ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/335712808

Large Scale Unit Testing for Go Programming Language Packages

Research · September 2019

DOI: 10.13140/RG.2.2.36308.76166

CITATIONS
0

READS
3

2 authors:

Kean Ho Chew
ZORALab Enterprise, Puchong, Malaysia

7 PUBLICATIONS **1** CITATION

SEE PROFILE

Lee Booi Lim

9 PUBLICATIONS **36** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project IT and Software Technologies View project

All content following this page was uploaded by Kean Ho Chew on 10 September 2019.

The user has requested enhancement of the downloaded file.

Large Scale Unit Testing for Go Programming Language Packages

Chew, Kean Ho[1]; Lim, Lee Booi[2]

[1]*ZORALab Enterprise*
kean.ho.chew@zoralab.com
[2]*Independent*
lee.booi.lim@gmail.com

July 2019, 1st Issues

Abstract

Testing in Go is reasonably easy compared to all other programming languages due to its out of the box tools availability and robustness. Moreover, there are many tutorials made available for beginners to learn and practice.

When at scaled, such as growing beyond 1000 test cases, the existing test approaches introduced by the tutorials become too complicated, causing a huge refactoring efforts like 2-3 days overhauling efforts. Hence, there is a need for a new large scale test approach for unit testing Go packages.

This paper first introduces Go programming language. Then, it proceeds to present the current trends related to Go testing practices and its approaches. After that, the paper presents all encountered problems while using the existing test approaches at scale. The root causes of the problems are identified and mitigation actions are determined.

Lastly, the paper proposes a large scale test approach based on the learning and mitigation actions. The approach is then discussed and concluded.

1. Introduction

Testing in Go programming language is reasonably easy to develop due to the design nature of the language. Go itself is specifically designed for programmers to be more productive by being expressivess, concise, clean, and efficient as both human and machine understandable language[1]. Hence, there is no exception to its unit testing facility. However, a lot of Go testing tutorials and guidelines are useful at the beginning when the package is small; they are cumbersome and heavy when scaled. Normally, this problem happens when there is a test requirement to have the package to be tested thoroughly, like building a stable security module. Considering writing a cryptography wrapper library, the issue arises when the total test cases easily grown to >1000 units test cases.

This paper started off by introducing the Go programming language itself and reviewing current test methodologies in Go. Then, the paper listed all the encountered problems while running the test suite at scale. These problems were analyzed for their causes and determines the necessary mitigation actions. With the mitigation actions determined, The paper then proposes the large scale test approach alongside explaining its design mechanics. The approach was discussed and concluded.

2. Overview

In this section, this paper provided the necessary information related to Go programming language, its current test methodologies and approaches for Go packages.

Then, the paper explained the problem encountered during large scale testing while using the specified approaches.

2.1. Go Programming Language

Go programming language was developed by Google since 2007 to create dependendable and efficient software[1][2]. It is designed based on analyzing pros and cons of various programming languages x to resolve software engineering issues and to provide an alternative to C++[2]. Among the competitor languages analyzed are C[1][2], C++[1][2], C#[3], Java[2], Python[3], Javascript[3], Swift[3], and Haskell[3].

The most notable benefit for using Go is that Go has the best portability among all competitors[2]. It is done via resolving dependencies and closely compiled into a single, self-contained executable binary file at build time[4]. Compared to its competitors with the like of Python, Ruby, and Javascript, the compiled Go program simply runs without needing to setup the language interpreter[5]. As compared to Java, it eliminated the need of virtual machine, further boosting the program's performance[5]. As a result, Go fully fulfilled for the "Write Once, Run Anywhere (WORA)" design mantra in software programming language from start[5].

The second notable benefit is that it can perform concurrency processing easily without heavy library and simple to understand[2]. Its goroutine and channel functionalities are lightweight, consuming almost 2 kilobytes of heap memory compared to 1 megabyte size when using Java programming language[6].

The third benefit is the well-prepared development tools made available for Go programmers out of the box[2]. Go standard development package comes with a formatting tool known as "Gofmt" that can format the style of codes to a single style standard automatically, making it easier for all Go programmers; "Go get" quickly sources and manages Go dependencies from Git version control system (GVCS); "Go doc" readily generates documentation directly from the source code itself[2]. Hence, the programmer only needs to focus on working on source codes regardless of what level of proficiency.

Although the programming language is relatively young compared to all of its competitors, it did overcome some of the shortcomings of the other languages. Therefore, it is worth pursuing in-depth research with Go programming language giving that it provided Go programmers a lot of benefits out of the box.

2.2. Testing in Go

Like any other languages, testing in Go offers the conventional test methodologies like[8]:

1. Static Analysis
2. Unit Testing
3. Test Coverage
    a. Standard Node CFG
    b. Edge CFG
    c. Condition CFG
    d. Boundary Value Analysis

For static analysis, Go communities built a large set of various static analysis linters grouped under an umbrella Go package known as "golangci-lint"[8]. This Go linter program is an independent executable program meant to perform all forms of static analysis not limited to standard cyclomatic complexity scan, duplicated codes scan, critics, global constant and variables scan, style, security, etc[8]. With some tweaking to bundle the command lines under a single command or text editor macros, one can run static analysis easily and automatically after saving the source codes[9].

Go heavily depends on unit testing to the point of providing robust and flexible tools for it[9]. Due to the flexibility nature of the Go programming language[2][3], any developer can implement various types of unit testing approaches[7]. Moreover, Go even provide a test coverage heat-mapping tool, allowing one to view which part of codes are not tested or tested intensively. Figure 2.2-1 shown a snapshot of such heatmap. With all these tools available for programmer, one can proceed to perform effective testing and code refactoring without wasting resources over redundant test cases.

Go also provides performance benchmarking tool in its testing package, allowing the programmer to benchmark statistically across different executions[10]. This tool allows programmer to view and analyze execution flow and path to understand bottlenecks and weaknesses[10]. However, Go developers are advised to focus on simplicity, readability, and productivity first instead of performance or concurrency[11]. Hence, the use of benchmarking tool should be done sparingly and sensibility. Figure 2.2-2 shown a snapshot of the benchmark statistic outputs while Figure 2.2-3 shown the execution flow chart with statistical timing.

In this paper, the testing focuses on some static analysis tools and unit testing components only while leaving benchmark and its associated tools outside of research scope. This is to maintain the research focus to present large scale unit testing for Go packages, not about an overall test methodologies in Go programming language.

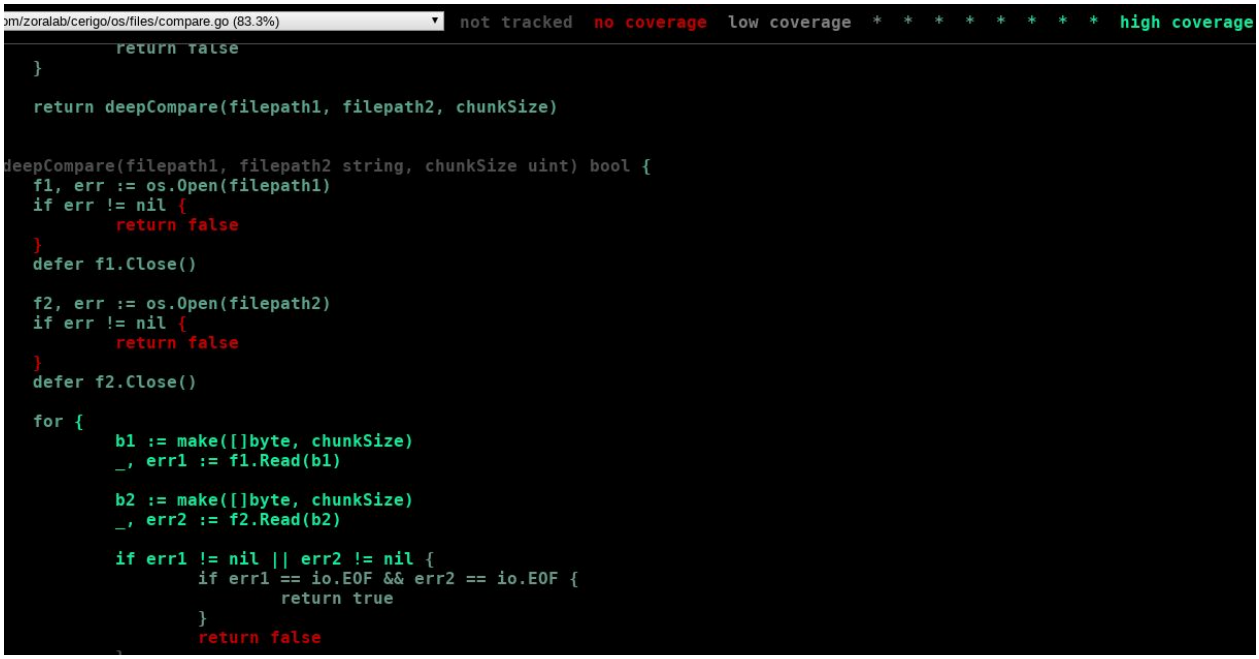

Figure 2.2-1 - the test coverage heatmap tool

```
benchmark                          old ns/op     new ns/op     delta
BenchmarkAlgoSearchShort-8         12.9          12.9          +0.00%
BenchmarkAlgoSearchMedium-8        33.0          32.5          -1.52%
BenchmarkAlgoSearchLong-8          34.1          32.5          -4.69%
BenchmarkAlgoX2SearchShort-8       270           255           -5.56%
BenchmarkAlgoX2SearchMedium-8      104180        100286        -3.74%
BenchmarkAlgoX2SearchLong-8        111012        104952        -5.46%
BenchmarkAlgo2SearchShort-8        154           152           -1.30%
BenchmarkAlgo2SearchMedium-8       215           215           +0.00%
BenchmarkAlgo2SearchLong-8         216           215           -0.46%

benchmark                          old allocs     new allocs     delta
BenchmarkAlgoSearchShort-8         0              0              +0.00%
BenchmarkAlgoSearchMedium-8        0              0              +0.00%
BenchmarkAlgoSearchLong-8          0              0              +0.00%
BenchmarkAlgoX2SearchShort-8       3              3              +0.00%
BenchmarkAlgoX2SearchMedium-8      16             16             +0.00%
BenchmarkAlgoX2SearchLong-8        16             16             +0.00%
BenchmarkAlgo2SearchShort-8        0              0              +0.00%
BenchmarkAlgo2SearchMedium-8       0              0              +0.00%
BenchmarkAlgo2SearchLong-8         0              0              +0.00%

benchmark                          old bytes     new bytes     delta
BenchmarkAlgoSearchShort-8         0             0             +0.00%
BenchmarkAlgoSearchMedium-8        0             0             +0.00%
BenchmarkAlgoSearchLong-8          0             0             +0.00%
BenchmarkAlgoX2SearchShort-8       176           176           +0.00%
BenchmarkAlgoX2SearchMedium-8      352243        352243        +0.00%
BenchmarkAlgoX2SearchLong-8        352243        352243        +0.00%
BenchmarkAlgo2SearchShort-8        0             0             +0.00%
BenchmarkAlgo2SearchMedium-8       0             0             +0.00%
BenchmarkAlgo2SearchLong-8         0             0             +0.00%
```

Figure 2.2-2 the benchmark statistics outputs

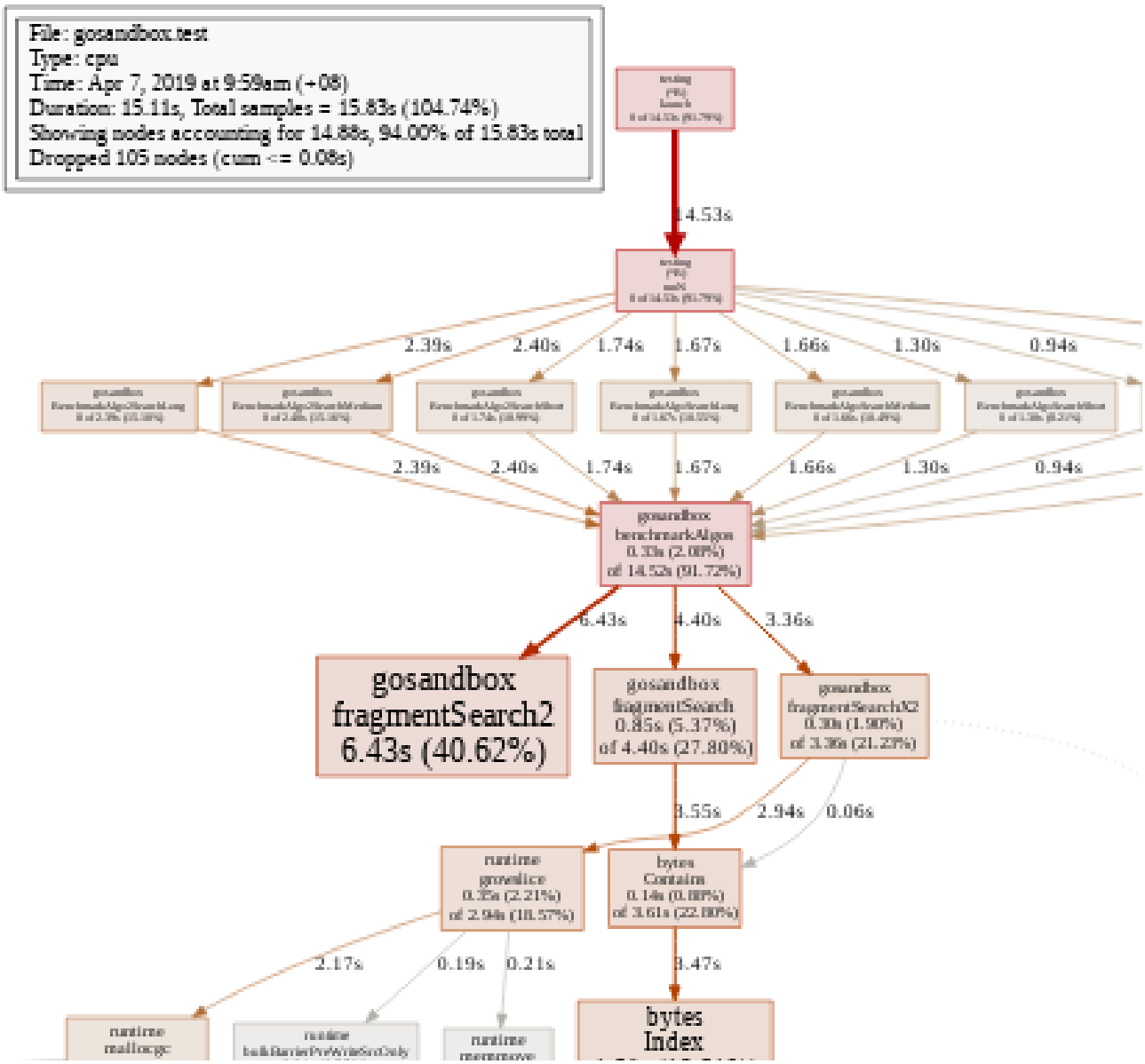

Figure 2.2-3 - the benchmark execution path map

2.3. Test Approaches

In this section, the paper reviewed numerous approaches and recommendations used by various Go programmers and official site. This gives us a clear understanding about the current practices for writing test codes in Go programming languages.

The section covers a minimum of 2 important aspects: the file structures, the testing approach.

2.3.1. Basic Testing Approach

The very basic unit testing approach would be direct data comparison or direct data assertion[12]. The idea is to:

1. Write a test function
2. Execute the subject and get its output
3. Compare the output directly with the desired output.

Figure 2.3.1-1 shown a simple example for `Add(...)` function. This function, acting as test subject, is being called inside the `TestAdd(t *testing.T)` test function. The output of a fixed value where `a = 2, b = 2` is directly compared to the output value (`4`). If the output is not equal, the test function raises an error assertion.

```
func TestAdd(t *testing.T) {

    if Add(2, 2) != 4 {

          t.Errorf("Expected 2 + 2
to equal 4")

    }

}
```

Figure 2.3.1-1 - a basic unit testing approach for testing `Add(a, b)` function

The file structure, as usual, following the standard recommendation where the test codes are parked under a seperate `source_test.go` file for `source.go` source code[12][13]. Figure 2.3.1-2 shows the source code structure for `Add(a, b)` function.

```
directory
├── add.go
└── add_test.go
```

Figure 2.3.1-2 - file structure for hosting source codes and test codes separately.

This is highly suitable for beginners and tester to write simple assertion functions for simple Go implementations. Additionally, for simple source codes that requires 1 or 2 changes like “Hello World” application, this basic approach should suffice.

However, if the requirement expands like having a bigger boundary value analysis coverage[7], this basic test approach is not scalable since the test parameters are hard-coded into the test code. From Figure 2.3.2-1, if we want to know whether `Add(a, b)` function is able to handle negative addition, tester needs to create a duplicate test function with different test parameters, yielding the total test codes in Figure 2.3.2-3.

Although it is simple to understand, it is hard to maintain due to duplications. Example, what if the requirement is to rename `Add` to `Sum`, while having more than 1000 repeated test cases, it would be scary to review all the changes after a simple “find and replace” editing command.

This is where the table-driven test approach kicks in to facilitate such changes.

```
func TestAdd(t *testing.T) {

    if Add(2, 2) != 4 {

          t.Errorf("Expected 2 + 2 to equal
4")

    }

}



func TestNegativeAdd(t *testing.T) {

    if Add(2, -2) != 0 {

            t.Errorf("Expected 2 + (-2) to
equal 0")

    }

    if Add(-2, 2) != 0 {

            t.Errorf("Expected (-2) + 2 to
equal 0")

    }

}
```

Figure 2.3.1-3 - expanded basic unit testing approach for testing `Add(a, b)` function

2.3.2. Table-Driven Test Approach

Table-driven test approach employs a table test structure to manage different parameters for a test case execution (can be known as test algorithm). This approach isolates the chaotic test parameters away from test algorithms, allowing testers to easily add new test cases by manipulating the parameters without changing or altering the test algorithm[12][13][14].

If we refactor the `Add(a, b)` function from Figure 2.3.1-3 using table-driven test approach, Figure 2.3.2-1 shown the new set of codes inside `add_test.go` without introducing any alteration to the existing file structures.

```
func TestAdd(t *testing.T) {
    scenarios := []struct {
        a int
        b int
        o int // output
    } {
        {2, 2, 4},
        {-2, 2, 0},
        {2, -2, 0},
    }

    // test algorithms
    for _, s := range scenarios {
        x := Add(s.a, s.b)
        log(t, s.a, s.b, s.o, x)
        assertError(t, x, s.o)
    }
}

func log(t *testing.T,
    a int,
    b int,
    o int,
    x int) {
    t.Logf(`a=%v b=%v o=%v x=%v`,
        a, b, o, x)
}

func (t *testing.T,
    x int,
    output int) {
    if x != o {
        t.Errorf("bad results.")
    }
}
```

Figure 2.3.2-1 - table-driven approach for testing `Add(a, b)` function

The parameters are tabulated under a temporarily declared `scenarios` structure, holding all the previous parameters and expects output as guided[12][13][14]. The test algorithms are refactored into 2 test functions: `log(...)`, `assertError(...)`, making it isolated from the test case parameters. When tester needs to add new boundary value analysis, say handling 0, or overflow, the tester would only needs to increase the scenario test cases without altering the test algorithm itself.

The test developer can also declares the scenario structure privately outside the test function, making it reusable across various test functions. This is useful for recycling the scenario structures, making the test codes shorter.

This table-driven test approach is the preferred and highly recommended test approach for all unit testing in Go packages due to its scalability in terms of parameters increment and test algorithm isolation[12][13][14].

2.3.3. Time Sensitive Test Approach

Time sensitive test approach is a unique test approach related to execution timing, not limited to just output assertion. Unlike benchmarking which is meant for comparison and competitiveness, this time measurement test approach is for setting a function executes within the expected time duration and time limits.

A good use case would be having a function to prevent side-channel timing attack in secret data comparison like password comparison[15][16]. The timing for any outcome from the secret data comparison (be in failed, system error, system unavailability, passed, hardware caching, etc.) should be executed within the same timing responses regardlessly. This is to prevent attackers from guessing the secret data by observing the execution timing behavior[15][16].

For testing time sensitive function in Go, instead of setting the system to execute test in parallel that can introduce more complications like guessing failure cause[17], it is preferable to opt for concurrent timestamping with timeout ranges. Figure 2.3.3-1 illustrates an example of measuring and testing runtime.

Using timestamping with timeout ranges, the approach simplifies the test algorithms and dependency on complex hardware or testing on various hardwares. Since we cannot measure time with 100% accuracy during runtime, using an acceptable range test approach is the only available option for the tester. Hence, this test approach is rarely seen in most of the guides.

```
func TestAdd(t *testing.T) {

    start := time.Now()

    _ := Add(2, 2)

    stop := time.Now()

    duration := stop.Sub(start)


    select {

        case duration:

            // do something pass

        case duration + (400 * time.Millisecond):

            // do something pass

            // within range

        default:

            t.Errorf("failed timing - %v", duration)

            // handle failed cases

    }

}
```

Figure 2.3.3-1 - a basic unit testing approach for testing `Add(a, b)` function

3. Encountered Problems

In this section, the paper lists out all the problems encountered when using the introduced test approaches at scaled. These problems were observed from some past projects' developments, especially when developing Go packages with strict requirements such as cryptography wrapper.

3.1. Unpredictable Architectural Changes for Test Codes

The most painful problem is managing many architectural changes of test codes in Go. These changes come in different forms like test algorithm changes, test parameters' data structure changes, chaotic requirements as an overall input changes, and test environment changes. Further details and mitigation actions will be discussed in section 5.

These architectural changes are interrelated. Thus, by adapting to these changes, the test suite is constantly under refactoring at an overhaul level. At scaled, this overhaul takes days to complete, which can be a maintenance nightmare.

Also, these changes are unpredictable throughout the early stages of development like prototyping, especially conforming to many Agile software development lifecycles. Go facilitates early stage testing to make sure the created prototype is testable while making design decisions. Since the changes are unpredictable, it is not easy to facilitate a future-proof architecture design for the test suite.

3.2. Wrong Test Approaches for Large Scale Testing

When tester chosen the wrong test approach at the start, for scaling, it is a big problem. This includes making bad decisions such as not using table-driven test approach, improper or no planning for test data structure, no proper planning for mitigating future changes like architectural changes mentioned earlier, and no proper file structure management. Consequently, the test suite becomes unmaintainable at scaled, which often requires either overhaul refactoring or redo from scratch. A good case study is shown in Figure 2.3.1-3 where a test approach was chosen wrongly. At scaled of 1000 cases, that would be a heavy problem.

Another case study is shown in Figure 3.2-1. This is the result of improper planning for data structure, which yielded a long data declaration per test cases. At scaled to 1355 total cases, the test data definition itself yielded a total of 48229 total lines of codes.

```
inBadKeyHasMalformedUpdatedTime: false,
inBadSymmetricKey:               false,
inBadAsymmetricRxKey:            false,
inBadAsymmetricTxKey:            false,
inBadSignatureKey:               false,
inBadAuthenticationKey:          false,
inMissingSymmetricKey:           true,
inMissingAsymmetricRxKey:        true,
inMissingAsymmetricTxKey:        true,
inMissingSignatureKey:           true,
inMissingAuthenticationKey:      false,
inLimitedUseSymmetricKey:        false,
inLimitedUseAsymmetricRxKey:     false,
inLimitedUseAsymmetricTxKey:     false,
inLimitedUseSignatureKey:        false,
inLimitedUseAuthenticationKey:   false,
inAssociatedKey:                 false,
inBadPEMTitle:                   false,
inMissingPEMTitle:               false,
inInvalidPEMPassphrase:          false,
inBadPEMPassphrase:              false,
inMissingPEMPassphrase:          false,
inBadPEMData:                    false,
inMissingPEMData:                false,
inEncryptedPEMData:              false,
inUnrelatedValidPEMData:         false,
inUnrelatedInvalidPEMData:       false,
inRelatedInvalidPlainPEMData:    false,
outDoesNotUseCount:              true,
outData:                         false,
outData2:                        false,
outError:                        false,

                                 48229,1
```

Figure 3.2-1 - Total test cases with 1355 total cases (off screen) and 48229 total lines of codes just to build the test table scenarios.

3.3. Long Test Codes When Scaled

Long test codes at scaled using the introduced test approaches is also a problem. Moreover, there was no guidance for file structure planning and usually have all the test codes squeezed under a single file. This makes the test suite very difficult to maintain since a small change can be snowballed into huge efforts. If the test suite scaled beyond control, it becomes unmaintainable.

A good case study is by observing Figure 3.2-1 and Figure 3.2-2, the optimized version of Figure 3.2-1. Both consumed 23746 to 48229 lines of codes for 1355 total cases. Even in optimized condition, it is still considerably long which calls for attention for careful handling.

```
·>-------inReceiverLabel:      goodRxLabel,
·>-------inError:              errorObject,
·>-------inAction:             nonAsymmetricPrepare,
·>-------inVerdict:            goodVerdict,
·>-------inPublicKey:          goodKey,
·>-------inPrivateKey:         goodKey,
·>-------inLimit:              nil,
·>-------switches: map[string]bool{
·>-------->-------useTestCipher: true,
·>-------},
·>-------outDoesNotUseCount: true,
·>-------outData:            false,
·>-------outData2:           false,
·>-------outError:           true,
·}, {
·>-------inID:                 1357,
·>-------inTestType:           testKeyringExportKey,
·>-------inKeyFunctionID:      goodKeyFunctionID,
·>-------inExportType:         EncryptedPublicKey,
·>-------inCipherID:           goodCipherID,
·>-------inCiphertextOrDigest: goodCiphertext,
·>-------inPlaintext:          goodPlaintext,
·>-------inSenderLabel:        goodTxLabel,
·>-------inReceiverLabel:      goodRxLabel,
·>-------inError:              errorObject,
·>-------inAction:             nonAsymmetricPrepare,
·>-------inVerdict:            goodVerdict,
·>-------inPublicKey:          goodKey,
·>-------inPrivateKey:         goodKey,
·>-------inLimit:              nil,
·>-------switches: map[string]bool{
·>-------->-------useTestCipher: true,
·>-------},
·>-------outDoesNotUseCount: true,
·>-------outData:            false,
·>-------outData2:           false,
·>-------outError:           true,
·},

ios_test.go" 23746L, 774299C                    23746,1
```

Figure 3.3-1 - Total test cases with 1357 total cases and 23746 total lines of codes to define test cases.

3.4. Missing Assertion and Limited Logging Functionality

Another problem is that Go's standard package only provides some primitive tools like minimum logging mechanism and no assertion functions for testing. The main reason is because of Go being a static type design. Go allows developers to spin various permutations of standard data types, but the assertion functions in the standard package are specific to a single rule or single data type. Therefore, tester needs to use 3rd party test package to workaround this problem.

When a tester uses Go's `interface{}` to make a generic input function, it introduces additional complications and defeat Go's clarity principle. Therefore, there is no easy, uniform, and clean way to log information and assert statement without building the package's own assertion.

3.5. Frequent Naming Collision

While complying to Effective Go naming convention practices, a problem found is frequent naming collision happening across source codes and test codes[18]. From Effective Go, it is mandatory to organize the naming pattern to be short and concise, and intuitively understandable, usually consisting of 1-3 short words[18]. Also, both source codes and test codes share the same naming pool in the package which means every content in a Go package is organized and references by a unique name.

When the tester is using these names without consideration, it disrupts the developer future development due to names being "magically" taken out by test codes. This results in wasting memory, unnecessary names for testing alone, unnecessary refactoring efforts due to renaming.

3.6. Infrastructure Differences

Different results are produced due to different infrastructures at development and testing stages is another notable problem. This is seen when using time-sensitive test approach while testing time-related Go packages. From developer's perspective, the test results passed via their hardware like their development laptop. However, upon reaching tester, the continuous integration infrastructure produced negative results with the same source codes due to the use of virtual machines. A good case study would be shifting from a hardware test machine to a Docker container, where a lot of local security restrictions are introduced by Docker itself. Another case is that the tester runs the test on actual hardware while leaving continuous integration testing in a virtual machine.

To have consistent results, the team has to make the continuous integration infrastructure built within their development hardware, ensuring that the source codes not only works in development machine but also in production infrastructure, which is duplicated efforts.

4. Potential Causes to Problems

In this section, the paper lists out all the causes from the aforementioned problems. These causes are analyzed so that a better mitigation can be planned and proposed. The primary objective is to find a consistent way to manage these causes in order to mitigate the problems effectively while seeking reasonable maintainability, readability, scalability, and affordable maintenance costs.

4.1. Rapid and Chaotic Requirement Changes

Rapid and chaotic requirement changes may affect both source codes and test codes respectively. These changes can be addition, removal, and alteration an inconsistent amount of source codes per changes. The changes can happen either systematically or chaotic depending various factors. Low morale talents tend to have chaotic changes since the focus is no longer on product development. Another case would be different product development stages where prototyping stages tends to have chaotic changes for market-fit requirements. Also, different development lifecycle (SDLC) such as cowboy SDLC can have chaotic requirement changes while systematic SDLC like agile or waterfall development lifecycles tends to introduce systematic changes.

Since the test codes are developed based on the requirements and source codes, any changes to the source codes may trigger unpredictable architectural changes to the test codes or breaking them. Hence, the objective is to identify all the encountered changes and isolate them from one another, providing a consistent and stable way to manage these changes.

4.2. Insufficient Experience

The technical debt and the experience of a tester with both testing in Go and the programming language itself can be the primary cause for some problems discussed in section 3. This is commonly seen since Go programming language is new, many developers and testers are currently transition from other languages to use Go.

Without sufficient experience, tester tends to create a number of problems like unable to deal with unpredictable architectural changes, selecting the wrong test approaches for large scale testing, frequently creates naming collision for developers, and creating infrastructure differences.

Hence, another mitigation objective is to facilitate a learning environment for inexperienced tester without compromising test suite scaling quality. This also includes finding a recoverable way to refactor rescuable test suite. Another objective is to have the tester practice the compliant consistency in naming convention and coding pattern.

4.3. Missing Guidelines or Tutorials for Large Scale Testing

The lack of guidance and tutorials covering large scale testing aspects is another problematic cause[12][13][14]. Most guides and tutorials only mentions about known approaches and their how-to but none actually mention about the scaling and the encountered problems mentioned earlier.

Without proper guidance and necessary experiences, inexperience tester tends to create a lot of problems, especially creating unmaintainable long test codes, frequently creating naming collision problems for the developers.

Hence, the mitigation objectives are to increase the technical competency either by identify and create the missing guidelines for large scale testing or facilitate a learning environment for large scale testing without compromising test suite quality.

4.4. Large Test Cases Quantity

Upon scaling, the massive test cases quantity itself is a problematic cause, creating long test codes problem. The large quantity of test cases not only magnifies the problems of the product; it also magnifies the magnitude of the impacts for these problems.

A small mistake such as defining test case data structure without label as shown in Figure 4.4-1 can be magnified into a huge problem. Bad and complex source codes is another case study. A single complicated test subject that offers a wide variety of unrelated functions can complicates its test data structure by consolidating all non-related test data together to form a huge, unnecessary test data structure.

Therefore, this reinforces the mitigation objective to isolate all identified architectural changes from influencing one another, and to reduce test cases quantity by making use of insightful test tools. By isolating all the changes, it indirectly facilitate a learning environment for inexperienced tester discussed in section 4.2.

```
scenarios := []struct {
        inStatus   int
        inMessage  string
        inArg      interface{}
        inStdout   *bytes.Buffer
        inStderr   *bytes.Buffer
        outMessage string
}{
        {
                InfoStatus,
                sample1,
                nil,
                &bytes.Buffer{},
                &bytes.Buffer{},
                "[ INFO ] " + sample1,
        }, {
                WarningStatus,
                sample1,
                nil,
                &bytes.Buffer{},
                &bytes.Buffer{},
                "[ WARNING ] " + sample1,
        }, {
                ... continue 500 lines more
```

Figure 4.4-1 - bad structure definition without element label

4.5. Infrastructure Influences

Infrastructure influences in both development and testing facility is causing the infrastructure differences problem. Testing facility can be influenced by various factors like project timeline, monetary assets, hardware, software, talent pool, resources availability/constraints, etc.

A case study is when a project has limited resources, the allocation usually focuses on development pool, trading off the quality of testing. Without sufficient resources, various problems can occur like inconsistent results from infrastructure differences.

Another case study is that when a tester is specialized in virtualization or container software, the testing tool can be done entirely in software. This benefits from cost saving from overall infrastructure in exchange for introducing possible infrastructure differences problem.

Therefore, another mitigation objective is to ensure the available resources like infrastructure are producing the same results as the development.

4.6. Compliances to Go Standards

The senary cause is the requirement to comply with Go standards like Effective Go coding standards for both source codes and test codes. By having test codes to comply with Go standards, it is easy to cause problems like frequent naming collision if the development team does not practice a consistent and agreed patterns.

Also, it is easy to get into code duplication collision during static analysis with test codes. Duplicated codes often appear in an unplanned test data structure, especially with test cases data definition.

Therefore, the mitigation actions should have an objective to facilitate a means to provide consistency and eliminating duplication for both present and future developments. This way, it ensures the test codes are always complying to Go standards.

5. Mitigations Actions

In this section, this paper lists out all the mitigation actions based on the objectives learned from analyzing the problems' causes for large scale testing. These actions act as important guidelines for developing the large scale testing approach in section 6.

5.1. Use Simulation Test Technique

Simulation test technique provides good isolation for each architectural changes, essentially mitigate the unpredictable architectural changes problem and facilitate easy management to rapidly and chaotic requirement changes. The about isolating each of changes is

One way is to use simulation test technique over the conventional direct data test technique. The process for performing the simulation type testing always follow the following steps:

1. *Learn* - learn the operating environment variables for offset preparations
2. *Prepare* - preparing the simulation environment
3. *Test* - have the test subject runs in the simulated environment
4. *Assert* - check all the output generated by the test subject.

Simulation test technique is also able to mitigate any changes related to test environment such as changing the test machines for running the time-sensitive test suite, essentially isolate any resources related influences. It is done by either heuristically learn the new environment offsets' value or easily-pre-run the test suite in the new environment.

The cost however, is that simulation test technique can be slightly more complicated if the tester is not familiar with simulation type test development. Also, it depends on the availability of mocking the test subject's.

5.2. Proper Isolations

Additionally, a good mitigation action is to essentially isolate all architectural changes. The prerequisite to implement this action is to use simulation technique described in section 5.1. There are different strategies to handle each changes from the start. Although it is unpredictable upfront, tester can use 1000 lines of codes in a single file as a decision factor. Here are some proactive pointers:

1. *You should use large scale approach* if the total test cases can go beyond 100 cases and each test scenario structure contains more than 25 elements, yielding a minimum of 2500 lines of codes excluding structure braces others.
2. *You can start to consider using large scale approach but not necessarily implementing it* if the total test cases is 10 but each test scenario structure contains more than 25 elements, yielding 250 lines of codes excluding struct braces.

These strategies are explained and shown in the following subsections.

5.2.1. Isolate Test Algorithm Changes

Test algorithm changes is related to testers refactoring the test algorithms. This includes the implementation process, assertion, preparation, test approaches etc without affecting the existing number of test cases or altering the source codes.

A good example would be the refactoring `Add(a, b)` function from Figure 2.3.1-3 to Figure 2.3.2-1. The test algorithm had changed from the basic testing approach to table-driven approach due to previous bad decisions like using the wrong test approach from the start. However, the test parameters, the total number of test cases, the source codes, and the test results remained noticeably unchanged.

To isolate this type of changes from others, tester usually separate test algorithm into a single own file.

5.2.2. Isolate Test Parameters Changes

Test parameters changes is usually related to altering the test parameters' data structure from the table-driven test approach. They change by increasing/decreasing the data elements declaration or altering element's data type.

A good example would be the test parameter structure transformation from Figure 5.2.2-1 to Figure 5.2.2-2. There is a huge change by eliminations (notable those inBad* or inMissing* parameters), and new additions (notable switches). This resulted in Figure 5.2.2-3 where the data definition

To isolate this type of changes from others, tester uses flexible "switches", usually in a form of `map[string]bool` listing to hold common elements. That way, common switching elements can be unified under a single "switches" element, allowing the tester to create/remove any common elements at a given time without altering the overall data structure. Also, this facilitates dynamic element declaration, providing tester some flexibility in controlling test data.

```
type testKeyringScenario struct {
>-------inID                                int
>-------inTestType                          strin
>-------inKeyFunctionID                     int
>-------inExportType                        uint
>-------inCipherID                          uint
>-------inCiphertextOrDigest                *[]by
>-------inPlaintext                         *[]by
>-------inInputPlaintext                    *[]by
>-------inSenderLabel                       strin
>-------inReceiverLabel                     strin
>-------inError                             error
>-------inAction                            int
>-------inPublicKey                         *[]by
>-------inPrivateKey                        *[]by
>-------inLimit                             *ciph
>-------inVerdict                           bool
>-------inBadSymmetricRing                  bool
>-------inBadAsymmetricRing                 bool
>-------inBadAuthenticationCipher           bool
>-------inBadKeyHasMalformedCipher          bool
>-------inBadSymmetricKey                   bool
>-------inBadAsymmetricRxKey                bool
>-------inBadAsymmetricTxKey                bool
>-------inBadSignatureKey                   bool
>-------inBadAuthenticationKey              bool
>-------inMissingSymmetricKey               bool
>-------inMissingAsymmetricRxKey            bool
>-------inMissingAsymmetricTxKey            bool
>-------inLimitedUseSymmetricKey            bool
>-------inLimitedUseAsymmetricRxKey         bool
>-------inLimitedUseAsymmetricTxKey         bool
>-------inLimitedUseSignatureKey            bool
>-------inLimitedUseAuthenticationKey       bool
>-------inAssociatedKey                     bool
>-------outDoesNotUseCount                  bool
>-------outData                             bool
>-------outData2                            bool
>-------outError                            bool
}
```

Figure 5.2.2-1 - scenario data structure before test parameter changes

```
type testKeyringScenario struct {
>-------inID                  int
>-------switches              map[string]bo
>-------inTestType            string
>-------inKeyFunctionID       int
>-------inExportType          uint
>-------inCipherID            uint
>-------inCiphertextOrDigest  *[]byte
>-------inPlaintext           *[]byte
>-------inInputPlaintext      *[]byte
>-------inSenderLabel         string
>-------inReceiverLabel       string
>-------inError               error
>-------inAction              int
>-------inPublicKey           *[]byte
>-------inPrivateKey          *[]byte
>-------inLimit               *ciphers.Limi
>-------inVerdict             bool
>-------outDoesNotUseCount    bool
>-------outData               bool
>-------outData2              bool
>-------outError              bool
}
```

Figure 5.2.2-2 - scenario data structure after test parameter changes

```
}, {
>-------inID:                  1346,
>-------inTestType:            testKeyringImpo
>-------inKeyFunctionID:       goodKeyFunction
>-------inExportType:          EncryptedPublic
>-------inCipherID:            goodCipherID,
>-------inCiphertextOrDigest:  goodCiphertext,
>-------inPlaintext:           goodPlaintext,
>-------inSenderLabel:         goodTxLabel,
>-------inReceiverLabel:       goodRxLabel,
>-------inError:               nil,
>-------inAction:              nonAsymmetricPr
>-------inVerdict:             goodVerdict,
>-------inPublicKey:           goodKey,
>-------inPrivateKey:          goodKey,
>-------inLimit:               nil,
>-------switches: map[string]bool{
>-------->-------invalidPEMPassphrase:   true,
>-------->-------missingSymmetricKey:    true,
>-------->-------missingAsymmetricRxKey: true,
>-------->-------missingAsymmetricTxKey: true,
>-------->-------missingSignatureKey:    true,
>-------},
>-------outDoesNotUseCount: true,
>-------outData:            false,
>-------outData2:           false,
>-------outError:           true,
}, {
```

Figure 5.2.2-3 Resultant definition for using the new scenario data structure after parameter changes.

5.2.3. Isolate Test Requirements Changes

Test requirements changes is the alteration from the design input, usually related to source codes addition/removal. With such alteration, the test codes have to be altered in order to react to the new changes.

One example would be enabling `VerifySignature(...)` function placeholder to a fully functional function in the source codes, shown in Figure 5.2.3-1 and Figure 5.2.3-2 respectively. By adding source codes into the test subject, there is a need to add new test algorithm, parameters, and test cases in the test suite to ensure the new source codes are within the test coverage.

To isolate these changes among other, tester can use simulation test technique to fully isolate between the test suite and source codes to cope with the chaotic requirement changes.

```
func (r *Keyring) VerifySignature(label string,
>-------data *[]byte,
>-------digest *[]byte) (verdict bool, err error) {
>-------return false, nil
}
```

Figure 5.2.3-1 - the initial placeholder for `VerifySignature(...)` function

```
func (r *Keyring) VerifySignature(label string,
>-------data *[]byte,
>-------digest *[]byte) (verdict bool, err error) {
>-------if digest == nil {
>-------->-------return false, r.errorf(label, ErrorNi
>-------}

>-------pub, err := r.getKey(Signature, label)
>-------if err != nil {
>-------->-------return false, err
>-------}
>-------if !pub.isUsable(digest, true) {
>-------->-------return false, r.errorf(label, ErrorEx
>-------}

>-------d, err := pub.cipher.Verify(digest, pub.publi
>-------if err != nil {
>-------->-------return false, r.errorf(label, err.Err
>-------}
>-------if d == nil {
>-------->-------return false, r.errorf(label, ErrorBa
>-------}

>-------// check data availability for integrity comp
>-------if data == nil {
>-------->-------return true, nil
>-------}
>-------return bytes.Equal(*d, *data), nil
}
```

Figure 5.2.3-2 the source code changes for `VerifySignature(...)` function.

5.2.4. Isolate Test Environment Changes

Test environment changes is related to the test environment and machine when the test suite is being executed. By changing the test environment, it affects the test results without any codes alteration.

One example is the usage of physical hardware in development environment while using virtual machine in the automated testing facility. The physical and simulated hardware differences can affect some functions like time and randomness in their respective means.

To isolate this problem, tester can deploy simulation test technique by either pre-run the test suite in the new test environment and react accordingly. Alternatively, tester can develop a “train” model to learn environment before preparing the simulation parameters.

5.3. Extensive Use Test Coverage Heatmap

To mitigate the problem with long codes upon scaling by reducing the test cases quantity to only the effective ones, one way is to always make full use of test coverage heatmap to accurately and effectively develop the test cases.

This is done via the heatmap visualization, which provides a “radar” awareness of the test suite against the source codes. This “radar” awareness acts as a directional input for the tester to know what and where to test the source codes. Hence, the tester only creates test cases necessary to achieve full test coverage.

5.4. Heed Golangci-lint Linter Warnings

Apart from using test coverage heatmap extensively, another important mitigation action is to always heed the linter warning from Golangci-lint static analysis report for test codes, especially the code duplication warning. This mitigates a number of problems like frequent naming collision, long test codes, and having the test codes fulfilling Go Standards compliance requirements.

There was a thought that most test codes are meant for duplication but this is a myth:

1. Repeating codes still means unnecessary redundancy.
2. Repeating test scenario value assignments means there is/are
    a. duplicated test cases
    b. unnecessary large scenario structure
    c. possible redundant test cases since tester usually only need to test boundary and invalid values.

If other linters like gosec once a while providing false positive warning, tester should flag them accordingly with the machine-reading comment: `//nolint:gosec` as instructed by the golangci-lint documentation[8].

5.5. Build Own Assertion and Log Functions

To effectively mitigate the missing assertion and limiting logging function problem, one good action would be always build the test suite’s own assertion and abstract the logging function into a consistent test output presenter. The prerequisite to implement this action is to use simulation technique described in section 5.1.

It is done by isolating any new invention or 3rd party assertion or logging tool under a single function, leaving development opportunities for inexperienced tester to gain experience while leaving simple optimization effort for experienced tester to improve the test suite. Tester can also abstract the logging function into a 3rd-party package to make output printing consistent and pretty shown in Figure 5.5-1.

This keeps the assertion from data processing for logging and remains a single judgement statement. Also, it keeps the data log reporting format consistent across all test cases.

```
keyring_test.go:100:
    CASE                                                521
    GIVEN:
    test type                                         = keyring.Sign
    inKeyFunctionID                                   = 0
    inCipherID                                        = 0
    inExportType                                      = 0
    cipher ciphertext/digest                          = &[0 1 2 3]
    cipher plaintext                                  = &[84 101 115
    input plaintext                                   = <nil>
    sender label                                      = uid1Good
    receiver label                                    = uid2Good
    cipher error object                               = <nil>
    cipher action ID                                  = 0
    cipher verdict                                    = false
    cipher public key                                 = <nil>
    cipher private key                                = <nil>
    cipher key properties                             = <nil>
    test switches                                     = map[badKeyHas

    EXPECT:
    does not use key count                            = false
    output 1st data                                   = false
    output 2nd data                                   = false
    output error                                      = true

    GOT:
    cipher has error                                  = <nil>
    cipher has ciphertext                             = &[0 1 2 3]
    cipher has plaintext                              = &[84 101 115
    cipher has digest                                 = &[0 1 2 3]
```

Figure 5.5-1 - a snapshot of a pretty print log output

5.6. Practice Consistent Syntaxes and Styles

Practicing consistent syntax and coding style, always complying to Go standards like Effective Go[18] is also a good way to mitigate numerous problems. It helps by reducing long test codes, reducing naming collision by fulfilling the objective of complying to Go standards, reducing long test codes by using insightful tools and practices, and providing a means for inexperienced tester to learn on-the-job without compromising maintainability. Maintaining syntaxes consistency also provides a searchable environment in a large test codes, which makes scaling easier.

Another way to simplify the naming convention (e.g. keeping it short and organized) in the test codes is to make use of Go's interface for test cases data structure. It effectively bind the test functions to the necessary test cases, keeping the name specific to it while freeing the global names for other usages.

When the coding styles and syntaxes are consistent, it provides an easier learning environment for inexperienced tester to quickly gain the necessary knowledge and experiences on the job, thus, reducing inexperience mistakes when contributing to the test suite.

5.7. Refactor One Step at A Time

To mitigate problems like rescuing a still refactorable test suite caused by technical debt and inexperienced, one good way is to refactor an existing rescuable test suite one step at a time. This is to avoid massive and untrackable problems when transforming an existing approach to the large scale test approach. It is done by refactoring one element at a time and let Go compiler to guides the step-by-step efforts.

Tester should keep an open mind that one can use mixed approaches for continuous improvement instead of one big major refactoring effort. Such big refactoring effort is a high maintenance cost should only execute when there is a business value in it.

A case study is shown in Figure 5.7-1, the tester changes one element (`s.inSwitches[inRelatedInvalidEncryptedPEMData]`) instead of all the cases at one time and runs the compiler. Once the transition is done, it is version controlled and the tester moved onto the next element.

Another case study is shown in Figure 5.7-2 and Figure 5.7-3. Both figures show mixed approaches usage for both scenario definition and algorithm respectively. Tester does not necessarily to fully migrate the test suite until the efforts are worthy to pursue.

```
func testKeyringCompareKeys(t *testing.T,
>-------s testKeyringScenario,
>-------k *key,
>-------kx *key) {
>-------switch {
>-------case s.inSwitches[inRelatedInvalidEncryptedPEMData] &
>-------->-------return
>-------case s.inSwitches[inRelatedInvalidPlainPEMData] && k
>-------->-------return
>-------case s.inUnrelatedInvalidPEMData && k == nil:
>-------->-------return
>-------case s.inUnrelatedValidPEMData && k == nil:
>-------->-------return
>-------case s.inMissingPEMPassphrase && k == nil:
>-------->-------return
>-------case s.inBadPEMPassphrase && k == nil:
>-------->-------return
>-------case s.inInvalidPEMPassphrase && k == nil:
>-------->-------return
>-------case s.inMissingPEMData && k == nil:
>-------->-------return
>-------case s.inBadPEMData && k == nil:
>-------->-------return
>-------case k == nil && kx == nil:
>-------->-------return
>-------case k == nil && kx != nil:
>-------->-------t.Errorf("TESTFAIL: missing key while referer
>-------->-------return
>-------case k != nil && kx == nil:
>-------->-------t.Errorf("TESTFAIL: bogus key when reference
>-------->-------return
>-------}
```

Figure 5.7-1 - changing one element at a time

```
-inCiphertextOrDigest:            goodCiphertext,
-inPlaintext:                     goodPlaintext,
-inSenderLabel:                   goodTxLabel,
-inReceiverLabel:                 goodRxLabel,
-inError:                         nil,
-inAction:                        nonAsymmetricPrepare,
-inVerdict:                       goodVerdict,
-inPublicKey:                     goodKey,
-inPrivateKey:                    goodKey,
-inLimit:                         nil,
-inSwitches:                      map[string]bool{
->-------inBadPEMData:                   true,
-},
-inBadSymmetricRing:              false,
-inBadAsymmetricRing:             false,
-inBadSignatureRing:              false,
-inBadAuthenticationRing:         false,
-inBadKeyHasMalformedCipher:      false,
-inBadKeyHasMalformedUID:         false,
-inBadKeyHasMalformedPurpose:     false,
-inBadKeyHasMalformedCreatedTime: false,
-inBadKeyHasMalformedUpdatedTime: false,
-inBadSymmetricKey:               false,
-inBadAsymmetricRxKey:            false,
-inBadAsymmetricTxKey:            false,
-inBadSignatureKey:               false,
```

Figure 5.7-3 - definition of mixed approaches for scenario definition

6. Large Scale Test Approach

In this section, this paper proposes the resultant large scale test (LST) approach for testing large Go packages. It is based on all the mitigation actions from section 5, covering the approach's objective, basic rules and mindset, file structure, steps of executions and growth, and refactoring efforts recommendation.

This is to facilitate learning environment without compromising test suite quality when scaled for inexperienced tester to work effectively together with those experienced and specialized teammates

6.1. Main Objectives

For implementing large scale test approach, there are a number of main objectives to be cleared of. The main objectives are:

1. **Proper isolation** - to provide proper isolation for all types of identified changes via various ways such as file structuring by responsibilities, its contents, and steps to implement the changes.
2. **Compliant to Go standard** - to ensure the approach complies to Go standards without special exceptions, tools, and rules for keeping things simple, reusing existing test tools for keeping the test suite consistent with source codes.

6.2. Basic Rules and Good Practices

In order to achieve all the objectives effectively, tester has to comply with a list of rules and mindset. Otherwise, it would be very difficult and confusing to implement LST. These rules and good practices are:

1. **Use Table-Driven Test Approach** - It allows the tester to scale a test suite by quantity without duplicating test test algorithms all over the place.
2. **Always use Data Structure to organize Test Parameters** - It facilitates organizations and flexibility. This ensures any new test parameters being added or being removed in future will not introduce heavy lifting changes to the entire test suite.
3. **Always Comply to Go Programming Standards and Practices** - It keeps the syntaxes and codes consistent, heed the warning from static analysis tools like golangci-lint for both source codes and test codes. Without having the same development attention as source codes for test codes, they can grow to become a time-bomb.
4. **Always use Helpers and Subroutines** - It helps to reduce test codes duplication. When using helpers and subroutines, it allows the tester to build simulated environment for its test subject, isolated the test parameters away from test cases configurations. This simplifies the test configurations by introducing a standardized "switches" instead of feeding test parameters to the test subject.

6.3. Proposed File Structure

To maintain proper organization with anticipation of possible addition/removal changes in future, this approach employs a different file structure organization. Figure 6.3-1 shows the directory tree structure holding both the source codes and the test codes. This is a huge difference compared to the original file structure shown in Figure 2.3.1-2.

```
Directory
├── <source>.go
├── <source>_test.go
├── <source>_<MISC>_test.go
├── <source>_<publicAPI>_test.go
└── <source>_scenarios_test.go
```

Figure 6.3-1 - the naming pattern for large scale testing approach.

The roles and responsibilities for each type of files are:

1. **`<source>.go`** holds the source codes.
2. **`<source>_test.go`** holds the test scenario structure declaration, helpers, simulation subroutines, assertion functions, etc. It is the "library" or placeholder for all test codes in this package.
3. **`<source>_scenarios_test.go`** holds the definition of all test cases with the parameters' values. For any increment of test cases, its description, configurations are defined here.
4. **`<source>_<publicAPI>_test.go`** holds the test algorithms for a public API offered by the `<source>.go`. This file describes how a test subject is being tested, calling the helpers and subroutine functions from `<source>_test.go`.
5. **`<source>_<MISC>_test.go`** holds any test algorithms that is not compatible with public API. A good case is to set up

pre-testing tools before running the test like the "`TestMain(m *testing.M)`" function.

The `<source>.go` tag means the main source codes. It should comply to effective Go standards and nothing should change. When a given name is available (example: `passphrase`), the tester should replace all the `<source>` tag with that name. Like the standard Go, all test codes reside in the file with `_test.go` suffix.

6.3.1. Important Guidelines

Although the file structure is clearly defined, there are some important guidelines to comply for avoiding costly pitfalls. The following list holds some warning guides requiring attention:

1. If Golang-lint linter is reporting warnings and bad practices (e.g. "don't repeat yourself") in any test files, tester must refactors and simplify that file before it becomes a complication.
2. The content for each files should not go outside of its roles and responsibilities. Example, when the tester starts defining test cases inside `<source>_test.go`, it is a warning sign that it was done wrongly. Tester should move those test cases into `<source>_scenarios_test.go` instead.
3. Only import any other packages in `<source>_test.go`. If any other files is importing packages aside from `testing` package, it is done wrongly and tester should wrap the involved codes into a subroutine and sent it to the `<source>_test.go` instead.
4. In the test parameter data structure, it must have a `testType` data element. This `testType` acts as a gatekeeper for test algorithms to operate with the associated test values or skip it. This is mainly because all the given test cases' data, related or otherwise, are fed to all test algorithms regardlessly under a loop. Only the gatekeeper can distinguish the test cases is meant for a particular test algorithm.
5.

6.4. Naming Conventions

When deploying large-scale testing approach into the test suite, there are some naming conventions to comply with in order to achieve the objectives and practices.

6.4.1. Go Standards

There should not be any changes or differentiation from Go standards such as commentary styles, short syllabus and insightful names, package naming, function's private and public exposures, etc. Anything that does not complies to *gofmt*, a formatter rule tool should not be committed in anyway.

For any variables, structures, constants, and functions related to test suite, it should always start with *`test`* prefixes. Similar from Go Standards, the capitalized prefix (`Test-`) such as "`func TestObject(...)`" is a reserved function naming convention for test executions.

Figure 6.4.1-1 shows an example template of `<source>_test.go` with a library of assertion. Notice that each subroutine functions begins with the private function *`test-`* prefix. Also, the test parameter data structure also has the same naming prefix. Therefore, the naming pools for test suite and source codes can be safely separated with strict discipline.

```
const (

    testAddStandardLabel = 123

)

type testAddScenario struct {

    ...

}

func testAssertAdd(...) {

    ...

}

func testPrepareAdd(...) {
    ...

}
```

Figure 6.4.1-1 - naming example for LST

6.4.2. Private First

Aside from complying to Go Standards like the *test* prefixes, tester should always use private naming (lower case starting name like "`testName(...)`". Firstly, this can avoid unnecessary public API exposure through test codes. Secondly, it keeps the test suite private to that package alone. In Figure 6.4.1-1, all test functions including internal assertion are in private exposure.

6.4.3. Go Interface Organization

To maintain the one-three short naming convention, tester can use Go interface feature onto the test parameters' data structure. This way, all the functions are grouped inherently, greatly reduces the name length and privatized all the internal functions. Due to the interface grouping effect, that the same name can be reusable onto different data parameters' data structure.

Figure 6.4.3-1 shows the Go interface improvement over Figure 6.4.1-1 example. Notice that both functions' name are now `assert` and `prepare` respectively instead of long phrases.

```
const (

    testAddStandardLabel = 123

)

type testAddScenario struct {

    ...

}

func      (s      *testAddScenario)
assert(...) {

    ...

}

func      (s      *testAddScenario)
prepare(...) {
    ...

}
```

Figure 6.4.1-1 - naming example for LST

6.5. Proper Isolations

To isolate each type of changes, tester should follow the guidelines and uses the large scale approach file structure. This section shows all the steps of large scale testing approach sequentially.

6.5.1. Writing Test Algorithms

The first step is to write test algorithms. The only file involved in this step is `<source>_<publicAPI>_test.go`, where the tester write the test algorithms here. The goal in this step is to clearly write out how the test subject is being simulated and tested, which is to handle any changes came originated from:

1. Test algorithm changes
2. Source codes changes

To isolate test parameter changes, test case quantity changes, and test environment changes from the above changes, tester should keep the test algorithm as simple as possible by:

1. Calling helper or subroutines instead of writing the execution codes.
2. Use simulation configuration, usually in a form of `map[string]bool` data type. It allows the tester to create simulation switches by defining the string and set the value to true. The call out is usually something like: `s.switches[string]`.
3. Write the panic capturing function here since it is part of test algorithm.
4. Use test cases generator function (e.g. `test<DataType>Scenarios()`) to generate all table-driven test scenarios and then loop over them.
5. At the beginning of the loop, always check the test case (scenario) containing the `testType` value permitted for the algorithm. This is to ensure the test algorithms should run on the correct test cases. If the value is incorrect, the algorithm should skip the loop by continuing to the next iteration.

Some tips for writing the test algorithm is always keep the entire test sequences into 4 general steps:

1. *Learn* - learn the environment and prepare offset calibrations for *Prepare* step.
2. *Prepare* - prepare the simulation environment including generating the

required input and output (for later assertion and logging usage).

3. *Test* - shows how the test subject is being tested.
4. *Assert* - perform assertion and logging for the test case. It takes the output of the test subject and test it against the output generated by the simulation environment subroutine in *prepare* step earlier.

For source code changes that breaks its public API structure, it is an entirely new function. Hence, it is expected to break the test algorithm. However, such changes usually involved altering the *Test* section where the function call requires an update.

For subroutine/helper functions changes, this is usually caused by refactoring subroutine/helper functions in the `<source>_test.go`. It is commonly seen as package expands, it involves abstracting a cleaner subroutine/helper functions to use across different test algorithms. However, this change should not break the existing test algorithm in any way.

By doing only this step yields compilation error since the `<source>_test.go` and `<source>_scenarios_test.go` are yet to facilitate all the helpers, data structure, and subroutines functions used in this test algorithm.

Some case studies are shown in Figure 6.5.1-1 and 6.5.1-2. Figure 6.5.1-1 shown the conventional implementation, while 6.5.1-2 shows how to handle test subject with panic behavior. Due to the static data type in Go, it is better to wrap the panicking test subject inside a function that captures the output panic object.

```
package filehelper

import (
>-------"testing"
)

func TestCompare(t *testing.T) {
>-------var v bool
>-------scenarios := testFileHelperScenarios()

>-------for i, s := range scenarios {
>-------->-------if s.testType != testCompare {
>-------->-------->-------continue
>-------->-------}

>-------->-------// prepare
>-------->-------th := s.prepareTestHelper(t)
>-------->-------f := &FileHelper{}
>-------->-------s.prepareFileHelper(f)
>-------->-------f1, f2, verdict := s.prepareFiles()

>-------->-------// test
>-------->-------if s.switches[useDeepCompare] {
>-------->-------->-------v = f.deepCompare(f1, f2)
>-------->-------} else {
>-------->-------->-------v = f.Compare(f1, f2)
>-------->-------}

>-------->-------// assert
>-------->-------th.ExpectUIDCorrectness(i, s.uid, false)
>-------->-------s.assertVerdict(th, verdict, v)
>-------->-------s.log(th, map[string]interface{}{
>-------->-------->-------"filepath1":      f1,
>-------->-------->-------"filepath2":      f2,
>-------->-------->-------"verdict":        v,
>-------->-------->-------"expect verdict": verdict,
>-------->-------})
>-------->-------th.Conclude()
>-------}
}
~
```

Figure 6.5.1-1 - a case study for writing test algorithm

```go
package args

import (
>-------"testing"
)

func TestManagerAdd(t *testing.T) {
>-------scenarios := testManagerScenarios()

>-------for i, s := range scenarios {
>-------->-------if s.testType != testManagerAdd {
>-------->-------->-------continue
>-------->-------}

>-------->-------// prepare
>-------->-------th := s.prepareTHelper(t)
>-------->-------m := NewManager()
>-------->-------f, _, _ := s.prepareDataFlags(m)

>-------->-------// test
>-------->-------err := runTestManagerAdd(m, f)

>-------->-------// assert
>-------->-------th.ExpectUIDCorrectness(i, s.uid, false)
>-------->-------s.assertFlagExists(th, m, f)
>-------->-------s.assertError(th, err)
>-------->-------s.log(th, map[string]interface{}{
>-------->-------->-------"got manager": m.flags,
>-------->-------->-------"got error":   err,
>-------->-------->-------"input flag":  f,
>-------->-------})
>-------->-------th.Conclude()
>-------}
}

func runTestManagerAdd(m *Manager, f *Flag) (err error) {
>-------defer func() {
>-------->-------switch e := recover().(type) {
>-------->-------case error:
>-------->-------->-------err = e
>-------->-------default:
>-------->-------->-------err = nil
>-------->-------}
>-------}()
>-------panic(m.Add(f))
}
```

Figure 6.5.1-2 - a case study for writing test algorithm with panic handling

6.5.2. Writing Test Suite's Library

With test algorithm is now available, tester can proceed to develop the test parameters data structure, helper, and subroutine functions used by the test algorithm in `<source>_test.go`. This is equivalent to developing "library" for the entire test suite or similar to developing developing Go package's source code.

For test parameter, it has its own data structure usually known as "scenario". It holds the test parameters and offers its helper/subroutine function interfaces. With the defined clarity by referencing the test algorithm, tester can pinpoint and develop a reusable helper/subroutine functions instead of blindly develop the test execution codes.

The goal for this step is to handle changes originated from:

1. Test parameters changes
2. Test environment changes

To isolate test algorithm changes, source codes changes, and test cases quantity changes, tester should keep `<source>_test.go` as the test suites' library codes facilitating:

1. subroutine functions for the test algorithms.
2. assertion function to process the test output and results.
3. logging functions to process log data.
4. any helper functions like test helper object creations.
5. no repeating/duplicated helper and assertion functions.
6. test parameter data structure declaration.
7. labelling for magic numbers and values.

For test parameter data structure, at a minimum, it should always offers the following 4 elements:

1. *uid* - the test unique identification number
2. *testType* - the gatekeeper for letting a test algorithm to operate on a given test case.
3. *description* - for tester to communicate and understand a test case, usually a long string.
4. *switches* - for test case to configure the simulation preparations.

For better control and presentation, the *switches* can use string data type as the label while boolean as the value. This provides a constant value for consistency and uniformity, recyclability purposes at coding level.

At this point, if the test cases (scenarios) function is missing, the compilation continues to yield error.

The case studies are shown in both Figure 6.5.2-1 and Figure 6.5.2-2. Their `prepareFunction(...)` and `assertFunction(...)` always relies on the *switches* to alter the parameters generations or checking according to the test cases. Also, in Figure 6.4.2-2, due to the chaotic nature of the data used in a test algorithms, it is okay to use `map[string]interface{}` type to hold various amounts and different types of input/output data from the test algorithm. This makes logging efforts easier.

```go
type testStrHelperScenario struct {
>-------uid         int
>-------testType    string
>-------description string
>-------switches    map[string]bool
}

func (s *testStrHelperScenario) log(th *thelper.THelper,
>-------data map[string]interface{}) {
>-------th.Logf("CASE\n%v\n", s.uid)
>-------th.Logf("TEST TYPE\n%v\n", s.testType)
>-------th.Logf("DESCRIPTION%v\n", s.description)
>-------th.Logf("SWITCHES")
>-------for k, v := range s.switches {
>-------->-------th.Logf("%-20v = %v", k, v)
>-------}
>-------th.Logf("\n")
>-------th.Logf("GOT")
>-------for k, v := range data {
>-------->-------th.Logf("%-20v = %#v", k, v)
>-------}
}

func (s *testStrHelperScenario) prepareTHelper(t *testing.T) *thelper.THelper
>-------return thelper.NewTHelper(t)
}

func (s *testStrHelperScenario) prepareString() (sample, expect string) {
>-------switch {
>-------case s.switches[useCRLF]:
>-------->-------sample = s.constructStrings(useCRLF)
>-------case s.switches[useCR]:
>-------->-------sample = s.constructStrings(useCR)
>-------case s.switches[useLF]:
>-------->-------fallthrough
>-------default:
>-------->-------sample = s.constructStrings(useLF)
>-------}

>-------switch {
>-------case s.switches[expectCR]:
>-------->-------expect = s.constructStrings(useCR)
>-------case s.switches[expectLF]:
>-------->-------expect = s.constructStrings(useLF)
>-------case s.switches[expectCRLF]:
>-------->-------fallthrough
>-------default:
```

Figure 6.5.2-1 - a case study of defining test parameter data structure with its helper function interfaces

```go
type testManagerScenario struct {
>-------uid         int
>-------testType    string
>-------description string
>-------switches    map[string]bool
}

func (s *testManagerScenario) log(th *thelper.THelper,
>-------data map[string]interface{}) {
>-------th.Logf("CASE\n%v\n", s.uid)
>-------th.Logf("TEST TYPE\n%v\n", s.testType)
>-------th.Logf("DESCRIPTION%v\n", s.description)
>-------th.Logf("SWITCHES\n%v\n", s.switches)
>-------th.Logf("GOT")
>-------for k, v := range data {
>-------->-------th.Logf("%-20v = %v", k, v)
>-------}
}

func (s *testManagerScenario) assertError(th *thelper.THelper, err error) {
>-------switch {
>-------case s.switches[missingFlag]:
>-------->-------if err != nil {
>-------->-------->-------return
>-------->-------}
>-------->-------th.Errorf("missing error when having missing flag")
>-------case s.switches[preExistedFlag]:
>-------->-------if err != nil {
>-------->-------->-------return
>-------->-------}
>-------->-------th.Errorf("panic error is incorrect")
>-------default:
>-------}
}

func (s *testManagerScenario) assertFlagExists(th *thelper.THelper,
>-------m *Manager,
>-------f *Flag) {
>-------if m == nil {
>-------->-------return
>-------}

>-------if f == nil {
>-------->-------if !s.switches[missingFlag] {
>-------->-------->-------th.Errorf("given input flag was nil")
>-------->-------}
>-------->-------return
>-------}
```

Figure 6.5.2-2 - a case study of defining test parameters with its assertion function

6.5.3. Writing Test Cases (Scenarios)

With both test parameters and test algorithms available, tester can proceed to develop the test cases in `<source>_scenarios_test.go`. Normally, these test cases are wrapped into a single function, usually named like `<test_parameter_structure>_scenarios(...)`. Some examples are `testFileHelperScenarios()` and `testManagerScenarios()` as seen in Figure 6.4.1-1 and Figure 6.4.1-2 respectively. This function yields a `slice` (in English: list) of test scenarios containing test configurations to run on various associated test algorithms.

The goal for this step is to handle changes originated from:

1. Test cases quantity changes

To isolate test parameters changes, test algorithm changes, source code changes, and test environment changes, tester should keep the test cases scenario generations to:

1. Only defining the values of the test cases and nothing else.
2. Strictly use the scenario switches to define the test cases values.
3. Write the test cases as if it will not be revisited after scaling.

Due to the *switches* data element availability, tester can freely defines any given switches for test helpers/subroutines functions to configure the simulation environment. If any unforeseen future test parameter changes occurs, user can safely alters the test parameters without needing to go through all the test cases.

To effectively write the test cases, tester can use Go test coverage heat map tool to expand the test cases effectively and accurately. This allows tester to focus on boundary values and invalid values testing instead of blindly cherry-picking test values.

For keeping the test case's description in a readable, sane manner, tester can use Go's raw string (`` `...` ``) opening convention instead of conventionals. Using raw string opening convention reduces the needs to perform strings concatenation coding, which is an obfuscation to readability.

If the test cases is written from scratch, tester can first define the desired passing test case value (known as "happy path") to keep everything running. This way, tester can ensure various parts of the test suite are working fine before expanding to other test cases.

Figure 6.5.3-1 and Figure 6.5.3-2 shown some case studies for writing test cases (scenarios). Notice that both 6.5.3-1 and 6.5.3-2 has similar data structure patterns but is able to serve different test packages due to the *switches* data element. Also, both case studies has shown good usage of *testType* data element where the test cases are specifically meant for their respective test algorithms. The *description* value is clear, clean, and readable at code level.

At this point, the Go compilation should work properly since every dependencies are met. Any compilation errors at this point should be related to smelly codes like syntax errors, etc.

```
package strhelper

func testStrHelperScenarios() []testStrHelperScenario {
>-------return []testStrHelperScenario{
>-------->-------{
>-------->-------->-------uid:      1,
>-------->-------->-------testType: testToUNIX,
>-------->-------->-------description: `
ToUNIX should work properly when given a proper string with CRLF new lines.
`,
>-------->-------->-------switches: map[string]bool{
>-------->-------->-------->-------useCRLF:  true,
>-------->-------->-------->-------expectLF: true,
>-------->-------->-------},
>-------->-------}, {
>-------->-------->-------uid:      2,
>-------->-------->-------testType: testToUNIX,
>-------->-------->-------description: `
ToUNIX should work properly when given a proper string with LF new lines.
`,
>-------->-------->-------switches: map[string]bool{
>-------->-------->-------->-------useLF:    true,
>-------->-------->-------->-------expectLF: true,
>-------->-------->-------},
>-------->-------}, {
>-------->-------->-------uid:      3,
>-------->-------->-------testType: testToUNIX,
>-------->-------->-------description: `
ToUNIX should work properly when given a proper string with CR new lines.
`,
>-------->-------->-------switches: map[string]bool{
>-------->-------->-------->-------useCR:    true,
>-------->-------->-------->-------expectLF: true,
>-------->-------->-------},
>-------->-------}, {
>-------->-------->-------uid:      4,
>-------->-------->-------testType: testToWindows,
>-------->-------->-------description: `
ToWindows should work properly when given a proper string with CRLF new lines.
`,
>-------->-------->-------switches: map[string]bool{
>-------->-------->-------->-------useCRLF:    true,
>-------->-------->-------->-------expectCRLF: true,
>-------->-------->-------},
>-------->-------}, {
>-------->-------->-------uid:      5,
>-------->-------->-------testType: testToWindows,
```

Figure 6.5-3-1 - A simple case study for defining test cases scenarios.

```
package args

func testManagerScenarios() []testManagerScenario {
>-------return []testManagerScenario{
>------->-------{
>------->------->-------uid:      1,
>------->------->-------testType: testManagerAdd,
>------->------->-------description: `
Add should work properly given a proper flag object`,
>------->------->-------switches: map[string]bool{
>------->------->------->-------seekInt: true,
>------->------->-------},
>------->-------}, {
>------->------->-------uid:      2,
>------->------->-------testType: testManagerAdd,
>------->------->-------description: `
Add should work properly given a missing flag object`,
>------->------->-------switches: map[string]bool{
>------->------->------->-------missingFlag: true,
>------->------->-------},
>------->-------}, {
>------->------->-------uid:      3,
>------->------->-------testType: testManagerParse,
>------->------->-------description: `
Parse should work properly given a proper flag seeking Int. The argument
pattern is "./program --arg 123`,
>------->------->-------switches: map[string]bool{
>------->------->------->-------seekInt: true,
>------->------->-------},
>------->-------}, {
>------->------->-------uid:      4,
>------->------->-------testType: testManagerParse,
>------->------->-------description: `
Parse should work properly given a proper flag seeking Int. The argument
pattern is "./program -a 123`,
>------->------->-------switches: map[string]bool{
>------->------->------->-------seekInt:                true,
>------->------->------->-------prepareShortArgKeyword: true,
>------->------->-------},
>------->-------}, {
>------->------->-------uid:      5,
>------->------->-------testType: testManagerParse,
>------->------->-------description: `
Parse should work properly given a proper flag seeking Int. The argument
pattern is "./program -a=123`,
>------->------->-------switches: map[string]bool{
>------->------->------->-------seekInt:                true,
```

Figure 6.5-3-2 - Another case study for defining test cases scenarios

6.6. Refactoring Guidelines

For refactoring an existing package, both developer and tester must consider the business value behind it. If it is too costly (may cost a month), they can consider using mixed-approaches to refactor small pieces of changes at a time instead of as a whole.

When it is worthy to refactor an existing test suite to match this test approach, one should do the following in sequence:

1. Lock the entire test suite with the existing test results. The goal is to maintain the same test result at all times.
2. Alters 1 parameter at a time / apply 1 changes at a time. Example, adding a switch map object:
    a. Formulate the map switch
    b. Delete the parameter from the struct
    c. Delete the log printing that deleted parameter
    d. Save and let error report guides you
    e. Change each deployed parameter to match the new switches
    f. Change the scenarios listing (can take time)
    g. Save.
    h. Re-run test. The result should be consistent to 1
3. Stage the code into the version control commitment like Git add.
4. Repeat step #2 and step #3 for other changes..
5. Re-run final testing to confirm test results consistent as step #1.
6. Perform thorough code reviews and apply corrections when needed.
7. Commit the codes.

With the large scale test approach set up and working, developers and testers can now scale the package with controllable means. They can now identify the type of changes introduced in the future and alters the package without getting into unmaintainable test codes nightmares.

7. Conclusion

Testing in Go is relatively easy with readily available test tools and facilities provided by the language itself. There are many tutorials made available for Go beginners to learn and adapt to it. However, when scaled such as going beyond 10,000 lines of codes, the approaches introduced by these tutorials are insufficient, potentially causing a scarily large, unmaintainable test codes which require big efforts to refactor.

This is where large scale test (LST) approach is filling the gap. LST provides proper isolation for various architectural changes, practices to manage naming conventions, rules and practices to build a large scale test suite. Also, it utilizes Go's test coverage heatmap tools for effective testing.

Tester can consider using mixed-approaches or full LST approach depending on business values. In any cases, LST approach has a refactoring guidelines to smoothen the transformation.

However, LST approach is not recommended to Go beginners since their priority is to get familiar with testing in Go, not about advanced testing like benchmarking or dealing with scaling. This approach should be briefly mentioned in the testing in Go introduction but let the beginners to explore on his/her own.

8. License

This paper is licensed under:

CC-BY

This license lets others distribute, remix, tweak, and build upon your work, even commercially, as long as they credit you for the original creation. This is the most accommodating of licenses offered. Recommended for maximum dissemination and use of the licensed materials.

9. Acknowledgement

Thank Lim Lee Booi for her continuous constructive criticism of the manuscript despite all the hardships.

ありがとうございました | Thank you

10. References

[1] GOLANG.ORG, 2019, "Documentation", Golang.org, viewed July 08, 2019, available at: https://golang.org/doc/

[2] MARGARET ROUSE, SARAH LEWIS, 2018, "Go (Programming Language)", TechTarget, viewed July 08, 2019, available at: https://searchitoperations.techtarget.com/definition/Go-programming-language

[3] IRINA SIDORENKO, 2019, "Should I Go? The Pros and Cons of Using Go Programming Language", Hackernoon.com, viewed July 08, 2019, available at:

https://hackernoon.com/should-i-go-the-pros-and-cons-of-using-go-programming-language-8c1daf711e46

[4] PLURALSIGHT, 2016, "An Introduction to the Go Compiler", Pluralsight LLC, viewed July 08, 2019, available at: https://www.pluralsight.com/blog/software-development/the-go-compiler

[5] JOHANNES LIEBERMANN, 2017, "Why Golang Is Great for Portable Apps", Codeburst, viewed July 08, 2019, available at: https://codeburst.io/why-golang-is-great-for-portable-apps-94cf1236f481

[6] KEVAL PATEL, 2017, "Why should you learn Go?", medium.com, viewed July 08, 2019, available at: https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65

[7] CHEW KEAN HO, LIM LEE BOOI, 2018, "Descriptive Review for Software Testing Algorithm", ResearchGate.net, viewed July 08, 2019, available at: http://doi.org/10.13140/RG.2.2.11325.10724

[8] GOLANGCI, 2019, "Golangci-lint", Github.com, viewed July 08, 2019, available at: https://github.com/golangci/golangci-lint

[9] CHEW KEAN HO, 2018, "Testing", *Guides > Go*, Google Sites, viewed July 08, 2019, available at: https://sites.google.com/view/chewkeanho/guides/go/testing?authuser=0

[10] CHEW KEAN HO, 2018, "Benchmark", *Guides > Go*, Google Sites, viewed July 08, 2019, available at: https://sites.google.com/view/chewkeanho/guides/go/benchmark?authuser=0

[11] DAVE CHENEY, 2019, "Practical Go: Real world advice for writing maintainable Go programs", dave.cheney.net, viewed July 08, 2019, available at: https://dave.cheney.net/practical-go/presentations/qcon-china.html

[12] ELLIOT FORBES, 2018, "An Introduction to Testing in Go", *Tutorial - Golang*, Tutorial Edge, viewed July 08, 2019, available at: https://tutorialedge.net/golang/intro-testing-in-go/

[13] CALEB DOXSEY, 2019, "Testing", *Go Resources - An Introduction to Programming in Go*, viewed July 08, 2019, available at: https://www.golang-book.com/books/intro/12

[14] MARTIN TOURNOJI, 2018, "TableDrivenTests", *Golang/go - Wiki*, Golang group via Github.com, viewed July 08, 2019, available at: https://github.com/golang/go/wiki/TableDrivenTests

[15] BIV, AXAPAXA, 2016, "Timing attack and good coding practices", Cryptography StackExchange, viewed July 08, 2019, available at: https://crypto.stackexchange.com/questions/41691/timing-attack-and-good-coding-practices

[16] PETER SCHWABE, 2016, "Timing Attack and Countermeasures", *Summer school on real-world crypto and privacy*, Šibenik, Croatia, viewed July 08, 2019, available at: https://summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf

[17] MITCHELL HASHIMOTO, 2016, "Advanced Testing in Go", *Gophercon 2017*, Speakerdeck.com, viewed July 08, 2019, available at: https://speakerdeck.com/mitchellh/advanced-testing-with-go

[18] GOLANG.ORG, 2019, "Effective Go", *The Go Programming Language > Docs*, golang.org, viewed July 08, 2019, available at: https://golang.org/doc/effective_go.html

[19] CHEW KEAN HO, 2018, "Struct Memory Alignment", *Guides > Software Coding Style*, Google Sites, viewed July 09, 2019, available at: https://sites.google.com/view/chewkeanho/guides/software-coding-styles/in-general/struct-memory-alignment