A Gap in the Analysis of Technical Debt in Procedural Languages

An Experiential Report of Go

Grant Nelson
Gianforte School of Computing
Montana State University, Bozeman, USA

Clemente Izurieta
Gianforte School of Computing
Montana State University, Bozeman, USA

***Abstract*—Although the last ten years have seen significant growth in the approaches to managing and measuring technical debt and in the strength of its community, few resources are available today to practitioners in the procedural space. The focus has been vastly geared toward object-oriented paradigms, paying little attention to growing communities in non-object-oriented practices. This is a missed opportunity, because procedural approaches are gaining popularity in both information (IT) and operational technology (OT) environments. This article provides a perspective from one language: Go. We describe the analysis complications faced and the potential for future developments to help practitioners.**

■ THE PROGRAMMING LANGUAGE GO (golang.org) is growing in popularity and becoming widely used. Fullstackacademy.com ranks Go as the fourth best language to learn in 2021 and Geeksforgeeks.org lists Go in the top 10 languages which "will rule in 2021." Go is used by many well known companies including Google, Uber, Workiva, Twitch, Dailymotion, Dropbox, and SoundCloud.

Go is typically used for multi-threaded applications such as servers and desktop software. Its design was influenced by C and Pascal. However, Go has some more modern capabilities such as a garbage collector and memory safety. It is an imperative procedural language with some support for object-oriented designs via duck typing and type embedding. Duck typing is a dynamic approach that binds an object to an interface based on the presence of matching function signatures rather than explicitly defined inheritance. Type embedding is similar to inheritance except the functions can't be overwritten, instead the embedded type's members are accessible from the wrapping object.

The use of many procedural languages such as FORTRAN, C, Pascal, and VB has dwindled since object-oriented languages such as C++, C#, Objective-C, and Java gained popularity. However, procedural languages such as Rust and Go, have become more prevalent during the last 10 years.

As with all software, while a Go project is being developed, Technical Debt (TD) inevitably builds up from developers, intentionally or unintentionally, lowering quality in order to deliver source code by a deadline. The lower quality causes an increase in the delivery time and costs for future maintenance efforts [1].

We have chosen to do an experiential report to aid practitioners who are using or are interested in using Go. Practitioners in the exploratory phases of researching TD analysis tools targeting procedural languages will also benefit. Because a systematic search for TD tools that support Go reveals little, we instead report on concrete experiences trying to manage and measure TD. We chose Go due to its rising popularity and author experience. We synthesize our findings by reporting on three areas we found that limit TD analysis.

This article discusses the current TD analysis, any lacking analysis, and the complications which arise while determining TD in Go with modern TD analysis techniques. It provides insights that our community can not afford to ignore, given new growth predictions in modern (and loved) procedural languages [2]. In section 1 we discuss tools and techniques currently being used to analyze TD in Go. In section 2 we discuss TD analysis which would help identify increased maintenance costs. In section 3 we discuss Go features which makes current TD analysis techniques more difficult to apply to Go. Finally, in section 4 we discuss directions that can be taken to support the practitioner community.

1. Current Technical Debt Analysis

There is an abundance of static and change analysers for Java and other object-oriented languages. Go has several tools for linting (e.g., golint, staticcheck.io), validating (e.g., govet), and formatting (e.g., gofmt, goimports). There are also tools to help manage Go package dependencies (e.g., godep, vender, gomod, spaghetti-cutter), however; few tools exist to help with TD analysis. The majority of them, such as gocyclo and SonarQube, focus on code complexity metrics.

SonarQube exemplifies a typical and highly used TD tool due to its popularity and support for many languages [3]. SonarQube provides 38 built-in rules for statically checking Go code (see **Figure 1**); 29 of those rules are aimed at a type of code smell, different from the more

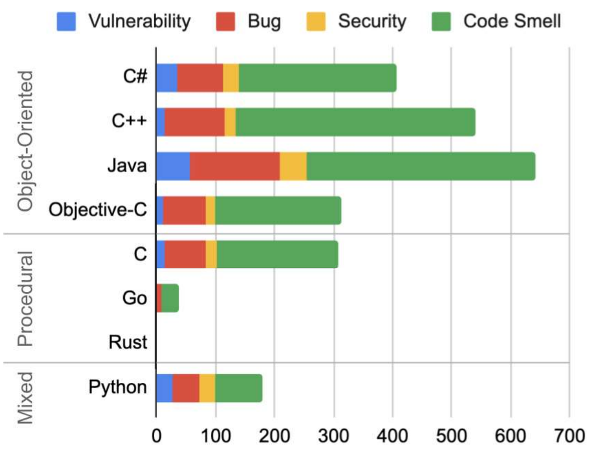


Figure 1. SonarQube rule count by language

commonly known Fowler code smells [4]. Out of these rules, the only TD analysis performed is in the form of Cyclomatic Complexity of single functions. Section 3.1 discusses how Cyclomatic Complexity is used when determining TD issues such as God Objects.

CodeScene provides tools for evaluating Go in a unique way compared to SonarQube. As part of CodeScene's analysis for determining maintenance candidates it calculates an intention-based complexity metric [5]. As with SonarQube, this complexity measurement is also calculated at the function level. CodeScene groups the complexities by file. As further discussed in Section 3.1, in Go, functions with specific receivers can be spread across several files in a package, thus making the complexity of a structure seem lower because complexity is measured locally, when in reality, the maintenance costs of that structure can be much higher.

2. Lacking Technical Debt Analysis

Many reusable design patterns for object-oriented software [6] can be applied to Go to create high quality, maintainable software. Go does not have inheritance; however, using its form of an interface and duck-typing, it is possible to create implementation inheritance. This means that several patterns, such as the composite and decorator patterns, can be used.

The pattern and anti-pattern analysis being used to find decay and grime [7] [8] [9] could be applied to Go if the equivalent objects created by

Go's structures and functions can be determined. Since functions with receivers, see **Figure 2**, may only be defined to receive a structure in the same package, object discovery isn't as complicated as it would be in other procedural languages.

Go may have patterns not yet seen in object-oriented patterns. Go has some multithreading language features not seen in many other languages. It provides goroutines, channels, and channel selectors. These can be used to quickly build dynamic multithreaded applications when used correctly. They can also be used poorly, making maintenance and development time cost more. Research could be expanded to understand designs that can benefit from patterns, and also those that can increase the TD of the system.

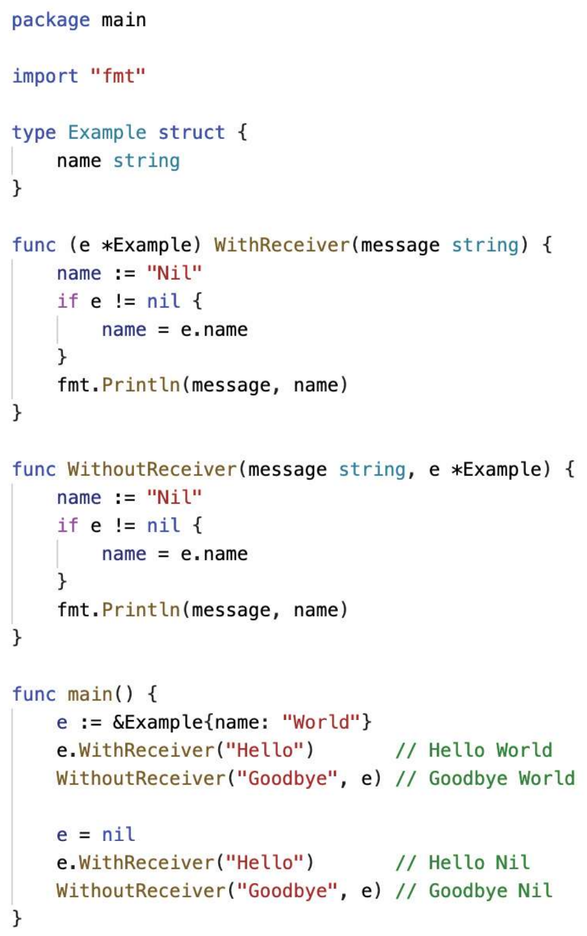
All existing TD analysis and any TD analysis for Go's unique qualities should be used when creating Go projects to retain high quality code. CI tools to perform this analysis would be greatly beneficial to the practitioner communities and their respective organizations.

3. Analysis Complications

The following sections describe three areas where current TD tools are hindered by procedural constructs. We use Go to exemplify such problems. Beyond the three areas discussed in this section, procedural languages have other complications such as functions embedded inside of functions, increased usage of closures and anonymous functions, and type definition for types other than structures and interfaces. Takeaways from this section are meant for practitioners and researchers working towards building the TD analysis tools and to explain why analysis tools for object-oriented languages cannot simply be used as-is for procedural languages. We must include developers that understand these procedural concerns to help evolve current TD analysis tools.

3.1. Finding Objects For Patterns

As discussed in section 2, to apply some of the current TD analysis to Go, tools would have to find equivalent objects in the source code to those in design patterns. This can be complicated since the tools may not be aware of which Go structures are intended to be used in place of some interfaces. Structures that are duck-typed do not explicitly specify the interfaces they intend to



```go
package main

import "fmt"

type Example struct {
    name string
}

func (e *Example) WithReceiver(message string) {
    name := "Nil"
    if e != nil {
        name = e.name
    }
    fmt.Println(message, name)
}

func WithoutReceiver(message string, e *Example) {
    name := "Nil"
    if e != nil {
        name = e.name
    }
    fmt.Println(message, name)
}

func main() {
    e := &Example{name: "World"}
    e.WithReceiver("Hello")       // Hello World
    WithoutReceiver("Goodbye", e) // Goodbye World

    e = nil
    e.WithReceiver("Hello")       // Hello Nil
    WithoutReceiver("Goodbye", e) // Goodbye Nil
}
```

Figure 2. Receivers in Go

implement.

In some cases, TD tools find that a structure is implementing an interface because it duck-types to that interface. However, the developers may have had no intention of implementing that interface and the tool's finding occurs by coincidence. It is common to have multiple packages define identical interfaces. For example, two packages with a "writer" interface which have a single function, *write(string)*.

It is difficult for a TD tool to know if the interfaces are part of multiple separate designs with their own patterns, or part of a single design and pattern. Developers will sometimes create multiple identical interfaces in different packages and rely on duck-typing to help prevent dependency cycles between those packages.

Developers may also choose to not use a receiver on a function when writing a function specifically for a structure. In Figure 2 the function *WithReceiver* is part of the *Example* structure. However, some developers could use a function, similar to *WithoutReceiver*, for functions which are still intended to be specifically designed for the *Example* structure but don't appear to be. The two example functions are functionally the same, the only difference is in how they are invoked. This means that some functions may not be available as a method to an object if the tool only uses receivers to indicate membership. Since interfaces may not be used as a receiver in a function, it is common for developers to not use a receiver.

When part of the object's whole picture is missing, some analytics such as total complexity may not be calculated correctly. Take for example a God Object. A God Object is a structure junk-drawer which is too big and does too much, and is hard to maintain but easy to create. When determining a God Object the Weighted Method Count (WMC) is used [10]. The WMC is the sum of the method complexities for an object. In a procedural language an equivalent overly complicated structure could be created without ever using a function with a receiver. This makes it difficult to determine which functions belong to that structure and calculate the correct WMC.

In the cases where receivers aren't being used, even code change frequency, as used in CodeScene, may have difficulty detecting a God Object. The structure may be distributed across many files and even many packages.

3.2. Encapsulation Not Encouraged

Go does not provide ways to use custom iterators in a foreach-statement so it is up to the developer to create map methods, their own iterator, or expose the underlying map or slice (slice is a type of array). In many cases the developer will do the easiest thing, exposing the underlying data, thus incurring TD. This can also make it difficult to protect the data and programmatically determine if this is a TD item. The developer may pass around a slice of components, instead of taking the time to encapsulate the data into a pattern, such as composite.

This can even lead to a dichotomy. Proper encapsulation would encourage good practices and slow the growth of TD in the long run. However, some developers feel the encapsulation is difficult to use and extend because they cannot use constructs like the foreach-statement. This also means that each form of encapsulation requires the developer to learn how the designer of that object intended it to be used. This is counter to object-oriented languages, such as C# or Java, which simply provides an interface to implement as part of the encapsulation, and therefore is quickly learned and can be used in language specific features.

3.3. Less Reusable Functionally

Go currently does not support parameterized types (e.g. generics or templates). Many projects will need to have more complicated data structures or specialized objects to be optimized for speed and memory. Without parameterized types developers have three options.

1) The data structure uses a functionless interface, *interface*{}, which is similar to Java's *Object*. However, this consumes extra memory and takes extra time to cast. Casting to and from the interfaces can also complicate code using the data structure. Lastly, it doesn't allow for compile time type checking.
2) The developers can write a custom data structure for each type. This creates nearly identical code. This gives multiple places which need to be updated when a bug is found to keep all implementations maintained. However, this gives the best performance and is easier for developers to create.
3) There are tools similar to Mustache or C macros (e.g. gomacro) which take a Go file with special symbols in it that get replaced prior to compilation. This generates code with predefined types. This writes fast code and makes only one spot to maintain but the added step makes CI more complicated and some of the other Go tooling can't cope with the special symbols.

This complication may be elevated in future versions of Go. Since developers typically choose option 2 and create duplicate code, the community has argued that providing functionality to support this requirement is of high priority and the solution needs to be easy to maintain, reusable, fast, and memory efficient [11]. The implications and consequences of duplicate code lower maintainability, increase technical debt (resources and costs), and potentially impact time to market.

4. Minding the Gap

Practitioners who have chosen or are thinking about choosing Go or other procedural languages for their projects can benefit from the takeaways presented in this report.

TD analysis tools aid practitioners by providing metrics that can be used to determine which parts of the project require additional effort to maintain. In the absence of these tools for procedural languages (i.e., Go), organizations will have to manage TD in the same way that TD was managed prior to the emergence of these tools. Teams should be provided with the time to do maintenance and refactor as needed, and must rely on developers to make the best judgements regarding quality tradeoffs [12]. This is difficult because it requires developers to agree on which tasks are important, and management to recognize this importance even if the part of the code is not functionally wrong nor customer facing. TD items need to be treated as first class citizens in team backlogs.

When Go releases a version with parameterized types it will be important for developers of a project to have the time to refactor existing code and take advantage of this feature. The TD that has accumulated by duplicating code or by not using compile time type checking will need to be paid down.

Due to the rising popularity and usefulness of new procedural languages, the community is adopting them in larger and more complex systems. These new procedural languages can produce high quality code. However, our community needs to be aware of the gap in TD analysis, and current tooling is not enough. Experienced procedural engineers are required to help expand the corpus of TD rules necessary to analyse these languages.

Practitioners using modern procedural languages should stay aware of new TD analysis tools that will be released and plan accordingly for necessary refactors. The TD research community needs to help bridge the gap with relevant studies that address new modern procedural languages that TD tool developers can adopt, hopefully sooner than in another ten years.

REFERENCES

1. Clemente Izurieta, Ipek Ozkaya, Carolyn Seaman, Philippe Kruchten, Robert Nord, Will Snipes, and Paris Avgeriou. Perspectives on managing technical debt: A transition point and roadmap from dagstuhl. In *International Workshop on Technical Debt Analytics (TDA)*, Hamilton, New Zealand, 2016.
2. Kamaruzzaman. Available December 23, 2019. https://towardsdatascience.com/top-7-modern-programming-language-to-learn-now-156863bd1eec
3. Paris Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Sas, Saulo de Toledo, and Angeliki Tsintzira, An Overview and Comparison of Technical Debt Measurement Tools, In *IEEE Software, vol. 38, no. 3*, pages 61-71, May-June 2021, doi: 10.1109/MS.2020.3024958.
4. Valentina Lenarduzzi, Franscesco Lomio, Heikki Huttunen, and Davide Taibi. Are SonarQube rules inducing bugs? *arXiv:1907.00376v2 [cs.SE]*, 2019.
5. Adam Tornhill. Prioritize technical debt in large-scale systems using CodeScene. In *2018 ACM/IEEE International Conference on Technical Debt*, 2018.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson/Addison-Wesley, 1995. ISBN 978-0-201-63361-0.
7. Derek Reimanis and Clemente Izurieta. Behavioral evolution of design patterns: Understanding software reuse through the evolution of pattern behavior. *Peng X., Ampatzoglou A., Bhowmik T. (eds) Reuse in the Big Data Era.* ICSR 2019, 11602, 2019.
8. Clemente Izurieta and James M. Bieman. How Software Designs Decay: A Pilot Study of Pattern Evolution. *1st ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, Madrid, Spain, September 2007.
9. Daniel Feitosa, Paris Avgeriou, Apostolos Ampatzoglou, Elisa Y. Nakagawa. (2017) The Evolution of Design Pattern Grime: An Industrial Case Study. In *Felderer M. et al. (eds) Product-Focused Software Process Improvement.* PROFES 2017. Lecture Notes in Computer Science, vol 10611. Springer, Cham. doi: 10.1007/978-3-319-69926-4_13

10. Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE international Conference on Software Maintenance,* pages 350–359, ICSM. IEEE Computer Society, Washington, DC, 2004.
11. Ian Lance Taylor. Available January 12, 2021. https://blog.golang.org/generics-proposal
12. Zadia Codabux and Byron Williams, Managing technical debt: An industrial case study *4th International Workshop on Managing Technical Debt (MTD),* 2013, pages 8-15, doi: 10.1109/MTD.2013.6608672.