

# Einstieg in die Ausführung und Analyse von Tests in GO

**Götz-Henrik Wiegand | Matr. 70722**

Projektbericht im Rahmen des 5. Semesters des Informatik Bachelor Studiums

Betreuer: Prof. Dr. Martin Sulzmann

Hochschule Karlsruhe – Technik und Wirtschaft

an der

Fakultät für Informatik und Wirtschaftsinformatik

WS2020/21

---

## Zusammenfassung

Der gezielte Einsatz von Tests in der Entwicklung kann viele Probleme und Stunden sparen und doch scheuen sich viele Entwickler davor, Tests zu schreiben und diese in ihren Entwicklungsprozess früh mit ein zu binden. Damit in der GO-Programmiersprache Entwickler eine kleinere Hürde haben, Tests zu schreiben und ihre Programme mit Tests zu analysieren, wurden einige sehr hilfreiche Mechanismen und Werkzeuge in dem Test-Tool implementiert.

Für die Präsentation und Erklärung dieser wurde ein Command-Line-Interface(CLI)<sup>1</sup> mit dem Cobra-Framework<sup>2</sup> erstellt in dem für die verschiedenen, in der Projektarbeit behandelten, GO-Tools entsprechende Beispiele implementiert wurden. Für jedes Beispiel wird der entsprechende Code präsentiert und die Ausgabe im Terminal dokumentiert. Alle in der Projektarbeit gezeigten Beispiele sind nur mit den GO-eigenen Tools sowie einer Kommandozeile und einem Webbrowser ausführbar.

Ziel der Projektarbeit ist es, eine einsteigerfreundliche Übersicht über die Thematik zu geben und an die Ausführung und Analyse von Tests in GO heranzuführen.

---

# Einleitung

Im Rahmen der Projektarbeit wird ein Programm in Go geschrieben, an dem das GO-Test-Tool vorgestellt werden soll. Hierfür werden gängige Testmethoden wie z.b. Unit-Tests gezeigt und wie diese in GO umgesetzt werden können.

Die Arbeit gliedert sich in Voraussetzungen, Testausführungen und Testanalyse sowie ein kurzes Fazit am Schluss. In der Voraussetzung werden die Materialien und Methoden vorgestellt, die die Grundlage bilden, auf der die Tests ausgeführt und analysiert wurden. In der Testausführung wird schrittweise erklärt, wie Tests in GO aus der Kommandozeile aufgerufen werden können und welche Parameter und Möglichkeiten die Sprache in dieser Hinsicht mit sich bringt. In der Testanalyse werden Tools und Möglichkeiten vorgestellt, die eigenen Tests zu analysieren und für andere bereitzustellen. In diesem Kapitel wird das Thema Benchmarks aufgrund des Umfangs der Arbeit vernachlässigt.

# Voraussetzungen

Die Programmiersprache Go wurde vor allem für das Entwickeln von skalierbaren Webanwendungen und Cloud Computing entwickelt. Neben diesen Aufgaben, hat sich GO auch als eine gängige Sprache für das Entwickeln von Command Line Interfaces (CLI) herausgestellt.

Mit Hilfe eines für diesen Zweck erstellten CLIs<sup>1</sup> werden verschiedene Beispiele dargestellt und Möglichkeiten aufgezeigt, die das Go-Testing Tool bereitstellt. Die Entwicklung des Programms sowie die Ausführung der Tests werden mithilfe von Visual Studio Code und der GO-Erweiterung<sup>3</sup> durchgeführt. Für die Programmierung des CLIs werden das Cobra-Framework<sup>2</sup> sowie die GO eigenen Pakete[^GO-Packages] "fmt", "testing" und "strings" genutzt. Alle Testbefehle und Programmausführungen werden über den Terminal ausgeführt und beschrieben. Die Einbettung und Umsetzung der Ausführung von Tests in Grafischen Oberflächen wird hier nicht weiter berücksichtigt.

Im Folgenden werden mehrere *Flags* besprochen. Flags sind Parameter die in der Kommandozeile mit übergeben werden. Flags werden durch ein Leerzeichen von dem Kommando getrennt und haben als Präfix einen oder zwei Bindestriche oder auf DOS-Systemen einen Schrägstrich. In dem GO-Kommandozeilenwerkzeug wird die Schreibweise mit einem Bindestrich bevorzugt.

# Testausführung

Im folgenden Abschnitt werden verschiedene Möglichkeiten und Arten wie Tests in GO ausgeführt werden können gezeigt. Es werden verschiedene Parameter und Möglichkeiten besprochen und an Codebeispielen ausgeführt. Der gesamte Code zu dieser Arbeit befindet sich in einem Repository auf Github<sup>1</sup>. In jedem Abschnitt wird der entsprechende *Code* der Tests präsentiert und die schrittweise Ausführung im Terminal als *Ausgabe* dokumentiert.

## Allgemein

Der erste Tests soll die Funktion `printHelloWorld()` in `./cmd/bsp1.go` testen. Hierfür wurde eine weitere Datei im gleichen Ordner und Paket mit dem Namen `bsp1_test.go` angelegt. Die Funktion `printHelloWorld()` gibt als Rückgabewert einen String mit dem Inhalt `"Hello World"` zurück.

In der Commandozeile wird die Funktion nach dem Installieren mit `go install` durch das eingeben von `mycli bsp1` aufgerufen und gibt den Rückgabewert der Funktion `printHelloWorld()` auf dem Terminal aus.

Um die Funktion zu testen wird in der Test-Datei (`./cmd/bsp1_test.go`) eine Funktion angelegt, die überprüfen soll, ob der zurückgegebene Wert mit dem String `"Hello World"` übereinstimmt. Sollte der Test fehlschlagen, so wird als Fehler-Mitteilung der Text `" 'Hello World' test failed"` zurückgegeben.

Um diesen Test aus zu führen benötigen wir den `go test`-Befehl. Wird dieser ohne weitere Argumente ausgeführt, so kompiliert GO das aktuelle Verzeichnis und führt alle gefundenen Tests darin aus.

Um eine spezifische Datei zu testen reicht der Befehl `go test` mit dem entsprechenden Pfad der Testdatei dahinter. In dem hier gezeigten Beispiel also `go test ./cmd/`.

Bei einer erfolgreichen Ausführung bekommen wir in der Ausgabe eine kurze Übersicht über den Status, den Paketpfad und die Dauer der Ausführung des Testes. Um eine Auflistung der durchgeführten Tests zu bekommen wird das `verbose`-Flag benötigt. Test die mit diesem Flag ausgeführt werden, zeigen für jeden durchgeführten Tests ein Startpunkt, der mit `=== RUN` gekennzeichnet wird und einen Endpunkt, der bei erfolgreichem Test mit `--- PASS` oder wenn ein Fehler auftritt mit `--- FAIL` beendet wird. Wird ein Test übersprungen, erhält man für diesen Test `--- SKIP` als Ausgabe. Logs werden bei erfolgreichem Testdurchlauf auch nur angezeigt, wenn das `verbose`-Flag gesetzt wurde.

Das `verbose`-Flag kann man als `go test -v` oder als `go test -verbose` einsetzen. Im weiteren Verlauf des Berichtes wird nur die kürzere Schreibweise (`-v`) verwendet, alle Befehle sind analog auch mit der vollen Schreibweise (`-verbose`) ausführbar.

## Code

```
// printHelloWorld() aus ./cmd/bsp1.go
func printHelloWorld() string {
    return ("Hello world")
}

// TestPrintHelloWorld() aus ./cmd/bsp1_test.go
func TestPrintHelloWorld(t *testing.T) {
    if "Hello world" != printHelloWorld() {
        t.Error("'Hello world' test failed!")
    }
}
```

## Ausgabe

```
tr33@bug:~$ myCli bsp1
Hello world
tr33@bug:~$ go test ./cmd/bsp1
ok      github.com/Tr33Bug/myCli/cmd    0.164s
tr33@bug:~$ go test -v ./cmd/bsp1
=== RUN   TestPrintHelloWorld
--- PASS: TestPrintHelloWorld (0.00s)
PASS
ok      github.com/Tr33Bug/myCli/cmd    0.162s
```

## Spezifische Tests ausführen

Werden nun Änderungen an einer Funktion vorgenommen und man möchte diese anhand der geschriebenen Tests prüfen, ohne alle Tests des gesamten Projektes aus zu führen, so kann man dies über `go test` verknüpft mit dem `run`-Flag und dem Namen der Testfunktion erreichen. Damit können eine oder mehrere Testfunktionen nach ihrem Namen aufgerufen werden.

Es soll aus dem `./cmd/bsp2_test.go` nur die Funktion `TestPrintHello()` ausgeführt werden. Hierfür wird in dem Terminal `go test -run=TestPrintHello$ ./cmd/` verwendet. Das Dollarzeichen am Ende des Namens legt fest, dass GO nach genau diesem Test suchen soll (direkter Aufruf). Lassen wir das Dollarzeichen weg und führen `go test -run=TestPrint ./cmd/` aus, so werden nach allen Tests gesucht, die mit "TestPrint" anfangen und dann ausgeführt("beginnt mit" Aufruf). In unserem Beispiel sind in der Ausgabe dann die Testergebnisse beider Tests und des Tests aus dem `./cmd/bsp1_test.go` zu sehen (siehe Ausgabe).

In der `Test Print world()` Funktion wird mit einem Subtest gearbeitet. Diese sind besonders sinnvoll, um komplexere Funktionen zu testen und abzudecken. In diesem Beispiel zu Demonstrationszwecken stark vereinfacht. Auch diese Subtests können über die `run`-Flag aufgerufen werden. Hierfür wird zuerst der übergeordnete Test und dann mit einem Schrägstrich der Subtest genannt. Auch hier sind wieder "direkte" oder "beginnen mit"-Aufrufe möglich.

Für weitere Informationen zu zum Beispiel Regulären Ausdrücken siehe <https://golang.org/pkg/testing/> in dem Kapitel "Subtests and Sub-benchmarks".

## Code

```
// printHello() und printWorld() aus ./cmd/bsp2.go
func printHello() string {
    return ("Hello")
}

func printWorld() string {
    return ("World")
}

// () aus ./cmd/bsp2_test.go
func TestPrintHello(t *testing.T) {
    if "Hello" != printHello() {
        t.Error("Printing Hello has failed!")
    }
}

func TestPrintWorld(t *testing.T) {
    t.Run("subtestPrintWorld", func(t *testing.T) {
        if "World" != printWorld() {
            t.Error("Printing world has failed!")
        }
    })
}
```

## Ausgabe

```
tr33@bug:~$go test -v -run=TestPrintHello$ ./cmd/
=== RUN    TestPrintHello
--- PASS: TestPrintHello (0.00s)
PASS
ok        github.com/Tr33Bug/myCli/cmd    0.161s
tr33@bug:~$go test -v -run=TestPrint ./cmd/
=== RUN    TestPrintHelloWorld
    bsp1_test.go:11: This Log should only appear in verbose mode or if something went
wrong
--- PASS: TestPrintHelloWorld (0.00s)
=== RUN    TestPrintHello
--- PASS: TestPrintHello (0.00s)
=== RUN    TestPrintWorld
=== RUN    TestPrintWorld/subtestPrintWorld
--- PASS: TestPrintWorld (0.00s)
--- PASS: TestPrintWorld/subtestPrintWorld (0.00s)
PASS
ok        github.com/Tr33Bug/myCli/cmd    0.160s
tr33@bug:~$go test -v -run=TestPrintWorld$/subtestPrint ./cmd/
=== RUN    TestPrintWorld
=== RUN    TestPrintWorld/subtestPrintWorld
--- PASS: TestPrintWorld (0.00s)
--- PASS: TestPrintWorld/subtestPrintWorld (0.00s)
PASS
ok        github.com/Tr33Bug/myCli/cmd    0.160s
```

# Tests im Short-Modus ausführen

In GO gibt es die Möglichkeit, Tests in einem Short-Modus aus zu führen. Hat ein Projekt sehr komplexe Tests, die viel Zeit brauchen, um ausgeführt zu werden, so bietet es sich an, bei der Testentwicklung darauf schon Rücksicht zu nehmen und besonders lange Tests in dem Short-Modus zu überspringen.

Die Funktion `testing.Short()` gibt einen Boolean-Wert zurück der `true` ist, sollte der Test in diesem Modus aufgerufen worden sein. Nun lassen sich damit verschiedene Abhängigkeiten erstellen. In diesem Beispiel wird die `t.Skip()`-Funktion ausgeführt, die den Test im Short-Modus überspringt.

Um Tests in diesem Modus ausführen zu lassen, wird die `short`-Flag zu dem Testbefehl hinzugefügt. Für das unten gegebene Beispiel aus `./cmd/bsp3_test.go` wird somit `go test -short ./cmd/` wobei hier aus Übersichtsgründen nur der Test mit dem oben eingeführten direkten Aufruf ausgeführt wird.

## Code

```
// waitAndPrint() aus ./cmd/bsp3.go
func waitAndPrint() string {
    time.Sleep(5 * time.Second)
    return "Hello World"
}

// TestwaitAndPrint() aus ./cmd/bsp3_test.go
func TestwaitAndPrint(t *testing.T) {
    // tell the testengine to skip the test in short mode
    if testing.Short() {
        t.Skip("This test is to long, skip in short mode")
    }
    // start test
    if "Hello world" != waitAndPrint() {
        t.Error("waiting and printing Hello world has failed")
    }
}

}
```

## Ausgabe

```
tr33@bug:~$go test -short -v -run=TestwaitAndPrint$ ./cmd/
=== RUN    TestwaitAndPrint
    bsp3_test.go:13: This test is to long, skip in short mode
--- SKIP: TestwaitAndPrint (0.00s)
PASS
ok      github.com/Tr33Bug/myCli/cmd    0.160s
tr33@bug:~$go test -v -run=TestwaitAndPrint$ ./cmd/
=== RUN    TestwaitAndPrint
--- PASS: TestwaitAndPrint (5.00s)
PASS
ok      github.com/Tr33Bug/myCli/cmd    5.165s
```

# Umgang mit fehlschlagenden Tests

Tests sind dafür da, in einem Fehlerfall fehlszuschlagen. Das kann den Entwicklern viel Leid ersparen, da bei einer einheitlichen Teststruktur sehr schnell das Problem gefunden werden kann. Wird ein Testdurchlauf gestartet, werden alle Tests durchgeführt und am Ende zusammengefasst, welche fehlgeschlagen sind. Bei einem Projekt mit vielen und auch umfangreichen Tests kann das sehr lange dauern, selbst wenn der erste oder zweite Test fehlschlägt und damit vielleicht schon die nötigen Informationen für die Korrekturen oder Verbesserungen vorliegen.

Um die Tests nach dem ersten Fehlschlag zu stoppen, wird das *failfast*-Flag verwendet. In dem vierten Beispiel(`./cmd/bsp4_test.go`) ist ein Test Implementiert, der in der in jedem fall fehlschlagen sollte. In der `./cmd/bsp4_test.go`-Datei ist nach dem fehlschlagenden Test ein zweiter Test implementiert, der 4 Sekunden warten soll um ein komplexeren langen Test zu simulieren.

Bei der Ausführung der Tests ohne das *failfast*-Flag schlägt der Durchlauf nach ungefähr 9 Sekunden fehl. Mit dem *failfast*-Flag bricht der Durchlauf sofort bei einem Fehlschlag ab und braucht damit nur ungefähr 5 Sekunden. Je nach Anwendungsfall muss auch hier entschieden werden, ob das schnelle Fehlschlagen gewünscht ist oder nicht.

## Code

```
// sumUp() aus ./cmd/bsp4.go
func sumUp(x, y int) int {
    return x + y
}

// TestSumUp() aus ./cmd/bsp4_test.go
func TestSumUp(t *testing.T) {
    a, b := 5, 5
    result := sumUp(a, b)
    t.Log("The following test should fail. If you dont want it to fail, change the test in
./cmd/bsp4_test.go")
    if result != 11 {
        t.Errorf("This test should fail. Dont worry!")
    }
}
```

## Ausgabe

```
tr33@bug:~$go test ./cmd/
--- FAIL: TestSumUp (0.00s)
    bsp4_test.go:14: The following test should fail. If you dont want it to fail, change
the test in ./cmd/bsp4_test.go
    bsp4_test.go:16: This test should fail. Dont worry!
FAIL
FAIL    github.com/Tr33Bug/myCli/cmd    9.165s
tr33@bug:~$go test -failfast ./cmd/
--- FAIL: TestSumUp (0.00s)
    bsp4_test.go:14: The following test should fail. If you dont want it to fail, change
the test in ./cmd/bsp4_test.go
    bsp4_test.go:16: This test should fail. Dont worry!
FAIL
FAIL    github.com/Tr33Bug/myCli/cmd    5.160s
FAIL
```



# Zwischengespeicherte Tests und mehrfaches Ausführen von Tests

Es gibt Funktionen, die nach einer erneuten Durchführung einen anderen Wert zurückgeben. Ein einfaches Beispiel dafür wäre eine Funktion mit einer Abhängigkeit von einer Wahrscheinlichkeit. Wird der Test nun ein Mal ausgeführt und schlägt nicht fehl, so wird das Ergebnis in einem Cache gespeichert und bei erneuter Ausführung unter den gleichen Bedingungen (keine Änderung am Code des Paketes) wird der Test immer positiv ausfallen. Im normalen Fall ist das ein erheblicher Vorteil, da es Testvorgänge um ein Vielfaches beschleunigen kann und nur die Tests neu ausführt, die sich verändert haben. In einem Fall wie oben beschrieben, kann dies aber zu einer falschen Annahme führen, dass alle Funktionen immer den richtigen Wert liefern.

Soll der Cache geleert werden, damit das gesamte Projekt neu getestet wird, kann eine Leerung mit dem `go clean -testcache` erreicht werden.

Eine weitere Möglichkeit, einen oder alle Tests komplett neu ausführen zu lassen, ist das Stress-Testing. Beim Stress-Testing wird der Test viele Male ausgeführt, um bewusst zu testen, ob Funktionen bei erneuter Ausführung das erwartete Ergebnis liefern. Bei dem Stress-Testing ist es unerlässlich, die Ergebnisse nicht aus dem Cache zu laden, sondern diese jedes Mal neu auszuführen.

Wie oft die Tests durchgeführt werden, wird über die *count*-Flag geregelt. Mit dieser wird der Test mit `-count=1` noch einmal aufgerufen, ohne dass das Ergebnis aus dem Cache genommen wird. Soll der Test viele Male ausgeführt werden, wird der Counter auf eine höhere Zahl wie `-count=1000` gesetzt.

Das Beispiel aus `./cmd/bsp5.go` zeigt eine Funktion, die zufällig einen der Strings aus dem String-Array `answers` zurückgibt. Der entsprechende Test dazu in der `./cmd/bsp5_test.go`-Datei überprüft ob die Funktion den String "Hello" zurück gibt.

In dem ersten Testfall geht der Test ohne Fehlschlag durch, wie in der Ausgabe zu sehen ist. Bei erneuter Ausführung läuft der Test erneut fehlerfrei durch und statt der erwarteten Dauer, die der Test benötigt erscheint dort ein `(cached)` was verdeutlicht, dass das Ergebnis aus dem Cache geladen wurde und der Test nicht wirklich erneut ausgeführt wurde. Um nun den Test noch ein Mal ohne Cache ausführen zu lassen wird ein `-count=1` dem Testbefehl hinzugefügt. In diesem Fall ist das Ergebnis der nächsten Ausführung wieder fehlerfrei, braucht aber etwas länger und ist nicht aus dem Cache geladen worden.

Um zu zeigen, dass es wirklich Fehler geben kann, soll nun der Test zehnmal ausgeführt werden. In der Ausgabe im ausführlichen Modus ist zu sehen, wie oft der Test fehlgeschlagen und erfolgreich verlaufen ist. In dem Beispiel ist der Test nur viermal erfolgreich ausgeführt worden.

## Code

```
// randomReturnHello() aus ./cmd/bsp5.go
func randomReturnHello() string {
    rand.Seed(time.Now().UnixNano())
    answers := []string{
        "...",
        "Hello",
        "Hello",
        "...",
        "...",
    }

    // return answers[rand.Intn(len(answers))]
    return answers[rand.Intn(len(answers))]
}

// TestRandomReturnHello() aus ./cmd/bsp5_test.go
func TestRandomReturnHello(t *testing.T) {
    answer := randomReturnHello()
    if answer != "..." {
        t.Errorf("The test failed, probably try out one more time...")
    }
}
```

## Ausgabe

```
tr33@bug:~$go test -run=TestRandomReturnHello$ ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    0.063s
tr33@bug:~$go test -run=TestRandomReturnHello$ ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    (cached)
tr33@bug:~$go test -count=1 -run=TestRandomReturnHello$ ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    0.072s
tr33@bug:~$go test -v -count=10 -run=TestRandomReturnHello$ ./cmd/
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
--- PASS: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
--- PASS: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
--- PASS: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
--- PASS: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
--- PASS: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
=== RUN   TestRandomReturnHello
    bsp5_test.go:13: The test failed, probably try out one more time...
--- FAIL: TestRandomReturnHello (0.00s)
FAIL
FAIL    github.com/Tr33Bug/myCli/cmd    0.073s
FAIL
```

---

## Timeout setzen

Um zu verhindern, dass sich ein Test aufhängt, gibt es die Möglichkeit, einen *Timeout* zu setzen. Das ist eine Zeit, die der Test wartet, bis dieser vom System abgebrochen wird, wenn er keine Rückmeldung bekommt. Der *Timeout* wird über eine weitere Flag gesetzt(`-timeout [Zeit]`). Die Flag braucht als Argument eine Zeitangabe, die mit der `time.ParseDuration()`-Funktion<sup>4</sup> gelesen werden kann(Beispielsweise 30s oder 5ms).

Es empfiehlt sich dieses *Timeout* bei automatisierten Testdurchläufen zu setzen, da dort oft die einzelnen Tests nicht ersichtlich sind und somit Fehlerquellen leichter gefunden werden können.

## Ausgabe

```
tr33@bug:~$go test -timeout 20s ./cmd/
```

# Testanalyse

Bei der Analyse von Tests geht es darum, mithilfe von Tests Aussagen über die Stabilität des Codes zu treffen oder die Aussagen der Tests für andere Programme oder Menschen darzustellen. Die hier aufgeführten Möglichkeiten sind alle eng an das Testing-Tool geknüpft und so ausgewählt. Einen breiten Teil der Testanalyse behandelt das Benchmarking, welches an dieser Stelle vernachlässigt wird. In jedem Abschnitt wird die entsprechende schrittweise Ausführung als *Ausgabe* im Terminal dokumentiert, wobei der *Code* an dieser Stelle nur bei dem *Erkennen von Wettlaufsituationen* zum Verständnis beigelegt wurde.

## Erkennen von Wettlaufsituationen(Race Conditions)

Parallelisierbarkeit ist eine der Stärken von GO und auch Tests in GO sind parallelisierbar. Bei parallelen Anwendungen kann es zu sogenannten Wettlaufsituationen(*Race Conditions*) kommen, die im häufig sehr schwer zu finden sind. Eine solche "Wettlauf"-Situation entsteht immer dann, wenn einzelne Komponenten die parallel laufen, in ihrer Dauer und Reihenfolge nicht definiert sind und somit bei verschiedenen Ausführungen verschiedene Lösungen herauskommen können. Um einen solchen Fehler schnell zu erkennen, kann jedes Programm in GO mit dem *Data Race Detector*<sup>5</sup> ausgeführt werden. Um diesen Detector zu aktivieren, muss die *race*-Flag(*-race*) in dem Kommando enthalten sein. Der *Data Race Detector* kann neben dem Testen auch bei der Ausführung des Programmes (`go run -race main.go`), bei dem Bauen(`go build -race main.go`) und bei dem Installieren(`go install -race main.go`) ausgeführt werden.

Eine einfache Wettlaufbedingung ist, wenn zwei parallele Programme in die gleiche Variable schreiben möchten. Welches Programm später in die Variable schreibt, hat seinen Wert als Lösung in dieser festgehalten, wobei der vorige Wert verloren geht. Um so ein Beispiel zu kreieren, legen wir zunächst eine Variable an, auf welche die parallelen Tests zugreifen können(*superCount*). Im Folgenden werden zwei Funktionen erstellt, wobei eine Funktion (*countUp()*) die Variable um eins erhöht und eine Funktion (*countDown()*) die Variable um eins erniedrigt. Beide Funktionen werden nach dem Schreibvorgang der Variable ihren Wert als Log ausgeben und in der *TestStartRace()*-Funktion als Untertests(*subtests*) und parallel aufgerufen.

Der Test wird zunächst mit `go test -run=TestStartRace$ ./cmd/` ohne den *Data Race Detector* aufgerufen und wird fehlerfrei ausgeführt. Wird der *Data Race Detector* nun hinzugefügt(`go test -race -run=TestStartRace$ ./cmd/`) erscheint sofort eine Data-Race-Warnung. In der Warnung werden dem Entwickler weitere Informationen bereitgestellt, darüber wie die Wettlaufsituation entstanden ist.

## Code

```
// TestStartRace() mit countUp() und countDown() aus ./cmd/bsp6_test.go
var superCount int

func TestStartRace(t *testing.T) {
    // Start subtest
    t.Run("raceDown", func(t *testing.T) {
        t.Parallel()
        countDown(t)
    })
}
```

```
// Start second subtest
t.Run("raceUp", func(t *testing.T) {
    t.Parallel()
    countUp(t)
})

}

func countUp(t *testing.T) {
    superCount++
    t.Log(superCount)
}

func countDown(t *testing.T) {
    superCount--
    t.Log(superCount)
}
```

## Ausgabe

```
tr33@bug:~$go test -run=TestStartRace$ ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    0.297s
tr33@bug:~$go test -race -run=TestStartRace$ ./cmd/
=====
WARNING: DATA RACE
Read at 0x00000144e8e8 by goroutine 9:
  github.com/Tr33Bug/myCli/cmd.countUp()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:26 +0x55
github.com/Tr33Bug/myCli/cmd.TestStartRace.func2()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:20 +0x45
testing.tRunner()
    c:/go/src/testing/testing.go:1123 +0x202

Previous write at 0x00000144e8e8 by goroutine 8:
  github.com/Tr33Bug/myCli/cmd.countDown()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:31 +0x71
github.com/Tr33Bug/myCli/cmd.TestStartRace.func1()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:15 +0x45
testing.tRunner()
    c:/go/src/testing/testing.go:1123 +0x202

Goroutine 9 (running) created at:
  testing.(*T).Run()
    c:/go/src/testing/testing.go:1168 +0x5bb
github.com/Tr33Bug/myCli/cmd.TestStartRace()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:18 +0x93
testing.tRunner()
    c:/go/src/testing/testing.go:1123 +0x202

Goroutine 8 (running) created at:
  testing.(*T).Run()
    c:/go/src/testing/testing.go:1168 +0x5bb
github.com/Tr33Bug/myCli/cmd.TestStartRace()
    C:/Users/Tr33Bug/go/myCli/cmd/bsp6_test.go:13 +0x64
testing.tRunner()
    c:/go/src/testing/testing.go:1123 +0x202
=====
--- FAIL: TestStartRace (0.00s)
    --- FAIL: TestStartRace/raceDown (0.00s)
        bsp7_test.go:32: -1
    --- FAIL: TestStartRace/raceUp (0.00s)
        bsp7_test.go:27: 0
        testing.go:1038: race detected during execution of test
FAIL
FAIL    github.com/Tr33Bug/myCli/cmd    0.047s FAIL
```

# Darstellung der Testergebnisse als JSON

Alle durchgeführten Tests in dieser Arbeit wurden in einer Konsole mit Befehlen ausgeführt und die Ergebnisse auf der Konsole als Text in dem GO-eigenen Format wieder gegeben. Sollen Testergebnisse jedoch in weiteren Schritten von anderen Programmen gelesen werden, so bietet sich das *JSON*-Format an. Um die Testergebnisse in diesem Format dazustellen zu lassen, wird die *json*-Flag dem Testbefehl hinzugefügt (`go test -json main.go`). Um die Ausgabe an dieser Stelle etwas ein zu grenzen, wird nur der Test `TestPrintHello()` aus der Datei `./cmd/bsp2_test.go` ausgeführt.

## Ausgabe

```
tr33@bug:~$go test -json -run=TestPrintHello$ ./cmd/
{"Time":"2021-03-10T15:47:23.5256994+01:00","Action":"run","Package":"github.com/Tr33Bug/myCli/cmd","Test":"TestPrintHello"}
{"Time":"2021-03-10T15:47:23.5542249+01:00","Action":"output","Package":"github.com/Tr33Bug/myCli/cmd","Test":"TestPrintHello","Output":"=== RUN    TestPrintHello\n"}
{"Time":"2021-03-10T15:47:23.5547246+01:00","Action":"output","Package":"github.com/Tr33Bug/myCli/cmd","Test":"TestPrintHello","Output":"--- PASS: TestPrintHello (0.00s)\n"}
{"Time":"2021-03-10T15:47:23.5547246+01:00","Action":"pass","Package":"github.com/Tr33Bug/myCli/cmd","Test":"TestPrintHello","Elapsed":0}
{"Time":"2021-03-10T15:47:23.5547246+01:00","Action":"output","Package":"github.com/Tr33Bug/myCli/cmd","Output":"PASS\n"}
{"Time":"2021-03-10T15:47:23.5552258+01:00","Action":"output","Package":"github.com/Tr33Bug/myCli/cmd","Output":"ok   \tgithub.com/Tr33Bug/myCli/cmd\t0.314s\n"}
{"Time":"2021-03-10T15:47:23.5812483+01:00","Action":"pass","Package":"github.com/Tr33Bug/myCli/cmd","Elapsed":0.34}
```

## Testabdeckung des Codes

Um ein Projekt zu prüfen und zu sehen, wie viel des Quellcodes mit Tests abgedeckt ist, gibt es die *cover*-Flag. Wird die *cover*-Flag dem Kommando hinzugefügt, so wird als Ausgabe eine Prozentzahl gegeben, wie viel des Codes von Tests abgedeckt ist. In diesem Beispiel sind das 30.8 % Testabdeckung.

Um noch ein besseres Verständnis zu bekommen, welche Teile des Codes nicht abgedeckt sind, gibt es die Möglichkeit, sogenannte Coverprofiles die exakte Testabdeckung in dem Code Anzeigen zu lassen. Dafür wird zunächst ein solches Profil mithilfe der *coverprofile*-Flag angelegt und als `myCoverProfile` gespeichert. Auch bei dieser Ausgabe wird als Rückgabe neben der Testdauer und dem Teststatus die *coverage* also die Testabdeckung in Prozent gegeben.

Ein Blick in das neu erstellte Testprofil mit `cat ./myCoverProfile` zeigt in der ersten Zeile einen Modus, der gesetzt wird und in den folgenden Zeilen Informationen zu erstellten Tests mit Zahlen. Um diese Zahlen zu verstehen, wird das `go tool` mit dem Argument `cover` und genutzt. Dieses Tool bereitet das Testprofil als Darstellung auf.

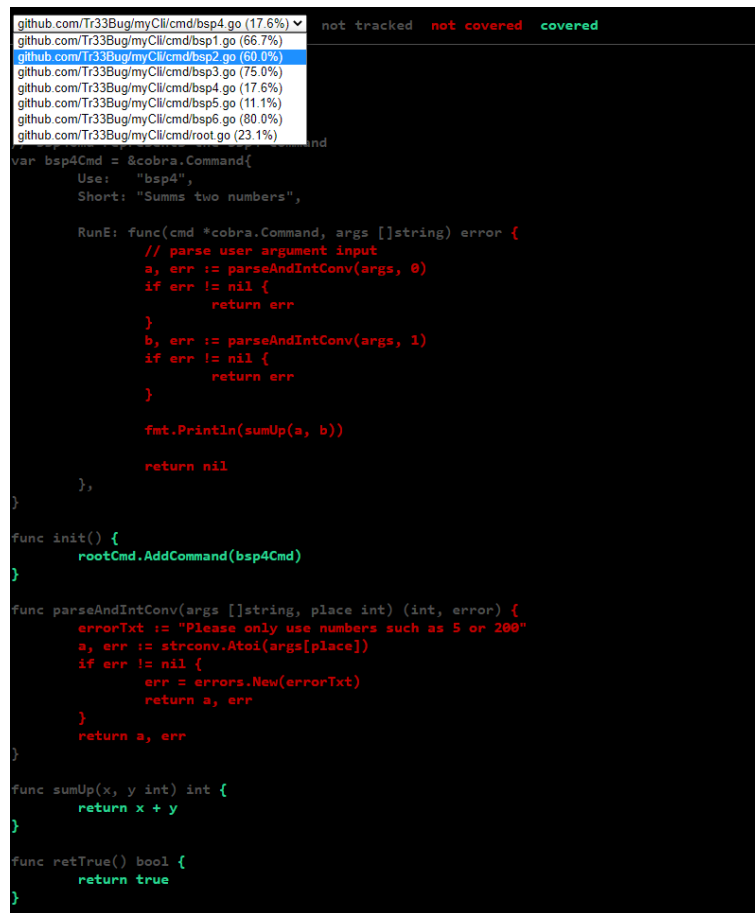
Als erste Möglichkeit der Darstellung soll für jede Funktion gezeigt werden, wie viel Prozent diese von Tests abgedeckt wird. Erreicht wird das mit der *func*-Flag (`go tool cover -func=myCoverProfile`)

Eine noch detaillierte Darstellung der Testabdeckung bietet GO als HTML an. Für diese Darstellung wird wie zuvor das `go tool cover` genutzt, nun aber mit der *html*-Flag. Nach dem Ausführen des `go tool cover -html=myCoverProfile`-Kommandos öffnet sich die HTML-Darstellung des Testprofils in einem Browser. Hier wird der Code der Anwendung gezeigt und je nach Abdeckung farbig angezeigt. Rot steht für nicht abgedeckt und Grün steht für einen bereits abgedeckten Codebereich. Die grauen Codeteile sind im Testfall zu vernachlässigende Codeteile wie Variablendeklaration, Kommentare und Weiteres. Die HTML-Seite gibt weiter die Möglichkeit zwischen den verschiedenen Dateien, in welche das Projekt aufgeteilt ist, mit Hilfe einer Drop-Down-Liste zu navigieren.

## Ausgabe

```
tr33@bug:~$go test -cover ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    9.294s  coverage: 33.9% of statements
tr33@bug:~$go test -coverprofile=myCoverProfile ./cmd/
ok      github.com/Tr33Bug/myCli/cmd    9.295s  coverage: 33.9% of statements
tr33@bug:~$cat ./myCoverProfile
mode: set
github.com/Tr33Bug/myCli/cmd/bsp4.go:31.54,34.17 2 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:37.3,38.17 2 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:42.3,44.13 2 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:34.17,36.4 1 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:38.17,40.4 1 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:48.13,50.2 1 1
github.com/Tr33Bug/myCli/cmd/bsp4.go:52.61,55.16 3 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:59.2,59.15 1 0
github.com/Tr33Bug/myCli/cmd/bsp4.go:55.16,58.3 2 0
...
tr33@bug:~$go tool cover -func=myCoverProfile
github.com/Tr33Bug/myCli/cmd/bsp1.go:34:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp1.go:39:      printHelloWorld 100.0%
github.com/Tr33Bug/myCli/cmd/bsp2.go:34:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp2.go:38:      printHello      100.0%
github.com/Tr33Bug/myCli/cmd/bsp2.go:42:      printWorld      100.0%
github.com/Tr33Bug/myCli/cmd/bsp3.go:34:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp3.go:38:      waitAndPrint    100.0%
github.com/Tr33Bug/myCli/cmd/bsp4.go:48:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp4.go:52:      parseAndIntConv 0.0%
github.com/Tr33Bug/myCli/cmd/bsp4.go:62:      sumUp           100.0%
github.com/Tr33Bug/myCli/cmd/bsp4.go:66:      retTrue         100.0%
github.com/Tr33Bug/myCli/cmd/bsp7.go:38:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp7.go:42:      startRace       0.0%
github.com/Tr33Bug/myCli/cmd/bsp7.go:48:      addFive         0.0%
github.com/Tr33Bug/myCli/cmd/bsp7.go:52:      addOne          0.0%
github.com/Tr33Bug/myCli/cmd/bsp5.go:36:      init           100.0%
github.com/Tr33Bug/myCli/cmd/bsp5.go:40:      randomReturnHello 100.0%
github.com/Tr33Bug/myCli/cmd/root.go:39:      Execute         0.0%
github.com/Tr33Bug/myCli/cmd/root.go:43:      init           100.0%
github.com/Tr33Bug/myCli/cmd/root.go:58:      initConfig      0.0%
total:                                     (statements) 47.5%
tr33@bug:~$go tool cover -html=myCoverProfile
--> open Browser, siehe "Browser Darstellung der Testabdeckung"
```

# Browser Darstellung der Testabdeckung



The screenshot shows a web browser interface. At the top, a dropdown menu is open, displaying a list of GitHub repository links and their corresponding test coverage percentages. The links are as follows:

- github.com/Tr33Bug/myCli/cmd/bsp4.go (17.6%)
- github.com/Tr33Bug/myCli/cmd/bsp1.go (66.7%)
- github.com/Tr33Bug/myCli/cmd/bsp2.go (60.0%)
- github.com/Tr33Bug/myCli/cmd/bsp3.go (75.0%)
- github.com/Tr33Bug/myCli/cmd/bsp4.go (17.6%)
- github.com/Tr33Bug/myCli/cmd/bsp5.go (11.1%)
- github.com/Tr33Bug/myCli/cmd/bsp6.go (80.0%)
- github.com/Tr33Bug/myCli/cmd/root.go (23.1%)

To the right of the dropdown, the text "not tracked" is displayed in grey, "not covered" in red, and "covered" in green. Below the dropdown, a code editor displays Go source code. The code defines a cobra command and its execution logic. The code is as follows:

```
var bsp4Cmd = &cobra.Command{
    Use: "bsp4",
    Short: "Summs two numbers",

    RunE: func(cmd *cobra.Command, args []string) error {
        // parse user argument input
        a, err := parseAndIntConv(args, 0)
        if err != nil {
            return err
        }
        b, err := parseAndIntConv(args, 1)
        if err != nil {
            return err
        }

        fmt.Println(sumUp(a, b))

        return nil
    },
}

func init() {
    rootCmd.AddCommand(bsp4Cmd)
}

func parseAndIntConv(args []string, place int) (int, error) {
    errorTxt := "Please only use numbers such as 5 or 200"
    a, err := strconv.Atoi(args[place])
    if err != nil {
        err = errors.New(errorTxt)
        return a, err
    }
    return a, err
}

func sumUp(x, y int) int {
    return x + y
}

func retTrue() bool {
    return true
}
```



# Fazit und Ausblick

Im Vergleich zu anderen Programmiersprachen bietet GO als relativ junge Programmiersprache viele spannende und einfach zu nutzende Tools, die nicht an eine grafische Oberfläche oder Entwicklungsumgebung gebunden sind. Das bietet den Vorteil, dass diese sehr einfach auf verschiedenen Plattformen genutzt werden kann, bringt aber auch mit sich, dass Entwickler sich mit etwas mit der Kommandozeile auseinandersetzen sollten. Wird der Einsatzbereich der Sprache betrachtet (Serveranwendungen und CLIs), so passt sich die Sprache in dieser Hinsicht Entwicklern und Serveradministratoren, deren gängiges Mittel die Kommandozeile ist, sehr gut an.

Für unerfahrene Programmierer bietet die Sprache und ihre Tools eine sehr gute Einstiegsmöglichkeit und für fortgeschrittene eine Möglichkeit, in einfachen Codezeilen komplexe Dinge darzustellen. Dieser Ansatz kann hier analog für die Unit-Tests und Analysen übernommen werden. Dieser Ansatz ermöglicht es Entwicklern, die vorher keine oder wenige Tests geschrieben haben, schnell und einfach in der Thematik Fuß zu fassen und unterstützt so in der Erstellung von stabilen und sicheren Anwendungen.

Die in diesem Bericht gezeigten Möglichkeiten sind als Einstieg in die Thematik zu verstehen und sollen eine gute Grundlage vermitteln. Um das Thema weiter zu vertiefen, werden die für diese Arbeit verwendeten Quellen<sup>6,7</sup>, die GO-Dokumentationen<sup>8,9</sup> und Vorträge der GopherCon<sup>10</sup> empfohlen.

- 
1. Das Repository, in welchem das gesamte Command Line Interface zu finden ist: <https://github.com/Tr33Bug/myCli> [↪](#) [↪](#) [↪](#)
  2. Das Repository, in welchem das Cobra-Tool zur Erstellung von Command Line Interfaces zu finden ist: <https://github.com/spf13/cobra> [↪](#) [↪](#)
  3. Visual Studio Code Extension für GO: <https://code.visualstudio.com/docs/languages/go> [↪](#)
  4. Die Dokumentation zum GO-Time Paket mit dem der ParseDuration-Funktion <https://golang.org/pkg/time/#ParseDuration> [↪](#)
  5. Die Dokumentation zum GO-Data-Race-Detector Paket: [https://golang.org/doc/articles/race\\_detector](https://golang.org/doc/articles/race_detector) [↪](#)
  6. Alex Edwards Blogbeitrag über die verschiedenen Tools, die GO mitbringt: <https://www.alexedwards.net/blog/an-overview-of-go-tooling> [↪](#)
  7. Einführung in die GO-Programmiersprache: <https://entwickler.de/online/development/einfuehrung-programmierung-go-166821.html> [↪](#)
  8. Die Dokumentation der GO-Sprache: <https://golang.org/doc/> [↪](#)
  9. Die Dokumentation zum GO-Testing Paket: <https://golang.org/pkg/testing/> [↪](#)
  10. Die Homepage der Jährlich stattfindenden GO Konferenz: <https://www.gophercon.com/> [↪](#)