daysRented * 3)
    case Regular:
      thisAmount += 2
      if rental.daysRented > 2 {
        thisAmount += float64(rental.daysRented-2) *
      }
    case Childrens:
      thisAmount += 1.5
      if rental.daysRented > 3 {
        thisAmount += float64(rental.daysRented-3) *
      }
    }
    frequentRenterPoints++
    if rental.movie.movieType == NewRelease && rental
      frequentRenterPoints++
    }
    result += fmt.Sprintf("\t%s\t%.1f\n", rental.movi
    totalAmount += thisAmount
  }
  return result + fmt.Sprintf("Amount owed is %.1f\n"
                              "You earned %d frequent
                              totalAmount, frequentRe
}
```

It is easy to see how this violates the rule. There are concepts in there that are at a very high level of abstraction, such as looping over `customer.rentals`. There are other concepts that are at an intermediate level of abstraction, such as the `switch` statement that selects the movie type. And there are still other concepts that are remarkably low level, such as `frequentRenterPoints++` and the code that calculates prices for each movie type.

Mixing levels of abstraction within a function is usually confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse, like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

Reading Code from Top to Bottom: The Stepdown Rule

We want the code to read like a top-down narrative.[3] We want every function to be followed by those at the next level of abstraction so that we can read the program from top to bottom, descending one level of abstraction at a time. I call this *The Stepdown Rule*. We can see this in the following refactoring of the previous example.

3. [KP78], p. 37.

```
func (statement *RentalStatement)
makeStatement() string {
  statement.clearTotals()
  return statement.makeHeader() +
         statement.makeDetails() +
         statement.makeFooter()
}

func (statement *RentalStatement)
clearTotals() {
  statement.totalAmount = 0.0
  statement.frequentRenterPoints = 0
}

func (statement *RentalStatement)
makeHeader() string {
  return "Rental Record for " + statement.name + "\n"
}

func (statement *RentalStatement)
makeDetails() string {
  rentalDetails := ""
  for _, rental := range statement.rentals {
    rentalDetails += statement.makeDetail(rental)
  }
  return rentalDetails
}

func (statement *RentalStatement)
makeDetail(rental *Rental) string {
  amount := rental.determineAmount()
  statement.frequentRenterPoints += rental.determineF
  statement.totalAmount += amount
  return statement.formatDetail(rental, amount)
}

func (statement *RentalStatement)
formatDetail(rental *Rental, amount float64) string {
  return fmt.Sprintf("\t%s\t%.1f\n", rental.getTitle(
}

func (statement *RentalStatement)
makeFooter() string {
  return fmt.Sprintf("Amount owed is %.1f\n"+
    "You earned %d frequent renter points",
    statement.totalAmount, statement.frequentRenterPc
}
```

To read this we start at the top with the `makeStatement` function. To make a statement we first clear the totals, then make the header, the details, and the footer, and concatenate them all together. All those statements are at the same level of abstraction, one level below the name of the function.

To clear the totals we set `totalAmount` and `frequentRenterPoints` to zero. Again, each line is at the same level of abstraction, one level below the name of the function.

To make the header we format the name from the rental statement with header text. The single line in that function is one level of abstraction below the name of the function.

In the ’70s and ’80s, we called this *functional decomposition*. It was an important part of the *structured programming* discipline. High-level functions are decomposed into lower-level functions, creating a hierarchy of function calls.

When you follow this discipline, your functions will generally be very small because each function contains just one level of abstraction and defers to the next level down through another nicely named function call.

The discipline of refactoring is a critical element of this approach. When we initially write the code for a function, we tend to do a depth-first search through the levels of abstraction. The code that results from that depth-first search looks like the first version above. But once we get the function working, it is very easy to refactor it by extracting out the different levels of abstraction into nicely named functions.

So, first, make it work. Then, make it right.

Entanglement

One objection to this approach, offered by John Ousterhout, is that when you decompose larger functions into smaller ones, you can sometimes create small functions that are tangled together. Two functions are entangled when, in order to understand one, you have to understand the other.

John is right about this, but he and I differ on the relative cost. I can tolerate a bit of entanglement so long as the functions step down one level of abstraction, and the lower-level functions are positioned after the higher-level functions. John is much more averse to entanglement and would rather two entangled functions be merged into a single function.

Switch Statements

It’s hard to make a small `switch` statement.[4] It’s also hard to make a `switch` statement that does one thing. By their nature, `switch` statements always do *N* things. Unfortunately, we can’t always avoid `switch` statements, but we can make sure that each `switch` statement is buried in a low-level module and is never repeated. We do this, of course, with polymorphism.

4. And, of course, I include `if` / `else` chains in this.

Consider, for example, this `switch` statement extracted from the example above.

```
func (rental Rental) determineAmount() float64 {
  amount := 0.0
  switch rental.movie.movieType {
  case NewRelease:
    amount += float64(rental.daysRented * 3)
  case Regular:
    amount += 2
    if rental.daysRented > 2 {
      amount += float64(rental.daysRented-2) * 1.5
  }
  case Childrens:
    amount += 1.5
    if rental.daysRented > 3 {
      amount += float64(rental.daysRented-3) * 1.5
    }
  }
  return amount
}
```

There are several problems with this function.

1. It's larger than I'd like, and it will grow when new movie types are added.
2. It very clearly does more than one thing.
3. It violates the Single Responsibility Principle[5] (SRP) because there is more than one reason for it to change.

5. See Chapter 19, "The SOLID Principles."

4. It violates the Open–Closed Principle[6] (OCP) because it must change whenever new types are added.

6. Ibid.

But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example, we already have `determinePoints`, and we might later have `applyCoupon` or `getAgeRestriction`. It's not hard to think of many others. All of which would contain a related `switch` statement.

---

7. [GOF95].

One good solution to this problem is to bury the `switch` statement in the basement of an Abstract Factory,[7] and never let anyone see it. The factory uses the `switch` statement to create appropriate instances of the derivatives of `RentalType`.

```
type RentalTypeFactory interface {
  make(typeName string) RentalType
}

type RentalTypeFactoryImpl struct{}

func (r RentalTypeFactoryImpl) make(typeName string)
  switch typeName {
  case "childrens":
    return ChildrensRental{}
  case "new release":
    return NewReleaseRental{}
  case "regular":
    return RegularRental{}
  }
  return nil
}
```

My general rule for `switch` statements is that they can be tolerated if they appear only once, are positioned in a concrete module like `main`, are used to create polymorphic objects, and are hidden behind an interface so that the rest of the system can't see them.

Clean Functions: A Deeper Look

A clean function should have five attributes. It should be nameable, insulated, homogenous, contextual, and pure. If you reorder that, it spells PINCH—but that's the wrong order in which to present them.

Contextual

The concept and discipline of modularity was not immediately obvious to the earliest of programmers. It slowly grew out of sheer necessity. In the late 1940s, Grace Hopper wrote down common sequences of code in her notebooks to help her program the Harvard Mark 1. She called these sequences *subroutines*. They were transcribed into programs, which were then painstakingly punched onto the long paper tapes that drove the execution of the machine.

Those subroutines were the first modules. From there, the concept of modularity grew into callable subroutines, and finally functions.

By the 1960s, programmers were adopting a discipline called *modular programming*, which stated that the best programs were decomposed into many different modules. By the late '60s, Dahl and Nygaard advanced the concept of modularity by inventing object-oriented programming.[8] In the early '70s, David Parnas continued the advancement by promoting the benefits of information hiding.

---

8. Though they did not give it that name. *Object-oriented* was coined by Alan Kay a few years later.

Nowadays, we understand that every function lives within a context. A context is a cooperative grouping of functions and data structures. Every context has an external interface and an internal implementation.

In OO languages, contexts are most commonly defined with classes. In Java, for example, the public methods of a class represent the external interface, while the private data and methods comprise the internal implementation.

In C, we use *.h* header files for the external interface and *.c* files for the internal implementation.

Some languages provide a rich syntax for defining a context. In other languages, the programmer must define a context by informal conventions such as filenames, directories, and comments.

But irrespective of the language and the syntax, every function lives within a context—and it is the programmer's job to identify, create, and manage those contexts.

Some of the functions within a context will be public, meaning that they are known and used outside of that context. They are part of the external interface of the context. Other functions within the context will be private, meaning that they are known and used only within the context and are not used outside of it.

Many OO languages provide a syntax to declare functions either public or private. Some have other categories, like `protected` or `package`, as ways to create partial restrictions. Other languages depend on the discipline of the programmers and various programmer conventions to achieve the same result.

A context creates a boundary between the things that are known external to the context and those things that remain private within the context. This boundary, specified by the programmer, describes the programmer's intent regarding the things that are higher level and likely won't change very often and those things that are lower level and likely will experience frequent change.

Nameable

There are at least two things to consider when naming a function. The name must be appropriately descriptive within its context, and it must be appropriately convenient.

Descriptive

Every clean function should have a name that describes what that function does. Since the name describes behavior, the name should be a verb, a verb phrase, or an implicit verb, like `Math.sign(x)` .

The description should be slightly more abstract than the code within the function. In other words, the name should hide the implementation.

So, as an absurd counter-example:

```
public static double addAtoBandDivideBy2(double a, dc
  return (a+b)/2.0;
}
```

That name certainly tells us what the code does; but it is much too concrete. Not only does it tell us what the function does, but it tells us how it does it. It exposes the implementation. We should abstract it just a bit.

```
public static double average(double a, double b) {
  return (a+b)/2.0;
}
```

That's nice. The name is an implicit verb (to average) that hides the implementation. That's good because any programmer who sees the name will know what to expect; but any programmer who wants to change the implementation will not have to change the name. For example, expanding the function to use a variable number of arguments would not affect the name of the function at all.

```
public static double average(double… ns ) {
  return Arrays.stream(ns).sum()/ns.length;
}
```

A one-word name, like `average` , may work for some functions, but others will need more words in order to be appropriately descriptive. For example:

```
public void addSalesReceipt(SalesReceipt salesReceipt
  salesReceipts.add(salesReceipt);
}
```

The function appears simple enough, but the name is three words long. Why? Perhaps we should look at the context.

```
public class CommissionedEmployee extends Employee {
  private List<SalesReceipt> salesReceipts = new Arra
  //…
  public void addSalesReceipt(SalesReceipt salesRecei
    salesReceipts.add(salesReceipt);
  }
}
```

This function is a method of a derived class. Derived classes are more detailed than their base classes. Thus, the `addSalesReceipt` function lives in a detailed context. Detailed contexts require more words to describe. Thus, you should expect that, in more detailed contexts, longer names will be necessary in order to be appropriately descriptive.

It is hard to overestimate the value of good names. Remember Ward Cunningham's principle: "You know you are working on clean code when each routine turns out to be pretty much what you expected." Half the battle to achieving that principle is choosing good names for small functions that do one thing.

Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is often better than a long descriptive comment. Use a naming convention like `CamelCase` , or `snake_case` , or `kebob-case` . This allows multiple words in a function's name to be easily read. Those words should describe what the function does.

Don't be afraid to spend time choosing a name. Indeed, you should try several different names and read the code with each in place. Modern IDEs make it trivial to change names, so experiment with different names until you find one that is as descriptive as you can make it.

Choosing descriptive names will clarify the design of the module in your mind and help you improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Convenient

In order to be convenient, the name of a function must be short and memorable. But short and memorable names are not always very descriptive. So there is a trade-off to make. The value of convenience depends on how frequently the function is called. A function that is called a lot should have a short and memorable name. A function that is called just once can afford a longer and much more descriptive name.

The two conflicting priorities of descriptiveness and convenience conspire to create the simple heuristic we discussed in Chapter 4.

The length of the name of a function should be inversely proportional to the size of the containing scope.

A scope is the span of code in which a name is known and used. For example, the scope of the `average` function is effectively global. Whereas the scope of `addSalesReceipt` is restricted to the users of the `CommissionedEmployee` class. Thus, the more detailed the context, the smaller the scope. The more general the context, the larger the scope.

Functions in a small scope do not require much convenience. So their names can be longer. This is good because functions in a detailed context usually require longer names to be descriptive. Functions in a large scope require more convenience. This is good because those contexts are usually more general and so do not usually require longer names.

For example, the function `File.open(String filename)` is short and memorable because it lives at the global scope and is very general. We would find it inconvenient if this function were named `openFileAndThrowExceptionIfNotFound` . By the same token, we would find it somewhat disconcerting if the `addSalesReceipt` function were named simply `add` .

So, the bigger the scope, the more general the function and the smaller the name. The smaller the scope, the more detailed the function and the longer the name.

From this, we would predict that global functions will usually have short, one-word names. Public functions within a context will have slightly longer names. Private functions called by the public functions will have even longer names. And private functions called by other private functions may have very long names. The deeper down you go in the function hierarchy, the longer the names will likely be. This is true of Java, C, Clojure, Python, Ruby, or any other language that allows long names.

Insulated

In days long past, we called the notion of insulation *low coupling* or sometimes *encapsulation*. But neither of these terms seem to me to be quite sufficient.

The word *complex* means more than one strand. One of the factors that make functions complex is the number of inputs and outputs they must deal with. Each of those inputs and outputs is a coupling, and each coupling is another strand weaving its way through that function. Therefore, to manage the complexity of our functions, we seek to limit the number of such couplings.

The more arguments a function has, the more it is coupled to its callers. A function with five arguments is likely more coupled than a function with two. Thus, we strive to limit the number of function arguments to some manageable limit. I prefer a limit of three where feasible. I'll talk more about that in the next chapter.

The fewer the arguments, the less coupled the function is and the simpler that function is to call. A function with no arguments is very easy to call. However, such functions aren't particularly useful unless they live within a context that has been prepared for them by other functions.

For example, you might have a context that represents a finite state machine. That context will contain variables that represent the current state of the machine, as well as the network of state transitions.

Let’s use the following state machine. It represents a simple subway turnstile. There are two states: `Locked` and `Unlocked`. There are two events: `Coin` and `Pass`, which represent the user dropping a coin in the slot, and passing through the gate, respectively. Finally, there are five actions: `alarmOff`, `alarmOn`, `lock`, `unlock`, and `thankyou`. The state transition table looks like this.

```
Locked   Coin Unlocked {alarmOff unlock}
Locked   Pass Locked   alarmOn
Unlocked Coin Unlocked thankyou
Unlocked Pass Locked   lock
```

The logic is very simple.

- When the machine is in the `Locked` state and it gets a `Coin` event, it will transition to the `Unlocked` state and invoke the `alarmOff` and `unlock` actions.
- When the machine is in the `Locked` state and it gets a `Pass` event, it will remain in the `Locked` state and invoke the `alarmOn` action.
- When the machine is in the `Unlocked` state and it gets a `Coin` event, it will remain in the `Unlocked` state and invoke the `thankyou`[9] action.

---

[9]. Perhaps it lights up a little “Thank You” light, or emits the words “Thank you” from a speaker.

- When the machine is in the `Unlocked` state and it gets a `Pass` event, it will transition to the `Locked` state and invoke the `lock` action.

From this, it should be obvious that when the system is running, the external software will need to send the `Coin` and `Pass` events. This can be accomplished by calling functions that take no arguments named `coin()` and `pass()`. We can capture this in an interface named `TurnstileEvents`.

```
public interface TurnstileEvents {
  void coin();
  void pass();
}
```

The five actions, which also take no arguments, can be captured in an interface named `TurnstileActions`.

```
public interface TurnstileActions {
  void lock();
  void thankyou();
  void alarmOn();
  void unlock();
  void alarmOff();
}
```

Then, the logic of the state machine can be captured in an abstract class named `TurnstileFSM`.

```
public abstract
class TurnstileFSM implements TurnstileActions, Turns
  private enum State {Locked, Unlocked}

  private enum Event {Coin, Pass}

  private State state = State.Locked;

  private void setState(State s) {
    state = s;
  }

  public void coin() {
    handleEvent(Event.Coin);
  }
  public void pass() {
    handleEvent(Event.Pass);
  }

  private void handleEvent(Event event) {
    switch (state) {
      case Locked -> {
        switch (event) {
          case Coin -> {
            setState(State.Unlocked);
            alarmOff();
            unlock();
          }
          case Pass -> {
            setState(State.Locked);
            alarmOn();
          }
        }
      }
      case Unlocked -> {
        switch (event) {
          case Coin -> {
            setState(State.Unlocked);
            thankyou();
          }
          case Pass -> {
            setState(State.Locked);
            lock();
          }
        }
      }
    }
  }
}
```

The running system that detects and sends events need know only about `TurnstileEvents`. It is completely decoupled from the logic and the actions. The logic of the state machine lives behind a very strong insulation barrier. Nothing outside of `TurnstileFSM` knows about the states or the logic.

Finally, the `TurnstileFSM` class is abstract in order to allow the action functions to be implemented in a derived class, thereby decoupling the logic of the machine from the implementations of the actions.

From this example, you can see the contexts that encapsulate the state machine logic, the events, and the actions. You can see how limiting the number of arguments in the functions that cross the boundaries between the contexts helps keep the coupling very low.

Of course, there will be many times when you will need functions that take one, two, or even three or more arguments. But once you pass more than three, you have to ask yourself why you aren’t passing in an object or data structure that encapsulates them. If you truly need that many arguments passed into a function, it’s worth considering whether the data could be separated into separate contexts, and whether the behavior of the function could be split up to live within those contexts.

Homogenous

The lines of a function should all be at the same level of abstraction, which is one level below the name. In the previous section, I called this *The Stepdown Rule*.

The operation of a function takes place at one particular level of abstraction described by the name of that function. The name of the function represents the abstraction that the lines within the function implement.

But what if the lines of a function exist at more than one level of abstraction? For example, consider the following function:

```
public static int orientation(Point a, Point b, Point
  double val = (b.x() - a.x()) * (c.y() - a.y()) -
               (b.y() - a.y()) * (c.x() - a.x());
  if (val == 0) {
    return 0;
  }
  return (val > 0) ? +1 : -1;
}
```

Given three points (a, b, c) on a plane, this function determines whether traveling from a to b to c is a turn to the right (−1), a turn to the left (1), or no turn at all (0).

The first part determines the cross product of the two vectors a⟶b and b⟶c. The sign of this product determines the answer.

The determination of the cross product is appropriately one level of abstraction below the name. But the other four lines are lower level than the first. So let’s put those four lines into a function of their own.

```
public static int orientation(Point a, Point b, Point
  double val = (b.x() - a.x()) * (c.y() - a.y()) -
               (b.y() - a.y()) * (c.x() - a.x());
  return sign(val);
}

private static int sign(double n) {
  if (n == 0) {
    return 0;
  }
  return (n > 0) ? +1 : -1;
}
```

Now both lines of the original function are at the same level of abstraction. This makes the implementation of the `orientation` function homogenous.

But we can do better, because the names are not particularly communicative, and the return values are unnamed and easy to get wrong.

```
public enum Chirality {left, right, none};

public static Chirality getChirality(Point a, Point b
  return chirality(crossProduct(a, b, c));
}

private static double crossProduct(Point a, Point b,
  double abx = b.x() - a.x();
  double aby = b.y() - a.y();
  double acx = c.x() - a.x();
  double acy = c.y() - a.y();
  return abx * acy - aby * acx;
}

private static Chirality chirality(double n) {
  if (n == 0) {
    return none;
  }
  return (n > 0) ? left : right;
}
```

You might be bothered that this is twice as long as the original; but it seems to me to be much more readable—especially for someone seeing it for the first time. You might also be bothered that this takes a few more CPU cycles than the original. If you have some desperate need to save 50ns, then by all means trim those cycles back. Just keep in mind that someone else is going to have to maintain this after you, and their time has value—possibly a lot more than 50ns will cost you.

Pure

The behavior of a pure function is dependent upon nothing other than its arguments; and its execution does not cause the behavior of any other function to change. Following are some of the advantages of pure functions.

- A pure function is a true mapping of arguments to return values—and can be replaced with such a mapping in order to trade execution speed for memory.
- Pure functions are much less susceptible[10] to race conditions and problems of concurrent update because they do not change the state of the underlying system.

---

10. They are not perfectly immune, because any stateful system can be simulated with pure functions. See [Functional Design].

- Pure functions are easy to combine into networks of functionality. This is because they have no external dependencies and so are free to be called at any time by any other function.
- Pure functions are easy to test. This is because the test setup does not require the coercion of the underlying system into any special state and the test does not need to assert that the underlying state of the system has been changed.
- Pure functions are easy to distribute between multiple processors. Since the output of a pure function depends only on its arguments, it doesn’t matter where, or in which processor, the function executes.

OK, so they’re great. How do you create a pure function? Simple. Don’t change the value of any variables; or to paraphrase the famous line from

Mommie Dearest: “No Assignment Statements Ever!” Or to say that in yet another way: Pure functions are immutable.[11]

11. [Functional Design], Chapter 1.

But how realistic is a rule like that? That depends upon your point of view. For example:

```
public static double sigma(double… ns) {
  var mu = mean(ns);
  var deviations =
    Arrays.stream(ns).map(x->(x-mu)*(x-mu)).boxed().m
  double variance = deviations.sum() / ns.length;
  return Math.sqrt(variance);
}
```

The `sigma` function is pure. The assignment statements within it are initializations, not assignments. They do not alter the state of any existing variable.

But what if we turn them into true assignments by splitting the initializations into declarations and assignments?

```
public static double sigma(double… ns) {
  double mu;
  mu = mean(ns);
  DoubleStream deviations;
  deviations =
    Arrays.stream(ns).map(x->(x-mu)*(x-mu)).boxed().m
  double variance;
  variance = deviations.sum() / ns.length;
  return Math.sqrt(variance);
}
```

This version of `sigma` is still pure because even though the values of the variables change, those changes are not seen outside of the function. How far can we push this? Let’s use a much more traditional[12] implementation:

12. Of all the `sigma` implementations shown in this chapter, this is my favorite. No goofy syntax; no strange concatenation of dots, parentheses, and arrows; no annoying conversion functions. Just nice, straightforward code.

```
public static double sigma(double… ns) {
  double mu = mean(ns);
  double variance = 0;
  for (double n : ns) {
    var deviation = n - mu;
    variance += deviation * deviation;
  }
  variance /= ns.length;
  return Math.sqrt(variance);
}
```

This version of `sigma` is modifying variables all over the place but it is still pure because all those modifications are hidden from any outside observer. The internals are not pure at all, but to the outside observer, the illusion of purity has been preserved.

We can even go so far as to create a mutable object and still keep the `sigma` function pure.

```
public static double sigma(double… ns) {
  return new SigmaCalculator(ns).invoke();
}

private static class SigmaCalculator {
  private double[] ns;
  private double mu;
  private double variance = 0;
  private double deviation;

  public SigmaCalculator(double… ns) {
    this.ns = ns;
  }

  public double invoke() {
    mu = mean(ns);
    for (double n : ns) {
      deviation = n - mu;
      variance += deviation * deviation;
    }
    variance /= ns.length;
    return Math.sqrt(variance);
  }
}
```

Now we’ve got instance variables and all manner of variable manipulations. And yet, the `sigma` function is pure. None of those impure operations are visible outside the `sigma` function.

The bottom line is that purity is an external characteristic of a function, not an internal characteristic. It does not matter how impure the internals of a function are—that function will be pure so long as all the impurity is hidden from all external observers (including other threads).

And, by the way, that is how all functional programming languages claim purity. They just hide the impurity of the underlying implementation.

Partial Purity

As we just saw, purity is an observed quality, not an intrinsic quality. Therefore, whether a function is pure or not depends upon what is being observed.

Consider the C library function `fopen(char* name, char* mode)`. This function has the side effect of leaving the named file open. This means that `fopen` is not pure, because there are functions, such as `fgetc` and `fputc`, that will not behave properly unless `fopen` is called first. Indeed, if you call `fopen` with the same arguments twice in a row without an intervening call to `fclose`, the second invocation will likely fail because the file is already open. So `fopen` is not pure, because of the side effect that leaves the system in an altered state.

However, we can make the `fopen` function appear to be pure, at least to certain observers, by ensconcing it in another function that hides the side effect.

```
void openAndDo(char* fileName, void (*doTo)(FILE*)) {
  FILE* f = fopen(fileName, "r+");
```

```
    doTo(f);
    fclose(f);
  }
```

This function opens the file, executes the function pointed at by `doTo`, and then closes the file. This puts the system back into the original state, hiding the impurity of `fopen`. For all intents and purposes, this use of `openAndDo` is pure. We might use it as follows:

```
  void show(FILE* f) {
    int c;
    while ((c=fgetc(f)) != EOF)
      putchar(c);
  }

  int main(int ac, char** av) {
    openAndDo("x.x", show);
  }
```

This program prints the `"x.x"` file on standard out. Let's make this a bit more interesting by modifying the contents of the file. We'll add a line end, if the file does not end in one already.

```
  void appendLineEnd(FILE* f) {
    int c;
    int lastC;
    while ((c = fgetc(f)) != EOF) {
      lastC = c;
    }
    if (lastC != '\n')
      fputc('\n', f);
    }

  int main(int ac, char** av) {
    openAndDo("x.x", appendLineEnd);
    openAndDo("x.x", show);
  }
```

Now the program adds the line end and then displays the file. This program does leave the system in an altered state, because the contents of the `"x.x"` file have been changed. However, if we do not care to observe the contents of the `"x.x"` file, then the `openAndDo` function is pure as far as we are concerned.

Conclusion

Clean functions are small, do one thing, and read like well-written prose. Each fits into its place in a hierarchy of functions that lead, one step at a time, from high-level abstractions to low-level details. Clean functions exist within a well-defined context, are isolated from each other, are well and conveniently named, and are pure to all external observers.