API Design Patterns

JJ Geewax

Foreword by Jon Skeet

MANNING

API Design Patterns

JJ GEEWAX

FOREWORD BY JON SKEET

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Christina Taylor
Technical development editor:	Al Krinker
Review editor:	Ivan Martinović
Production editor:	Deirdre S. Hiam
Copy editor:	Michele Mitchell
Proofreader:	Keri Hales
Technical proofreader:	Karsten Strøbæk
Typesetter:	Dennis Dalinnik
Cover designer:	Marija Tudor

ISBN: 9781617295850
Printed in the United States of America

3 Naming

This chapter covers

- Why we should bother caring about names
- What makes some better than others
- How to make choices about language, grammar, and syntax
- How context influences the meaning of a name
- A case study of what can happen with poor name choices

Whether we like it or not, names follow us everywhere. In every software system we build, and every API we design or use, there are names hiding around each corner that will live far longer than we ever intend them to. Because of this, it should seem obvious that it's important to choose great names (even if we don't always give our naming choices as much thought as we should). In this chapter, we'll explore the different components of an API that we'll have to name, some strategies we can employ to choose good names, the high-level attributes that distinguish good names from bad ones, and finally some general principles to help guide us when making tough naming decisions that we'll inevitably run into.

3.1 Why do names matter?

In the world of software engineering generally, it's practically impossible to avoid choosing names for things. If that were possible, we'd need to be able to write chunks of code that used only language keywords (e.g., `class`, `for`, or `if`), which would be unreadable at best. With that in mind, compiled software is a special case. This is because with traditional compiled code, the names of our functions and variables are only important to those who have access to the source code, as the name itself generally disappears during compilation (or minification) and deployment.

On the other hand, when designing and building an API, the names we choose are much more important, as they're what all the users of the API will see and interact with. In other words, these names won't simply get compiled away and hidden from the world. This means we need to put an extraordinary amount of thought and consideration into the names we choose for an API.

The obvious question here becomes, "Can't we just change the names if they turn out to be bad choices?" As we'll learn in chapter 24, changing names in an API can be quite challenging. Imagine changing the name of a frequently used function in your source code and then realizing you need to do a big find-and-replace to make sure you updated all references to that function name. While inconvenient (and even easy in some IDEs), this is certainly possible. However, consider if this source code was available to the public to build into their own projects. Even if you could somehow update all references for all public source code available, there is always going to be private source code that you don't have access to and therefore cannot possibly update.

Put a bit differently, changing public-facing names in an API is a bit like changing your address or phone number. To successfully change this number everywhere, you'd have to contact everyone who ever had your phone number, including your grandmother (who might use a paper address book) and every marketing company that ever had access to it. Even if you have a way to get in touch with everyone who has your number, you'd still need them to do the work of updating the contact information, which they might be too busy to do.

Now that we've seen the importance of choosing good names (and avoiding changing them), this leads us to an important question: What makes a name "good"?

3.2 What makes a name "good"?

As we learned in chapter 1, APIs are "good" when they are operational, expressive, simple, and predictable. Names, on the other hand, are quite similar except for the fact that they aren't necessarily operational (in other words, a name doesn't actually do anything). Let's look at this subset of attributes and a few examples of naming choices, starting with being expressive.

3.2.1 Expressive

More important than anything else, it's critical that a name clearly convey the thing that it's naming. This thing might be a function or RPC (e.g., `CreateAccount`), a resource or message (e.g., `WeatherReading`), a field or property (e.g., `postal_address`), or something else entirely, such as an enumeration value (e.g., `Color.BLUE`), but it should be clear to the reader exactly what the thing represents. This might sound easy, but it's often very difficult to see a name with fresh eyes, forgetting all the context that we've built by working in a particular area over time. This context is a huge asset generally, but in this case it's more of a liability: it makes us bad at naming things.

For example, the term *topic* is often used in the context of asynchronous messaging (e.g., Apache Kafka or RabbitMQ); however, it's also used in a specific area of machine learning and natural language processing called *topic modeling*. If you were to use the term *topic* in your machine learning API, it wouldn't be all that surprising that users might be confused about which type of topic you're referring to. If that's a real possibility (perhaps your API uses both asynchronous messaging and topic modeling), you might want to choose a more expressive name than `topic`, such as `model_topic` or `messaging_topic` to prevent user confusion.

3.2.2 Simple

While an expressive name is certainly important, it can also become burdensome if the name is excessively long without adding additional clarity. Using the example from before (`topic`, referring to multiple different areas of computer science), if an API only ever refers to asynchronous messaging (e.g., an Apache Kafka–like API) and has nothing to do with machine learning, then `topic` is sufficiently clear and simple, while `messaging_topic` wouldn't add much value. In short, names should be expressive but only to the extent that each additional part of a name adds value to justify its presence.

On the other hand, names shouldn't be oversimplified either. For example, imagine we have an API that needs to store some user-specified preferences. The resource might be called `UserSpecifiedPreferences`; however, the `Specified` isn't adding very much to the name. On the other hand, if we simply called the resource `Preferences`, it's unclear whose preferences they are and could cause confusion down the line when there are system- or administrator-level preferences that need to be stored and managed. In this case, `UserPreferences` seems to be the sweet spot between an expressive name and a simple name, summarized in table 3.1.

Table 3.1 Choosing between simple and expressive names

Name	Notes
`UserSpecifiedPreferences`	Expressive, but not simple enough
`UserPreferences`	Simple enough and expressive enough
`Preferences`	Too simple

3.2.3 Predictable

Now that we've gone through the balance between expressive and simple, there's one final and very important aspect of choosing a good name: predictability. Imagine an API that uses the name `topic` to group together similar asynchronous messages (similar to Apache Kafka). Then imagine that the API uses the name `messaging_topic` in other places, without much reason for choosing one or the other. This leads to some pretty frustrating and unusual circumstances.

Listing 3.1 Example frustrating code due to inconsistent naming

```
function handleMessage(message: Message) {
  if (message.topic == "budget.purge") {
    client.PurgeTopic({
      messagingTopic: "budget.update"
    });
  }
}
```

Here we use the name topic to read the topic of a given message.

Here we use the name messagingTopic to represent the same concept.

In the odd case that this doesn't seem frustrating, consider an important principle we're violating here. In general, we should use the same name to represent the same thing and different names to represent different things. If we take that principle as axiomatic, this leads to an important question: how is `topic` different from `messagingTopic`? After all, we used different names, so they must represent different concepts, right?

The basic underlying goal is to allow users of an API to learn one name and continue building on that knowledge to be able to predict what future names (e.g., if they represent the same concept) would look like. By using `topic` consistently throughout an API when we mean "the topic for a given message" (and something else when we mean something different), we're allowing users of an API to build on what they've already learned rather than confusing them and forcing them to research every single name to ensure it means what they would assume.

Now that we have an idea of some of the characteristics of good names, let's explore some general guidelines that can act as guard rails when naming things in an API, starting with the fundamental aspects of language, grammar, and syntax.

3.3 Language, grammar, and syntax

While code is all about ones and zeros, fundamentally stored as numbers, naming is a primarily subjective construct we express using language. Unlike programming languages, which have very firm rules about what's valid and what's not, language has evolved to serve people more than computers, making the rules much less firm. This allows our naming choices to be a bit more flexible and ambiguous, which can be both a good and bad thing.

On the one hand, ambiguity allows us to name things to be general enough to support future work. For example, naming a field `image_uri` rather than `jpeg_uri` prevents us from limiting ourselves to a single image format (JPEG). On the other hand, when there are multiple ways to express the same thing, we often tend to use them interchangeably, which ultimately makes our naming choices unpredictable (see section 3.2.3) and results in a frustrating and cumbersome API. To avoid some of this, even though "language" has quite a bit of flexibility, by imposing some rules of our own, we can avoid losing the predictability we value so highly in a good API. In this section, we'll explore some of the simple rules related to language that can help minimize some of the arbitrary choices we'll have to make when naming things.

3.3.1 Language

While there are many languages spoken in the world, if we had to choose a single language that was used the most in software engineering, currently American English is the leading contender. This isn't to say that American English is any better or worse than other languages; however, if our goal is maximum interoperability across the world, using anything other than American English is likely to be a hindrance rather than a benefit.

This means that English language concepts should be used (e.g., `BookStore` rather than `Librería`) and common American-style spellings should generally be preferred (e.g., *color* rather than *colour*). This also has the added benefit of almost always fitting comfortably into the ASCII character set, with a few exceptions where American English has borrowed from other languages (e.g., *café*).

This doesn't mean that API comments must be in American English. If the audience of an API is based exclusively in France, it might make sense to provide documentation (which may or may not be automatically generated from API specification comments) in French. However, the team of software engineers consuming the API is likely to use other APIs, which are unlikely to be exclusively targeted toward customers in France. As a result, it still holds that even if the audience of an API doesn't use American English as their primary language, the API itself should still rely on American English as a shared common language across all parties using lots of different APIs together.

3.3.2 Grammar

Given that an API will use American English as the standard language, this opens quite a few complicated cans of worms as English is not exactly the simplest of languages with many different tenses and moods. Luckily, pronunciation won't be an issue as source code is a written rather than spoken language, but this doesn't necessarily alleviate all the potential problems.

Rather than attempt to dictate every single aspect of American English grammar as it applies to naming things in an API, this section will touch on a few of the most common issues. Let's start by looking at actions (e.g., RPC methods or RESTful verbs).

IMPERATIVE ACTIONS

In any API, there will be something equivalent to a programming language's "functions," which do the actual work expected of the API. This might be a purely RESTful API, which relies only on a specific preset list of actions (Get, Create, Delete, etc.), then you don't have all that much to do here as all actions will take the form of `<StandardVerb><Noun>` (e.g., `CreateBook`). In the case of non-RESTful or resource-oriented APIs that permit nonstandard verbs, we have more choices for how we name these actions.

There is one important aspect that the REST standard verbs have in common: they all use the imperative mood. In other words, they are all commands or orders of the verb. If this isn't making a lot of sense, imagine a drill sergeant in the Army shouting at you to do something: "Create that book!" "Delete that weather reading!" "Archive that log entry!" As ridiculous as these commands are for the Army, you know exactly what you're supposed to do.

On the other hand, sometimes the names of the functions we write can take on the indicative mood. One common example is when a function is investigating something, such as `String.IsNullOrEmpty()` in C#. In this case, the verb "to be" takes on the indicative mood (asking a question about a resource) rather than the imperative mood (commanding a service to do something).

While there's nothing fundamentally wrong with our functions taking on this mood, when used in a web API it leaves a few important questions unanswered. First, with something that looks like it can be handled without asking a remote service, "Does `isValid()` actually result in a remote call or is it handled locally?" While we hope that users assume all method calls are going over the network, it's a bit misleading to have what appears to be a stateless call do so.

Secondly, what should the response look like? Take the case of an RPC called `isValid()`. Should it return a simple Boolean field stating whether the input was valid? Should it return a list of errors if that input wasn't valid? On the other hand, `GetValidationErrors()` is more clear: either it returns an empty list if the input is completely valid or a list of errors if it isn't. There's no real confusion about the shape the response will take.

PREPOSITIONS

Another area of confusion when choosing names centers on prepositions, such as "with," "to," or "for." While these words are very useful in everyday conversation, when used in the context of a web API, particularly in resource names, they can be indicative of more complicated underlying problems with the API.

For example, a Library API might have a way to list `Book` resources. If this API needed a way to list `Book` resources and include the `Author` resources responsible for that book, it may be tempting to create a new resource for this combination: `BookWithAuthor` (which would then be listed by calling `ListBooksWithAuthors` or something similar). This might seem fine at first glance, but what about when we need to list `Book` resources with the `Publisher` resources embedded? Or both `Author` and

Publisher resources? Before we know it, we'll have 30 different RPCs to call depending on the different related resources we want.

In this case, the preposition we want to use in the name ("with") is indicative of a more fundamental problem: we want a way to list resources and include different attributes in the response. We might instead solve this using a field mask or a view (see chapter 8) and avoid this oddly-named resource at the same time. In this case, the preposition was an indication that sometimes wasn't quite right. So even though prepositions probably shouldn't be forbidden entirely (e.g., maybe a field would be called bits_per_second), these tricky little words act a bit like a *code smell*, hinting at something being not quite right and worth further investigation.

Pluralization

Most often, we'll choose the names for things in our APIs to be the singular form, such as Book, Publisher, and Author (rather than Books, Publishers, or Authors). Further, these name choices tend to take on new meanings and purposes through the API. For example, a Book resource might be referenced somewhere by a field called Author.favoriteBook (see chapter 13). However, things can sometimes get messy when we need to talk about multiples of these resources. To make things more complicated, if an API uses RESTful URLs, the collection name of a bunch of resources is almost always plural. For example, when we request a single Book resource, the collection name in the URL will almost certainly be something like /books/1234.

In the case of the names we've used as examples (e.g., Book), this isn't much of an issue; after all, mentioning multiple Book resources just involves adding an "s" to pluralize the name into Books. However, some names are not so simple. For example, imagine we're making an API for a podiatrist's office (a foot doctor). When we have a Foot resource, we'll need to break this pattern of just adding an "s," leading to a feet collection.

This example certainly breaks the pattern, but at least it's clear and unambiguous. What if our API deals with people and therefore has a Person resource. Is the collection persons? Or people? In other words, should Person(id=1234) be retrieved by visiting a URL that looks like /persons/1234 or /people/1234? Luckily our guidelines about using American-style English prescribes an answer: use people.

Other cases are more frustrating still. For example, imagine we are working on an API for the aquarium. What is the collection for an Octopus resource? As you can see, our choice of American English sometimes comes back to bite us. What's most important though is that we choose a pattern and stick to it, which often involves a quick search for what the grammarians say is correct (in this case, "octopuses" is perfectly fine). This also means that we should never assume the plural of a resource can be created simply by adding an "s"—a common temptation for software engineers looking for patterns.

3.3.3 Syntax

We've reached the more technical aspects of naming. As with the previous aspects we've looked at, when it comes to syntax the same guidelines are in place. First, pick something and stick to it. Second, if there's an existing standard (e.g., American English spellings), use that. So what does this mean in a practical sense? Let's start with case.

CASE

When we define an API, we need to name the various components, which are things like resources, RPCs, and fields. For each of these, we tend to use a different case, which is sort of like a format in which the name is rendered. Most often, this rendering is only apparent in how multiple words are strung together to make a single lexical unit. For example, if we had a field that represents a person's given name, we might need to call that field "first name." However, in almost all programming languages, spaces are the lexical separation character, so we need to combine "first name" into a single unit, which opens the door for lots of different options, such as "camel case," "snake case," or "kebab case."

In camel case, the words are joined by capitalizing the letters of all words after the first, so "first name" would render as `firstName` (which has capital letters as humps like a camel). In snake case, words are joined using underscore characters, as in `first_name` (which is meant to look a bit like a snake). In kebab case, words are joined with hyphen characters, as in `first-name` (which looks a bit like a kebab skewering the different words). Depending on the language used to represent an API specification, different components are rendered in different cases. For example, in Google's Protocol Buffer language, the standard is for messages (like TypeScript interfaces) to use upper camel case, as in `UserSettings` (note the uppercase "U") and snake case for field names, as in `first_name`. On the other hand, in open API specification standards, field names take on camel case, as in `firstName`.

As noted earlier, the specific choice isn't all that important so long as the choices are used consistently throughout. For example, if you were to use the name `user_settings` for a protocol buffer (https://developers.google.com/protocol-buffers) message, it would be very easy to think that this is actually a field name and not a message. As a result, this is likely to cause confusion to anyone using the API. Speaking of types, let's take a brief moment to look at reserved words.

RESERVED KEYWORDS

In most API definition languages, there will be a way to specify the type of the data being stored in a particular attribute. For example, we might say `firstName: string` to express in TypeScript that the field called `firstName` contains a primitive string value. This also implies that term *string* has some special meaning, even if used in a different position in code. As a result, it can be dangerous to use restricted keywords as names in your API and should be avoided whenever possible.

If this seems difficult, it can be worthwhile to spend some time thinking about what a field or message truly represents and not what the easiest option is. For example, rather than “to” and “from” (from being those special reserved keywords in languages like Python), you might want to try using more specific terminology such as “sender” and “recipient” (if the API is about messages) or maybe “payer” and “payee” (if the API is about payments).

It’s also important to consider the target audience of your API. For example, if the API will only ever be used in JavaScript (perhaps it’s intended to be used exclusively in a web browser), then keywords in other languages (e.g., Python or Ruby) may not be worth worrying about. That said, if it’s not much work, it’s a good idea to avoid keywords in other languages. After all, you never know when your API might end up being used by one of these languages.

Now that we’ve gone through some of these technical aspects, let’s jump up a level and talk about how the context in which our API lives and operates might affect the names we choose.

3.4 Context

While names on their own can sometimes convey all the information necessary to be useful, more often than not we rely on the context in which a name is used to discern its meaning and intended use. For example, when we use the term *book* in an API, we might be referring to a resource that lives in a Library API; however, we might also be referring to an action to be taken in a Flight Reservation API. As you can imagine, the same words and terminology can mean completely different things depending on the context in which they’re used. What this means is that we need to keep the context in which our API lives in mind when choosing names for it.

It’s important to remember that this goes both ways. On the one hand, context can impart additional value to a name that might otherwise lack specific meaning. On the other hand, context can lead us astray when we use names that have a very specific meaning but don’t quite make sense in the given context. For example, the name “record” might not be very useful without any context nearby, but in the context of an audio recording API, this term absorbs the extra meaning imparted from the API’s general context.

In short, while there are no strict rules about how to name things in a given context, the important thing to remember is that all the names we choose in an API are inextricably linked to the context provided by that API. As a result, we should be cognizant of that context and the meaning it might impart (for better or worse) when choosing names.

Let’s change direction a bit and talk about data types and units, specifically how they should be involved in the names we choose.

3.5 Data types and units

While many field names are descriptive without units (e.g., `firstName: string`), others can be extraordinarily confusing without units. For example, imagine a field called “size.” Depending on the context (see section 3.4), this field could have entirely different meanings but also entirely different units. We can see the same field (`size`) that would have entirely different and, in many cases, confusing meaning and units.

Listing 3.2 An audio clip and image using size fields

```
interface AudioClip {
  content: string;
  size: number;
}

interface Image {
  content: string;
  size: number;
}
```

This might contain Base64-encoded binary audio content.

The units of this field are confusing. Is it the size in bytes? Or the duration in seconds of the audio? Or dimensions of the image? Or something else?

In this example, the size field could mean multiple things, but those different meanings also would lead to very different units (e.g., bytes, seconds, pixels, etc.). Luckily this relationship goes both ways, meaning that if the units were present somewhere the meaning would become more clear. In other words, `sizeBytes` and `sizeMegapixels` are much more clear and obvious than just `size`.

Listing 3.3 An audio clip and image using clearer size fields with units

```
interface AudioClip {
  content: string;
  sizeBytes: number;
}

interface Image {
  content: string;
  sizeMegapixels: number;
}
```

Now the meaning of these size fields is much more clear because the units are provided.

Does this mean that we should always simply include the unit or format for any given field in all scenarios? After all, that would certainly minimize any confusion in cases like those shown. For example, imagine that we wanted to store the dimensions of the image in pixels resource along with the size in bytes. We might have two fields called `sizeBytes` and `dimensionsPixels`. But the dimensions are actually more than one number: we need both the length and the width. One option is to use a string field and have the dimensions in some well-known format.

Listing 3.4 An image storing the dimensions in pixels using a string field

```
interface Image {
  content: string;
  sizeBytes: number;

  // The dimensions (in pixels). E.g., "1024x768".
  dimensionsPixels: string;
}
```

The format of the field is expressed in a leading comment on the field itself.

The units of the field are clear (pixels), but the primitive data type can be confusing.

While this option is technically valid and is certainly clear, it displays a bit of an obsession toward using primitive data types always, even when they might not make sense. In other words, just like sometimes names become more clear and usable when a unit is included in the name, other times a name can become more clear when using a richer data type. In this case, rather than using a string type that combines two numbers, we can use a `Dimensions` interface that has length and width numeric values, with the unit (pixels) included in the name.

Listing 3.5 An image with dimensions relying on a richer data type

```
interface Image {
  content: string;
  sizeBytes: number;
  dimensions: Dimensions;
}

interface Dimensions {
  lengthPixels: number;
  widthPixels: number;
}
```

In this case, the dimensions field name doesn't need a unit in the name as the richer data type conveys the meaning.

The units of the field are clear (pixels) without any special string formatting.

In this case, the meaning of the `dimensions` field is clear and obvious. Further, we don't have to unpack some special structural details of the field itself because the `Dimensions` interface has done this for us. Let's wrap up this topic of naming by looking at some case studies of what can go wrong when we don't take the proper caution when choosing names in an API.

3.6 Case study: What happens when you choose bad names?

These guidelines about how to choose good names and the various aspects worth considering during that choosing process are all well and good, but it might be worthwhile to look at a couple of real-world examples using names that aren't quite right. Further, we can see the end consequences of these naming choices and the potential issues they might cause. Let's start by looking at a naming issue where a subtle but important piece is left out.

SUBTLE MEANING

If you were to walk into a Krispy Kreme donut shop and ask for 10 donuts, you'd expect 10 donuts, right? And you'd be surprised if you only got 8 donuts? Maybe if you got 8 donuts you'd assume that the store must be completely out of donuts. It certainly wouldn't seem right that you'd get 8 donuts right away, then have to ask for 2 more donuts to get your desired 10.

What if, instead, you only had a way to ask for a maximum of N donuts. In other words, you could only ask the cashier "Can I have up to 10 donuts?" You'd get back any number of donuts, but never more than 10. (And keep in mind that this might result in you getting zero donuts!) Suddenly the weird behavior in the first donut shop example makes sense. It's still inconvenient (I've not yet seen a donut shop with this kind of ordering system), but at least it's not baffling and surprising.

In chapter 21, we'll learn about a design pattern that demonstrates how to page through a bunch of resources during a list standard method operation in a way that's safe, clear, and scales nicely to lots and lots of resources. And it turns out that this exclusive ability to ask only for the maximum (and not an exact amount) is exactly how the pagination pattern works (using a `maxPageSize` field).

The folks over at Google (for historical reasons) follow the pagination pattern as described except for one important difference: instead of specifying a `maxPageSize` to say "give me a maximum of N items," requests specify a `pageSize`. These three missing characters lead to an extraordinarily large amount of confusion, just like the person ordering donuts: they think they're asking for an exact number, but they actually are only able to ask for a maximum number.

The most common scenario is when someone asks for 10 items, gets back 8, and thinks that there must be no more items (just like we might assume the donut shop is out of donuts). In fact, this isn't the case: just because we got 8 back doesn't mean the shop is out of donuts; it just means that they have to go find more in the back. This ultimately results in API users to miss out on lots of items because they stop paging through the results before the actual end of the list.

While this might be frustrating and lead to some inconvenience, let's look at a more serious mistake made by mixing up units for a field.

UNITS

Back in 1999, NASA planned to maneuver the Mars Climate Orbiter into an orbit about 140 miles above the surface. They did a bunch of calculations to figure out exactly what impulse forces to apply in order to get the orbiter into the right position and then executed the maneuver. Unfortunately, soon after that the team noticed that the orbiter was not quite where it was supposed to be. Instead of being at 140 miles above the surface, it was far lower than that. In fact, calculations made later seemed to show that the orbiter would've been within 35 miles of the surface. Sadly, the minimum altitude the orbiter could survive was 50 miles. As you'd expect, going below that floor means that the orbiter was likely destroyed in Mars's atmosphere.

In the investigation that followed, it was discovered that the Lockheed Martin team produced output in US standard units (specifically, lbf-s or pound-force seconds) whereas the NASA teams worked in SI units (specifically, N-s or Newton seconds). A quick calculation shows that 1 lbf-s is equivalent to 4.45 N-s, which ultimately resulted in the orbiter getting more than four times the amount of impulse force needed, which ultimately sent it below its minimum altitude.

Listing 3.6 A (very simplified) example of the API for calculations on the MCO

```
abstract class MarsClimateOrbiter {
  CalculateImpulse(CalculateImpulseRequest):
      CalculateImpulseResponse;
  CalculateManeuver(CalculateManeuverRequest):
      CalculateManeuverResponse;
}

interface CalculateImpulseResponse {
  impulse: number;
}

interface CalculateManeuverRequest {
  impulse: number;
}
```

The CalculateImpulseRequest and CalculateManeuverResponse interfaces are omitted for brevity.

Here we have the impulse calculated, but there are no units! This implies we can feed the previous output as the next input.

If, on the other hand, the integration point had included the units in the names of the fields, the error would've been far more obvious.

Listing 3.7 Alterations to the example interfaces to include units

```
interface CalculateImpulseResponse {
  impulsePoundForceSeconds: number;
}

interface CalculateManeuverRequest {
  impulseNewtonSeconds: number;
}
```

Here it becomes obvious that you can't just take the output of one API method and feed it into the next method due to the different units.

Obviously the Mars Climate Orbiter was a far more complicated piece of software and machinery than portrayed here, and it's unlikely that this exact scenario (https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure) could have been avoided simply by using more descriptive names. That said, it's a good illustration of why descriptive names are valuable and can help highlight differences in assumptions, particularly when coordinating between different teams.

3.7 Exercises

1. Imagine you need to create an API for managing recurring schedules ("This event happens once per month"). A senior engineer argues that storing a value for seconds between events is sufficient for all the use cases. Another engineer thinks that the API should provide different fields for various time units (e.g., seconds, minutes, hours, days, weeks, months, years). Which design covers the correct meanings of the intended functionality and is the better choice?
2. In your company, storage systems use gigabytes as the unit of measurement (10^9 bytes). For example, when creating a shared folder, you can set the size to 10 gigabytes by setting `sizeGB = 10`. A new API is launching where networking throughput is measured in Gibibits (2^{30} bits) and wants to set bandwidth limits in terms of Gibibits (e.g., `bandwidthLimitGib = 1`). Is this too subtle a difference and potentially confusing for users? Why or why not?

Summary

- Good names, like good APIs, are simple, expressive, and predictable.
- When it comes to language, grammar, and syntax (and other arbitrary choices), often the right answer is to pick something and stick to it.
- Prepositions in names are often API smells that hint at some larger underlying design problem worth fixing.
- Remember that the context in which a name is used both imparts information and can be potentially misleading. Be aware of the context in place when choosing a name.
- Include the units for primitives and rely on richer data types to help convey information not present in a name.