Designing Programming Languages for Writing Maintainable Software

Aaron Friesen
University of Nebraska-Lincoln

DESIGNING PROGRAMMING LANGUAGES FOR WRITING MAINTAINABLE SOFTWARE

An Undergraduate Honors Thesis

Submitted in Partial Fulfillment of

University Honors Program Requirements

University of Nebraska–Lincoln

by

Aaron Friesen, BS

Software Engineering

College of Engineering

May 3, 2023

Faculty Mentor:

Christopher Bohn, PhD, School of Computing

Abstract

Maintainability is crucial to the long-term success of software projects. Among other factors, it is affected by the programming language in which the software is written. Programming language designers should be conscious of how their design decisions can influence software maintainability. Non-functional properties of a language can affect the readability of source code in ways beyond the control of programmers. Language features can cause or prevent certain classes of bugs, and runtime issues especially can require significant maintenance effort. Tools external to the language, especially those developed and distributed by language implementers, can aid in the creation of maintainable software. Languages designed with these aspects in mind will ease the burden placed on software maintainers by facilitating the development of robust, high-quality software.

Key Words: Programming Languages, Maintainability, Software Engineering

1 Introduction

Maintenance is one of the main activities of the software engineering process. It involves making incremental changes to software to fix issues, change or add functionality, and keep dependencies updated. Fulfilling these duties tends to involve reading code more than writing code. To complicate matters, the maintainers of a software system are not necessarily the same developers who originally implemented it.

Maintainability refers to how easy it is maintain a given software system. Various high-level and low-level factors contribute towards or detract from maintainability. For example, *The Zen of Go* highlights clarity, readability, and simplicity as positive aspects of maintainability [1]. Software with poor maintainability becomes harder to change as it grows in scale. Retroactively improving maintainability requires rework, which is likely to require more developer time overall than if the software were originally developed with maintainability in mind. However, not all systems are created with maintainability as a priority, and thus it is worth considering other ways to promote software maintainability.

Although maintainability ultimately comes down to the software system itself, all code is influenced by the language in which it was written. Programming language design can therefore have an effect on maintainability. Non-functional properties of the language, such as its syntax, can significantly affect the readability and understandability of code written in the language. Features of the language, such as its type system, can cause or prevent certain classes of bugs. Finally, the tools that comprise the language's development ecosystem can make it easier to develop stable, consistent, well-documented software.

2 Non-functional aspects

2.1 Simplicity

Simple code is easier to maintain than complex code. At a high level, a simple architecture makes it clear to the maintainer how changes to one component may affect another. At a low level, simple functions, classes, and modules are easier to understand, debug, and modify. Especially at this lower level, the complexity of the underlying programming language can influence the complexity of the code.

A complex language makes it easier for a programmer to write code that is hard to understand. Languages that provide more features allow programmers to use more of these features in their code. For example, the relative simplicity of C limits the kind of code that can be written with it, but at the same time reduces the likelihood that a reader will be unfamiliar with a particular feature used therein. On the other hand, C++ is more complex than C, so when a programmer uses a certain feature, it is less likely to be understood by any given reader. Maintainers that are not familiar with a particular feature often need to spend time learning about it before they can properly understand and maintain it. The use of additional features can simplify or improve code, but the cost of learning these features still remains. Therefore, before implementing a new feature, designers should consider the value it adds relative to its complexity.

Languages that have more complex syntax invite programmers to use this syntax to create complex expressions that are harder for maintainers to parse and debug. For example, the ternary operator found in many languages allows programmers to abbreviate simple expressions, such as conditional assignments, instead of writing a full conditional statement. However, programmers can also combine and nest ternary expressions to create complex statements that would be much more understandable if they were written as normal conditionals.

When deciding whether or not to include a new feature in a language, consider if a simpler solution already exists. For example, C includes pre-increment (`++x`) and post-increment (`x++`) operators, in addition to the simpler addition assignment operator (`x += 1`). These increment operators can be used to shorten certain statements, but they introduce complexities with regard to evaluation order. Other languages like Python and Rust intentionally omit pre-increment and post-increment operators to reduce complexity and ambiguity [2].

It is worth noting that it one can almost always write simple code in a complex language by neglecting to use complex features and syntactic elements. This can require intention on behalf of the programmer, though, especially if they are experienced with these complex aspects. On the other hand, complex code can still emerge in a simple language when simple features are combined in convoluted ways. However, keeping the language simple reduces the complexity contributed by the language, allowing maintainers to focus on the complexity of the code itself. Zig, a programming language which strives for simplicity, suggests that a simple language lets you “focus on debugging your application rather than debugging your programming language knowledge” [3].

2.2 Readability

Readable code is easier to maintain for reasons similar to why simple code is easier to maintain. The less time it takes the maintainer to read and understand the code, the less time it takes them to start doing actual maintenance.

One way languages can facilitate readability is by using readable keywords. For example, conditionals in POSIX-compliant shell scripts are opened with the `if` and `then` keywords, and closed by the `fi` keyword. `fi` was chosen as the closing keyword because it is `if` spelled backwards, but a new developer might not immediately realize or understand this reasoning. Furthermore, this logic is not applied consistently throughout the language. For instance, `while` loops are closed by the `done` keyword.

Similarly, built-in functions and standard library functions should be readable. The C standard libraries are full of abbreviated function names, such as `malloc`, `snprintf`, and `fgets`, for historical reasons. These abbreviations become more familiar to programmers as they spend more time with the language, but each abbreviation that a maintainer needs to look up requires a context switch that can derail their train of thought.

Another way languages can promote readability is by replacing symbols with text where appropriate. For example, the C-style boolean operators `&&`, `||`, and `!` can be replaced by their corresponding English names `and`, `or`, and `not`. This allows many boolean expressions to be read like natural language sentences, without increasing verbosity. Similarly, Python replaces the C-style ternary operator `condition ? trueValue : falseValue` with `trueValue if condition else falseValue`, changing the order of the operands to improve readability. Keywords tend to be longer than symbols, so when choosing to introduce a new symbol or keyword, weigh readability versus verbosity. Consider how much longer the keyword is, how often the symbol or keyword will be used, and how much more readable the keyword is than the symbol. Also keep in mind that ubiquitous symbols are usually just as readable as keywords. These include arithmetic operators, as well as symbols like semicolons, curly braces, and square brackets, when used in the same way as C.

As in the case of simplicity, the readability of a language is not directly proportional to the readability of code written with it. However, a readable syntax makes it easier for programmers to write readable code, and harder for them to write unreadable code.

2.3 Familiarity

An easy way for a language to achieve readability is to borrow common syntactic elements from well-known and influential languages. For example, numerous languages, such as Java, JavaScript, and Dart, borrow much of their basic syntax from C. Code written in a language with familiar syntax is likely to be at least as readable as code written in a language with a unique but inherently readable syntax. Supporting familiar design paradigms, such as imperative, functional, or object-oriented, lets programmers and maintainers apply their intuition from other languages.

When designing a language, one should consider which languages will be familiar to its users. For instance, a systems programming language with a Ruby-like syntax is unlikely to feel familiar to developers of embedded systems, who are probably more accustomed to C-like languages.

2.4 Explicitness

When reading a statement or expression, its behavior should be unambiguous. This is not only important to developers and maintainers, who benefit from a clearer understanding of the code, but it is also important when writing a parser for the language. Implicit casting is an example where a lack of explicitness can cause unintended results. For instance, operations on integers and floating point numbers involving implicit casts can result in an unexpected loss of precision.

Furthermore, understanding a statement's behavior should require a minimum of context. Searching for necessary context can take time, especially if that context could be in another file, and especially if it's not obvious that any external context exists in the first place. For this reason, hidden control flow harms readability [4]. For example, a method in a Python class can be declared as a "property", allowing it to be called as if one were accessing a member variable. This could lead a developer to think they are, in fact, just accessing a member variable. They might then access it multiple times, not realizing they are calling a method each time, which could reduce performance. Confusing bugs could occur if the process of computing the property generated side effects.

Operator overloading is another source of hidden control flow. A programmer or maintainer may see a common operator, such as plus or minus, and expect it to work as it normally does for primitives. However, if a custom type overloaded the operator, it could be calling a method instead. C++ notably supports operator overloading, which is dangerous considering its use in low-level programming and embedded systems, where an unintended method call can affect performance and memory more than usual.

2.5 Non-redundancy

In general, a programming language should provide one and only one way of accomplishing a given basic task. This sentiment is echoed in the design philosophies of multiple programming languages. *The Zen of Python* states, "there should be one — and preferably only one — obvious way to do it" [5]. Similarly, one principle in *The Zen of Zig* is "only one obvious way to do things" [6].

If a programming language provides many ways of accomplishing the same task, it may be slightly easier to learn the language, but it will be more difficult to read code written in it. Languages that heed the Principle of Least Astonishment, such as Ruby, can be particularly prone to this. If two programmers use differing approaches, this can be distracting when one programmer reads code written by the other. Furthermore, inconsistencies will arise across the code base, which can similarly distract or confuse maintainers.

2.6 Standardized style

Differences in coding styles can reduce readability and serve as a distraction to readers. Even if an organization implements a style guide for all of its software projects, a developer coming from another organization may be accustomed to a different style, which could slow their onboarding. Similarly, if developers in this organization need to interact with third-party code, they may face the same difficulties. One solution to this problem, from the standpoint of a language designer, is to suggest a standard style to be used by every program written in the language. For example, in PEP 8, the designers of Python suggest a common style guide for Python code [7]. If such a style becomes widely adopted, this smooths the transition between reading code written different people, even across code bases and organizations.

3 Language features

3.1 Type system

It could be argued that the enforcement of a type system is essential to any practical, high-level language. From a perspective of maintainability, a type system helps prevent confusing errors and makes code more readable. A type system can be categorized using three axes: strong or weak, static or dynamic, and explicit or implicit. A type system tending toward the strong, static, and explicit sides of these axes tends to provide the greatest benefits to maintainability.

The strength of a type system is not necessarily a formally defined concept, but one way of interpreting it is as a measure of type safety. A weaker type system may allow type errors to occur, or silently attempt to resolve them during runtime, rather than preventing them outright. Weaker typing tends to allow more confusing behaviors and can result in strange bugs. In the case of C, programmers can bypass the type system with type coercion and type punning. Although this is sometimes necessary for low-level development, such behavior should be considered unsafe and it should be difficult to do unintentionally. In the case of JavaScript, programmers can inadvertently mix data types, resulting in implicit casts, which can in turn cause bugs that are hard to track down. Type issues like these were significant enough in JavaScript to motivate the development of TypeScript. By adding syntax for types, TypeScript aims to "help developers feel more confident in their code, and save considerable amounts time in validating that they have not accidentally broken the project" [8].

Static type checking catches type errors at compile-time rather than allowing them to make it to runtime undetected. This reduces the number of bugs that can arise at runtime, reducing the amount of maintenance the software will need. Runtime errors can be caught by unit tests, but extensive code coverage may be needed for a reasonable degree of confidence. This preference for detecting errors at compile-time rather than runtime is broadly applicable. *The Zen of Zig* states generally that "compile errors are better than runtime crashes" [6]. Nonetheless, pure static type systems lack some of the major advantages of dynamic type systems, especially runtime type information, which enables features like dynamic dispatch and downcasting. Thus, a hybrid type system that performs static analysis but exposes type information at runtime offers a good compromise.

Explicit typing makes code more verbose but also more readable. For instance, under an implicit type system, if a maintainer wanted to know the type of a function return value, they would need to inspect the function's return statements, which could potentially involve calls to other functions. Furthermore, with dynamic, implicit typing, it is possible that different return statements in the same function could return different types. With an explicit type system, the function would need to declare what type or types it can return, removing the need for such a search and making the programmer's intentions clearer. Specifying types takes time when writing code, but the author of the code is more likely to know what types they are working with than a reader would.

3.2 Memory safety

In languages with manual memory management, improper memory manipulation is the cause of many strange errors. Segmentation faults, buffer overflows, write-after-free errors, and double-free errors are frequently difficult to debug. Perhaps worse still are cases where a valid memory address in the program's address space is unintentionally written to, not crashing the program but causing entirely unexpected behavior. Errors like these may go undetected for a long time and require a significant amount of maintainers' time to fix.

Memory safety mechanisms come in multiple forms and can significantly reduce the likelihood that memory errors will occur or go undetected. The most dramatic solution is to use automatic memory management, which makes it practically impossible for programmers to cause memory-related errors in the first place. However, garbage collection introduces additional overhead, which generally occurs in spikes when the garbage collector runs. These performance dips are unacceptable for certain systems, like real-time operating systems and video games. Multi-threaded garbage collectors, such as Java's Z garbage collector, can alleviate this problem by doing most of their work concurrently.

However, manual memory management is essential in some use cases, such as kernels and embedded systems. It is still possible to ensure memory safety in a language with manual memory management. Rust does this by implementing a borrow checker, which can detect memory errors at compile time. Rust's borrow checker relies on the Resource Acquisition Is Initialization paradigm, and Rust's models of ownership, lifetimes, and borrowing [9]. This allows Rust to enforce memory safety at compile time with little to no impact on runtime performance. It comes at the cost of extra restrictions on how memory can be accessed and modified, since, as a static analysis, it underapproximates the space of valid programs. This model of memory management is also less familiar to developers and maintainers, although the time saved debugging memory errors may counteract the additional time spent understanding the model.

If a borrow checker cannot be implemented in a language for any reason, there are still ways for the language to promote memory safety. Hare is one such programming language with manual memory management that lacks a borrow checker. However, it includes safety features such as bounds-checked arrays, mandatory initializers, mandatory error handling, and exhaustive matching [10]. Features like these can improve safety in all kinds of languages.

Even languages with automatic memory management often include the concept of null to signify the lack of a value. Thus, null dereference errors are still common among them. Some languages, such as Dart, attempt to combat this by implementing sound null safety [11]. This makes variables non-nullable by default, and enforces this property with static code analysis. This solution also works for pointers in languages with manual memory management. In Hare, nullable pointers must be checked before they can be dereferenced [10].

3.3 Exception handling

Exceptional cases are inevitable in software, and handling them is vital to software quality. If exceptions are not handled, they can crash a program, or in some cases, they can be silently ignored and allow undefined behavior to arise. For example, when calling some library functions in C, programs must check the function's return value for an error status code, or check `errno` to see if an error occurred. However, it is usually possible for a program to ignore the possibility of an error and continue running. This may cause unexpected results that may not manifest until later in the program's execution, to the detriment of traceability.

Out of these two options, programming languages should prefer to cause a program to crash when an exceptional situation occurs, ensuring that the programmer is aware of the issue. As *The Zen of Zig* says, "runtime crashes are better than bugs" [6]. It should be possible to ignore an error, but doing so should be an intentional decision rather than an accidental omission. According to *The Zen of Python*, "errors should not be silent, unless explicitly silenced" [5].

Many languages provide ways of defining exceptional control flow. One common approach in object-based languages, including Java and Python, is to treat exceptions as objects. Exceptions are thrown or raised and propagate back through the call stack until caught or until the main function is exited and the program crashes. The exceptions that can be thrown from a given function may be explicitly specified, such as in Java with the `throws` keyword, or not, such as in Python. With explicit exception specification, it is possible to statically determine which exceptions may be uncaught in a given scope. This is preferable from a maintainability perspective, because it reduces the likelihood that maintainers will need to write code to handle exceptional cases that are not detected until after deployment.

Rust offers a different approach to error handling that distinguishes between recoverable and unrecoverable errors [9]. Recoverable errors can be handled by the program, whereas unrecoverable errors terminate the program. This distinction is useful, because it prevents unrecoverable errors from being suppressed while providing additional ways of handling recoverable errors. A recoverable error is represented using a `Result` object, which stores either the expected result of an operation, or an error type. Rust provides various ways to handle `Result`s, including panicking via `unwrap` or `expect`, propagating the error, using closures, or handling each possibility as a separate case with pattern matching. Hare uses tagged unions for error handling, which serve a similar role of storing either an expected result or one of a specified list of errors [12]. When handling errors with pattern matching, the Hare compiler requires every possible error type to be addressed, preventing exceptions from unintentionally being left unhandled. Incorporating errors into the type system, like Rust and Hare do, makes it possible to see which errors can occur in a function simply by reading its signature, similar to explicitly specified exceptions.

3.4 Object-oriented principles

Object-oriented programming (OOP) is a programming language paradigm that emphasizes the grouping of related data and behavior into objects, which have defined, compound types called classes. OOP is a familiar pattern to many developers, and many design patterns are expressed in an object-oriented manner. While OOP itself has been criticized for increasing code complexity, it provides a few major improvements to code quality by means of encapsulation and abstraction. Regardless of whether a language fully embraces the OOP philosophy, implementing these aspects can help to improve the maintainability of its programs.

Encapsulation allows developers to restrict the interfaces of their objects, so that only relevant data and behaviors are accessible to external entities. This allows internal implementation details to be hidden, making it easier to see which changes to a class are likely to affect other classes, and which can be more safely modified. Information hiding is a related practice by which implementation-specific data is made private. Access to public data can also be restricted and controlled via the use of accessor functions. All of these features and practices reduce the amount of effort required to refactor an object, which is useful when stability is important, as in the case of maintaining software.

Abstraction similarly helps decouple objects from one another by letting them depend on abstractions rather than concretions. This makes it possible to easily substitute a different concrete object with a compatible interface without rewriting any code. This relates to the concept of polymorphism, in which any class derived from a specified dependency may be used in its place, with no apparent change from the caller's perspective. This makes it possible to change the concrete implementation of a dependency without rewriting the dependent code, which reduces the effort required to refactor such code.

3.5 Namespaces

Namespaces are a way of reducing identifier collisions by restricting the scope of identifiers. These kinds of collisions can otherwise occur quite easily in large software systems with many dependencies. Adding support for namespaces at the language level makes it possible to segregate identifiers into separate, named groups. The `#include` preprocessor directive in C is an example of an import system without namespaces. Most other languages support namespaced imports, including C++, Java, and Python.

It is possible to work around the collision issue by manually adding prefixes to identifier names in place of a namespace. However, this is not a foolproof solution, since collisions can also occur in included library code. Furthermore, prefixing identifiers increases the verbosity of the code.

Another problem that arises when importing external code without using namespaces is that it can be hard to find where an identifier is defined. This can be a challenge at the language level, since a code module cannot be properly parsed without processing all of its imports. This is especially a problem when the language supports macros, as is the case in C. It is also an inconvenience at the editor level, since it can be hard to find where an identifier comes from without intelligent tools.

Languages with namespaces often allow certain members of a namespace to be imported directly, removing the need to use their fully qualified names, and thereby reducing verbosity. For example, this can be done in Java with an `import static` statement, and in Python with a statement of the form `from module import identifier`. Importing members of a namespace this way can cause collisions, but if it does, the programmer always has the option to fall back on the fully qualified name.

These kinds of imports are best done by explicitly listing the members of the namespace to import. Nonetheless, some languages, such as Python, allow for wildcard imports, which import every member of a namespace. This eliminates most of the benefits of namespaces; collisions become possible and annoying to fix, and if multiple wildcard imports are used, then it once again becomes difficult to find where an imported identifier is defined.

Namespaces improve maintainability by making it easier for maintainers to add new internal and external dependencies without worrying about name collisions. They also eliminate the excessive verbosity that results from prefixing identifiers, making the code more concise and more readable. As *The Zen of Python* says, "namespaces are one honking great idea – let's do more of those!"

4 Ecosystem

4.1 Code analysis

Code analysis refers to a variety of techniques that extract properties from source code. In the context of software maintainability, one of the most useful applications of code analysis is to detect potential errors. Detecting potential errors during development allows them to be avoided or fixed prior to deployment, rather than being patched afterwards. Analysis can also be useful during maintenance, where it can detect potential regressions unintentionally introduced by other fixes or changes.

Code analysis comes in two main forms: static analysis and dynamic analysis. Static analysis involves analyzing code without running it, whereas dynamic analysis involves analyzing code by running it. Almost every language implementation does some basic static analysis in the form of lexical analysis (lexing) and syntactic analysis (parsing), and often other analyses such as type checking. Compiled languages generally need to do more static analysis so that they can generate valid code in an intermediate language. Interpreted languages may be able to delay analyzing parts of the code until the interpreter actually needs to execute them.

Being able to detect errors at compile time is preferable to detecting them at runtime, so all languages should consider including additional static analysis beyond what is strictly necessary. Third-party static analysis tools to detect potential bugs are often known as linters, but issues detected by official static analyses can simply be communicated through compiler warnings.

The most obvious form of dynamic analysis is testing, such as unit testing, integration testing, and system testing. A language can assist with testing in various ways, such as by providing a unit testing framework, support for mocking, and ways of measuring code coverage. A debugger is also a form of dynamic analysis, and one which can prove instrumental to maintenance. As with other tools, debuggers can be written by third parties, but language implementers can put their knowledge of the implementation to use by creating and providing an official debugger.

Languages can also provide methods for developers to formally verify their code. At the simplest, these can include compile time and runtime assertions. Slightly more advanced techniques include invariants, function contracts, and bounded verification. Implementers can even consider building an engine to formally prove assertions and properties in given code. Such techniques can theoretically verify a program's functional correctness with a high level of confidence. However, if programmers and maintainers are required to work through the formal verification process, this may result in longer development and maintenance times, rendering such formal methods excessive for non-safety-critical projects. Thus, they should remain optional if included in a general-purpose language.

4.2 Package management

It is rare for useful software to be built entirely from scratch, with no external dependencies. It is often preferable to reuse quality solutions to existing problems when they are suitable and available. However, external dependencies come with their own challenges, one of which is versioning. Because dependencies are typically developed independently by different teams, it often happens that dependencies change in ways that break their dependents. Furthermore, a single developer or maintainer may need to work on multiple projects, each of which may require different versions of the same dependency, or even different versions of the language. While this can be handled by containerization software, containers may require additional overhead, and they may interfere with other development tools and processes. For these reasons, language implementers should consider including a language-native solution for managing, versioning, and sandboxing installed packages.

A package manager needs a repository of packages to draw from. To encourage package developers to contribute their software to this repository, it should be easy for them to publish and update packages. Package repositories for popular languages can grow quite large — for instance, the repository for npm, the Node.js package manager, holds over 2 million packages [13]. Reviewing every package would require a large amount of effort, so it can be tempting to allow any user to freely publish and update packages. However, this is dangerous, because maintainers of commonly used packages can update them to include malicious code. Furthermore, users can upload malicious packages with similar names to popular packages, which programmers could unintentionally install by making a typo. By allowing such exploits to enter the repository, the repository maintainers shift the burden of security from themselves to the maintainers of dependent software.

This is a case where cost and convenience are at odds with security. Nonetheless, repository maintainers should strongly consider implementing security measures to prevent such exploits. To address the problems described above, for example, updates to packages with a certain number of dependents could require manual review. Similarly, newly submitted packages with names similar to popular packages could require manual review before being admitted to the repository.

Inspiration could also be taken from the repositories of various Linux distributions. Many distributions, like Debian, have stable repositories, which have been curated and tested, and an unstable repository, where active development occurs [14]. Security updates make their way to the stable repositories, but other changes remain in the unstable repository until the next stable release. Another model to consider can be found in the repositories of the Arch Linux distribution [15]. This model effectively consists of a curated repository and a user repository, both of which are always kept up to date. The curated repository includes popular and important packages, maintained and reviewed by the Arch Linux maintainers. The user repository is a centralized place for user-submitted packages, which come with no security guarantees. If applied to a programming language package repository, these models could give maintainers more confidence in the security of their packages.

4.3 Documentation generation

Documentation is one of the best ways to explain how software works. High-level written documentation and inline comments are both useful in achieving this objective. One way programming languages can help assist with regards to documentation is via documentation generation. Documentation generation generally works by parsing comments and signatures at the module, class, function, and field level, and compiling them into a readable document. This allows maintainers, among others, to easily reference the API of a software module when making changes, fixing bugs, or just coming to an understanding of its functionality.

To improve the quality of the documentation, documentation generators often specify a particular format for comments to be included in the generated output. This format can also include additional markup syntax that is ignored by the language's parser but parsed by the documentation generator. For instance, JavaDoc and Doxygen are documentation generators that provide syntax for documenting individual function parameters and return values, among other properties [16, 17]. This allows for richer output and improves the usability of the generated documentation.

HTML is a popular and useful format for generated documentation. Hyperlinks make it easier to navigate a hierarchy of classes compared to a static text document, and allow writers of the documentation to provide links to indirectly related components of the software. Furthermore, HTML can be easily deployed on a website for maintainers and API users to reference without needing to compile the documentation themselves. JavaDoc and Doxygen both support HTML output. JavaDoc also allows HTML tags to be used in its documentation comments, which can increase the readability of the generated documentation at the cost of the readability of the documentation comments themselves.

It is possible to leave it up to third parties to create documentation generators and design a syntax for documentation comments. However, this risks the emergence of multiple competing standards and inconsistencies between them. Therefore, creating official specifications and a reference implementation is preferable.

4.4 Standard library

Providing a good standard library can help improve code quality. From a maintenance perspective, there are many factors that contribute to the quality of a standard library. A good standard library should be robust, stable, and backwards-compatible. It should be well-documented and easy to learn. It should be broad enough to encompass at least the main use cases of the language, and ideally more, as long as it can remain well-supported. A standard library also provides an opportunity to exemplify and demonstrate the idioms of the language to developers.

Introducing a third-party library into a software project involves various risks. It may have security vulnerabilities, incompatibilities with other dependencies, or incompatibilities with newer versions of the language. Smaller libraries especially may contain bugs due to a lack of testing, may lack documentation, and may be less likely to receive updates. Due to these concerns, introducing third-party dependencies is likely to increase the amount of maintenance work required relative to a standard library.

Another advantage of a standard library is its ubiquity. Almost all software is likely to use it to some extent, which builds familiarity among developers. If software uses a standard library instead of a third-party library, maintainers will not need to learn a new library to maintain it. If maintainers need to troubleshoot or debug code that relies on the standard library, they are likely to be able to find documentation due to this ubiquity. This includes not only official documentation, but unofficial documentation such as tutorials, examples, and answered questions.

Not every feature should be included in the standard library, though. The larger the standard library, the more maintenance it will require from language implementers. Furthermore, the size of the standard library can affect the size of the runtime environment or compiled program executables. Therefore, language implementers should consider which features are worth implementing in the standard library, considering their size and maintenance cost. Particularly common and important features can be considered for inclusion as built-in features of the language.

4.5 Automatic formatting

As discussed, suggesting a standardized coding style is useful, as it reduces the number of distracting style inconsistencies in code, making it more readable and therefore more maintainable. However, not every developer and organization will go out of their way to follow a standard style, especially if they are already following a different style. Furthermore, even with a style guide in place, it is possible for style inconsistencies to slip through the review process. Going back to fix these style inconsistencies is not an efficient use of developer time, and doing so can lead to unnecessary commits and merge conflicts. Automatic code formatting tools address these problems by making it easy to use, adopt, and consistently enforce a standard style.

After the initial setup, a code formatter requires no additional effort on behalf of the programmer to use. A formatter can also be run on a code base that wishes to adopt a standard style, although this still has the downside of polluting the history of each affected file in the version control system. In both cases, a good code formatter solves the issue of style inconsistencies making their way into the code due to its systematic, algorithmic nature. Always committing with a consistent style is likely to result in smaller diffs in the long term.

Code formatters work well when run upon saving changes to a file, as this allows programmers to immediately see their code formatted in the proper style. It can also tip them off to potential syntax errors if the formatter fails or exhibits unexpected behavior. When using a version control system like Git, running a formatter in a pre-commit hook is another reasonable approach.

Code formatters do not necessarily need to be developed by the creator of the language, but doing so reduces the likelihood of competing formatters arising with potential inconsistencies. Furthermore, as the implementer of the language, one would already have access to and knowledge of their language's parser, which is necessary for non-trivial code formatting. Some practical examples of official formatting tools include `dart format` [18], distributed with the Dart language, and `gofmt` [19], distributed with the Go language. As of 2013, approximately 70% of Go packages surveyed were formatted following gofmt rules [19].

5 Conclusion

Although a programming language does not directly determine the maintainability of the software it is used to create, it has an influence nonetheless. Non-functional aspects, language features, and the language ecosystem all contribute to creating an environment that makes it easier or harder for developers to write maintainable software. Language designers should consider how their design choices at each of these levels may affect software maintainability, for better or for worse.

References

[1] David Cheney. *The Zen of Go*. Feb. 4, 2020. URL: https://the-zen-of-go.netlify.app/ (visited on 03/13/2023).

[2] Rust contributors. *The Rust Design FAQ*. Dec. 3, 2015. URL: https://github.com/rust-lang/rust/blob/87b865c83cdd1c6dd8ac4997744bb701e9f2dcda/src/doc/complement-design-faq.md (visited on 03/13/2023).

[3] Zig contributors. *In-depth Overview — Zig Programming Language*. URL: https://ziglang.org/learn/overview/ (visited on 03/13/2023).

[4] Zig contributors. *Why Zig When There is Already C++, D, and Rust?* URL: https://ziglang.org/learn/why_zig_rust_d_cpp/ (visited on 03/13/2023).

[5] Tim Peters. *PEP 20 — The Zen of Python*. Aug. 19, 2004. URL: https://peps.python.org/pep-0020/ (visited on 03/13/2023).

[6] Zig contributors. *The Zen of Zig*. URL: https://ziglang.org/documentation/master/#toc-Zen (visited on 03/13/2023).

[7] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8 — Style Guide for Python Code*. July 1, 2001. URL: https://peps.python.org/pep-0008/ (visited on 04/05/2023).

[8] Microsoft. *Why does TypeScript exist?* URL: https://www.typescriptlang.org/why-create-typescript (visited on 04/05/2023).

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2023.

[10] Drew DeVault. *Safety features of the Hare programming language*. June 21, 2022. URL: https://harelang.org/blog/2022-06-21-safety-features/ (visited on 03/13/2023).

[11] Dart contributors. *Dart overview*. URL: https://dart.dev/overview (visited on 03/13/2023).

[12] Drew DeVault. *Hare's advances compared to C*. Feb. 9, 2021. URL: https://harelang.org/blog/2021-02-09-hare-advances-on-c/ (visited on 03/13/2023).

[13] npm Inc. *npm*. URL: https://www.npmjs.com/ (visited on 04/05/2023).

[14] SPI et al. *Debian Releases*. Aug. 14, 2021. URL: https://www.debian.org/releases/ (visited on 04/05/2023).

[15] ArchWiki contributors. *Arch Linux*. Feb. 22, 2023. URL: https://wiki.archlinux.org/title/Arch_Linux (visited on 04/09/2023).

[16] Oracle. *javadoc*. URL: `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html` (visited on 04/05/2023).

[17] Doxygen. *Doxygen Manual: Special Commands*. Mar. 29, 2023. URL: `https://www.doxygen.nl/manual/commands.html` (visited on 04/05/2023).

[18] Dart contributors. *dart format*. URL: `https://dart.dev/tools/dart-format` (visited on 04/05/2023).

[19] Andrew Gerrand. *go fmt your code*. Jan. 23, 2013. URL: `https://go.dev/blog/gofmt` (visited on 04/05/2023).