Plan/Proof of Concept: Go Exercise

JI YOON OK, jyo1031@gmail.com
BENJAMIN GEE, bmgee27@gmail.com
CHRISTINE SAM, xtinesam@gmail.com

---

1 PROJECT OVERVIEW

1.1 Purpose

Using Go, we can take advantage of goroutines and channels for concurrency to create an exercise interval timer program. The goal of our program is to exploit and showcase the power and ease of creating a concurrent program using goroutines and channels.

1.2 Project Description and Technical Elements

In our exercise timer program, the user has the ability to choose a number of circuits. For each circuit, they can set the amount of time to exercise, as well as the amount of short breaks in between each exercise. Each circuit will essentially consist of a number of exercise and rest intervals, with the number of intervals being set by the user. In between circuits, the user can set a time for a longer break. Additionally, if the user chooses, they can include a warmup and cool down time at the beginning and end respectively. Finally, the application will allow the user to pause and resume the the timer at any point. Figure 1 below shows an example of what the timer could look like. The technical elements of our projects that utilize the concurrency features of Go are the use of goroutines to create threads for specific timers and channels to communicate between timers.

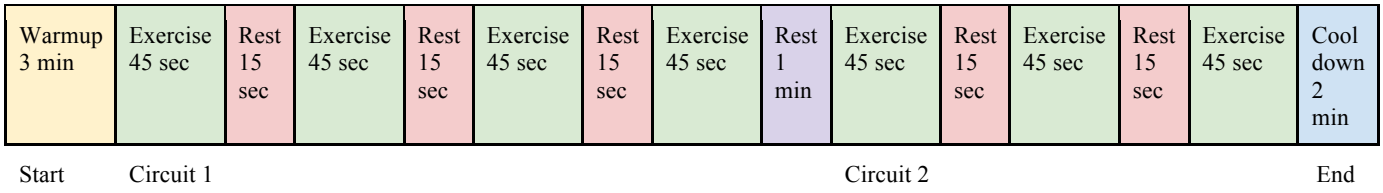

Figure 1. Sample timer program. This is an example of a timer program that can be created in our application. In this specific timer, a user has created two different circuits with a 1minute rest in between, as well as a warmup and cool down time.

2 TECHNICAL ABILITIES NEED FOR COMPLETION OF THE PROJECT

2.1 General Go Syntax

There are several technical skills and knowledge needed for the successful completion of the project. First of all, we must familiarize ourselves with the Go syntax. The syntax of Go is overall very similar to C, but uses less parenthesis and no semicolons [1, 2]. For example, Figure 2 shows a side to side comparison of a "Hello world" program in Go and in C.

```
package main

import "fmt"

func main() {
   fmt.Println("Hello World")
}
```

```
#include <stdio.h>

int main()
{
  printf("Hello, World!");
  return 0;
}
```

Figure 2. Comparison of Go and C syntax. On the left is a "Hello World" program in Go and on the right is a "Hello World" program in C.

There are also many other syntactic differences between Go and C, but only a few interesting and important ones are highlighted in Figure 3 below.

1. Type declarations go after variable identifiers.	`var foo int = 1`
	`var foo, bar int = 1, 2`
	`var a [10]int`

2. Multiple return values are allowed.	`func foobar() (int, string) {`
	`  return 1, "foobar"`
	`}`

3. Despite being called an object oriented language, there are no classes, only structs like C.	`type foobar struct {`
	`  A, B int`
	`}`

4. There is no exception handling. Functions instead can produce an extra 'error' as a return value.	`func main() {`
	`  result, error := foobar()`
	`  if (error != nil) {`
	`  // handle error`
	`  }`
	`}`

5. Go uses pointers like C, but lacks pointer arithmetic.	`p := 5` `q := &p`

Figure 3. Interesting Syntactic Features of Go. The table above summarizes a few interesting features of Go. Information from the table was referenced from [2].

2.1 Understanding How to use Goroutines and Channels

Given a function `print(s)` that will print parameter s to the console. In a synchronous program, the function `print` is invoked by simply calling the function as follows:

```
package main

import "fmt"

func print(s string) {
     fmt.Println(string)
}

func main() {
     print("Hello World")
}

// prints Hello World
```

To invoke concurrent functions in our program, a function needs to be executed in a goroutine. By including the prefix `go` to the call of a function will effectively execute that function concurrently. Goroutines can be invoked on both named and anonymous functions.

```
package main

import "fmt"

func namedPrint(s string) {
     for i := 0; i < 3; i++ {
          fmt.Println(s, i)
     }
}

func main() {

     go namedPrint("Hello World")

     go func anonymousPrint(s string) {
          for i := 0; i < 3; i++ {
               fmt.Println(s, i)
          }
     }("Bye World")
}

// Hello World 0
// Hello World 1
// Bye World 0
// Hello World 2
// Bye World 1
// Bye World 2
```

In the above example, two goroutines are created, one on the named function `namedPrint` and one on the anonymous function `anonymousPrint`. In a synchronous program, when we invoke the call to `namedPrint`, it would execute the entire body of the function to completion before proceeding to the next line. However, when namedPrint is invoked as a concurrent function, as soon as it is executed the program continues to the next line and does not wait for the function to complete. This explains why the output messages are not in sequential order between `"Hello World"` and `"Bye World"`.

To communicate between two concurrent goroutines, channels are used to receive information from one goroutine and deliver it to another. A channel is created by invoking `make(chan value-type)` function as follows:

```
package main

import "fmt"

func main() {
     myChannel := make(chan string)
}

// myChannel is a channel that accepts strings as an input
```

To deliver a message within a goroutine into a channel, invoking the following code `channel <- message` will provide the channel with a message. To receive the message simply assign the output of the channel to a variable, `variable := <-channel`.

```
package main

import "fmt"

func main() {
     myChannel := make(chan string)

     go func() {
            myChannel <- "Hello World"
     }

     message := <-myChannel
     fmt.Println(message)
}

// prints Hello World
```

3 LOW RISK APPROACH

3.1 Core Goals

The core goal of our project is to create a simple console timer program with a default exercise interval time, short rest time and long rest time. For our low risk approach to meet the team's project goals, we aim to complete the following tasks:

1. Create a console program in Go
2. When main function is invoked, the timer is started
3. Timer alerts the user when exercise or short/long rest intervals begin
4. Timer runs for the default length of time (i.e. 3 minutes for exercise, 1 minute for short rest and 2 minutes for long rest) for default number of repetitions
5. Timer alerts the user when exercise or short/long rest interval ends
6. Timer asks the user at the end of the exercise program if he/she wants to continue with another program
7. Program exits upon close/quit

3.2 Code – Simple Exercise Timer

```
package main

import (
   "fmt"
   "time"
   "os/exec"
)

var(
   currentExercise = 1
   allExercises =  5
)

func main() {
   exercise := make(chan bool)
   shortRest := make(chan bool)
   longRest := make(chan bool)
   done := make(chan bool)

   go circuit(exercise)
   go circuitService(exercise, shortRest, longRest, done)

   <-done
}

func circuit(exerciseChan chan bool) {
   beginExercise()
```

```
   time.Sleep(time.Minute*3)
   endExercise()
   exerciseChan <- true
}

func shortRest(restChan chan bool) {
   beginShortRest()
   time.Sleep(time.Minute)
   endShortRest()
   restChan <- true
}

func longRest(longRestChan chan bool) {
   beginLongRest()
   time.Sleep(time.Minute*2)
   longRestChan <- true
}

func beginExercise() {
   exec.Command("say", "Exercise begins").Output()
}

func endExercise() {
   exec.Command("say", "Exercise ends").Output()
}

func beginShortRest() {
   exec.Command("say", "Short rest begins").Output()
}

func endShortRest() {
   exec.Command("say", "Short rest ends").Output()
}

func beginLongRest() {
   exec.Command("say", "Long rest begins").Output()
}

func endLongRest() {
   exec.Command("say", "Long rest ends").Output()
}

func circuitService(exerciseChan, shortRestChan, longRestChan, doneChan chan bool) {
   for {
       select {

       case endExercise := <-exerciseChan:
           _ = endExercise
           if currentExercise >= allExercises {
               go longRest(longRestChan)
               currentExercise = 1
           } else {
```

```
                currentExercise += 1
                go shortRest(shortRestChan)
            }

        case endShortRest := <-shortRestChan:
            _ = endShortRest
            go circuit(exerciseChan)

        case endLongRest := <-longRestChan:
            _ = endLongRest
            input := askUser()
            for input != "Y" && input != "N" {
                input = askUser()
            }
            if input == "Y" {
                go circuit(exerciseChan)
            } else {
                doneChan <- true
            }

        }
    }
}

func askUser() string {
    fmt.Println("Would you like to continue with another circuit? (Y/N)")
    var response string
    fmt.Scanln(&response)
    return response
}
```

We will launch a goroutine for each of the following events: main program, exercise, short break, long break and circuit service manager. The main program goroutine is implicitly launched at the start of the entire program. Exercise interval begins immediately upon start of the program. It will run for the default time (3 minutes). When exercise goroutine is finished, it will message the circuit service manager goroutine, and short rest will begin. When short rest ends after 1 minute, it again messages the service manager, which launches the exercise goroutine again. At the end of specified number of exercise circuits (5 times), control returns to the main goroutine where it will wait for the user to end the program or start again. When the user is ready to start again, control transfers back to the previous goroutine (where the user left off). The circuit manager goroutine will be a special goroutine that will act as a coordinator to handle messages from the goroutines.

Once we have our goroutines, we then need channels for the separate goroutines to communicate with each other. We'll have one channel each for exercise, short rest and long rest, and done flag. We will be using unbuffered channels, which means the goroutines will be blocked until the message is received. Other than the main and the circuit service manager goroutine, every other goroutine will have its own channel. Go allows different types of channels (boolean, string, int, etc.). For our

purposes, having the channels as boolean values would be sufficient as we only need to signify whether the goroutine is “done” its turn.

4 HIGH RISK APPROACH

4.1 Full Goals

For our high risk approach to meet the team’s project goals, we aim to complete the following:

1. Create a program in go with user friendly interface. Because Go is mainly used for server side coding, we may have to find a cross platform library to create a graphical user interface.
2. User can put number of exercise repetitions, and length of exercise/short rest/long rest. This would involve being able to accept user input for these variables and creating the number of goroutines based on these variables.
3. User can opt to add warmup and cool-down. This will simply be two additional goroutines that will be created if the user chooses to.
4. Have a start button is pressed to begin exercise timer.
5. Timer displays an alert when exercise or short/long rest interval begins.
6. Timer shows how much time is left in current interval.
7. Timer displays an alert when exercise or short/long rest interval ends.
8. User can pause/resume with a button. This can be done by having an additional goroutine that will be switched to when the user chooses to pause the timer. When the user wants to resume, we can simple return back to the previous goroutine.
9. User can reset/quit the timer at any time.

REFERENCES

[1] Tutorials Point. 2017. Go Tutorial. Retrieved from https://www.tutorialspoint.com/go/index.htm
[2] Ariel Mashraki. 2017. Go Cheat Sheet. Retrieved from https://github.com/a8m/go-lang-cheat-sheet