



## ЛЕКЦИЯ 10

TDD. Профилирование и бенчмарки. Метрики

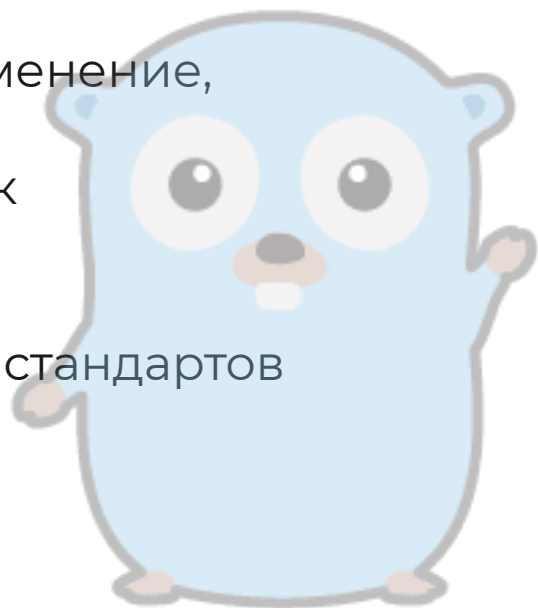
# Test-driven development

## Разработка через тестирование:

Техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки:

- 1) сначала пишется тест, покрывающий желаемое изменение,
- 2) затем пишется код, который позволит пройти тест,
- 3) и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Разумеется, к тестам применяются те же требования стандартов кодирования, что и к основному коду.



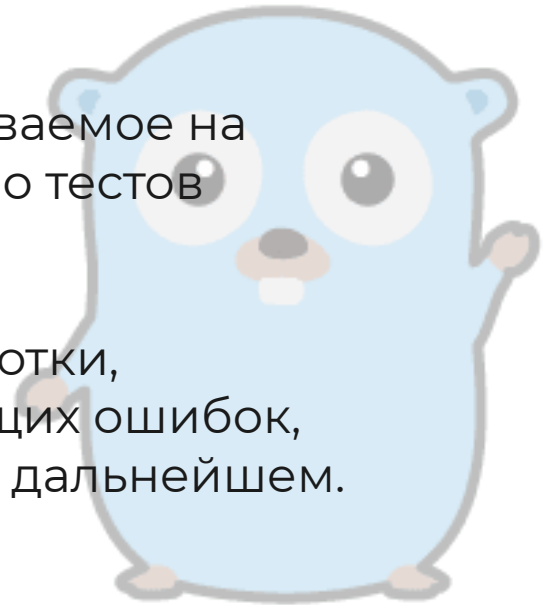
# Test-driven development

## Разработка через тестирование:

Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше.

Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно. Большое количество тестов помогает уменьшить количество ошибок в коде.

Устранение дефектов на более раннем этапе разработки, препятствует появлению хронических и дорогостоящих ошибок, приводящих к длительной и утомительной отладке в дальнейшем.



# Test-driven development

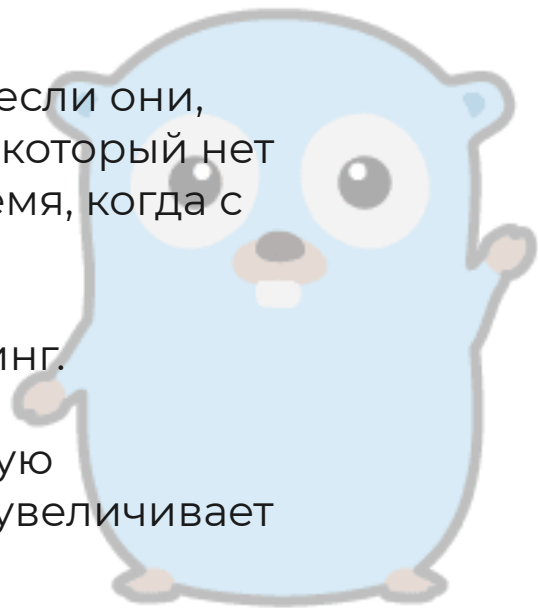
## Разработка через тестирование:

Тесты позволяют производить рефакторинг кода без риска его испортить. При внесении изменений в хорошо протестированный код риск появления новых ошибок значительно ниже.

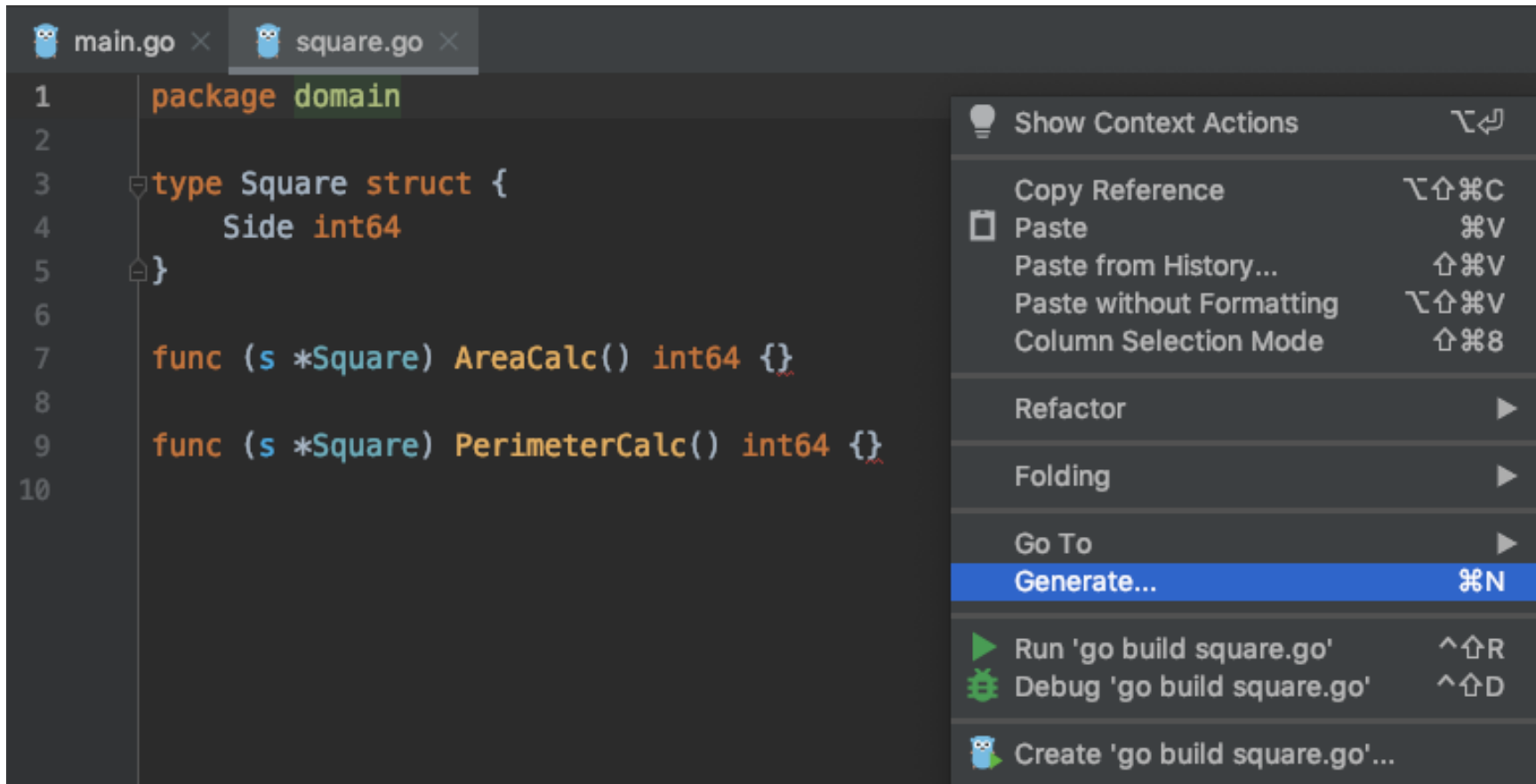
Если новая функциональность приводит к ошибкам, тесты, если они, конечно, есть, сразу же это покажут. При работе с кодом, на который нет тестов, ошибку можно обнаружить спустя значительное время, когда с кодом работать будет намного сложнее.

Хорошо протестированный код легко переносит рефакторинг.

Уверенность в том, что изменения не нарушат существующую функциональность, придает уверенность разработчикам и увеличивает эффективность их работы.



# Test-driven development



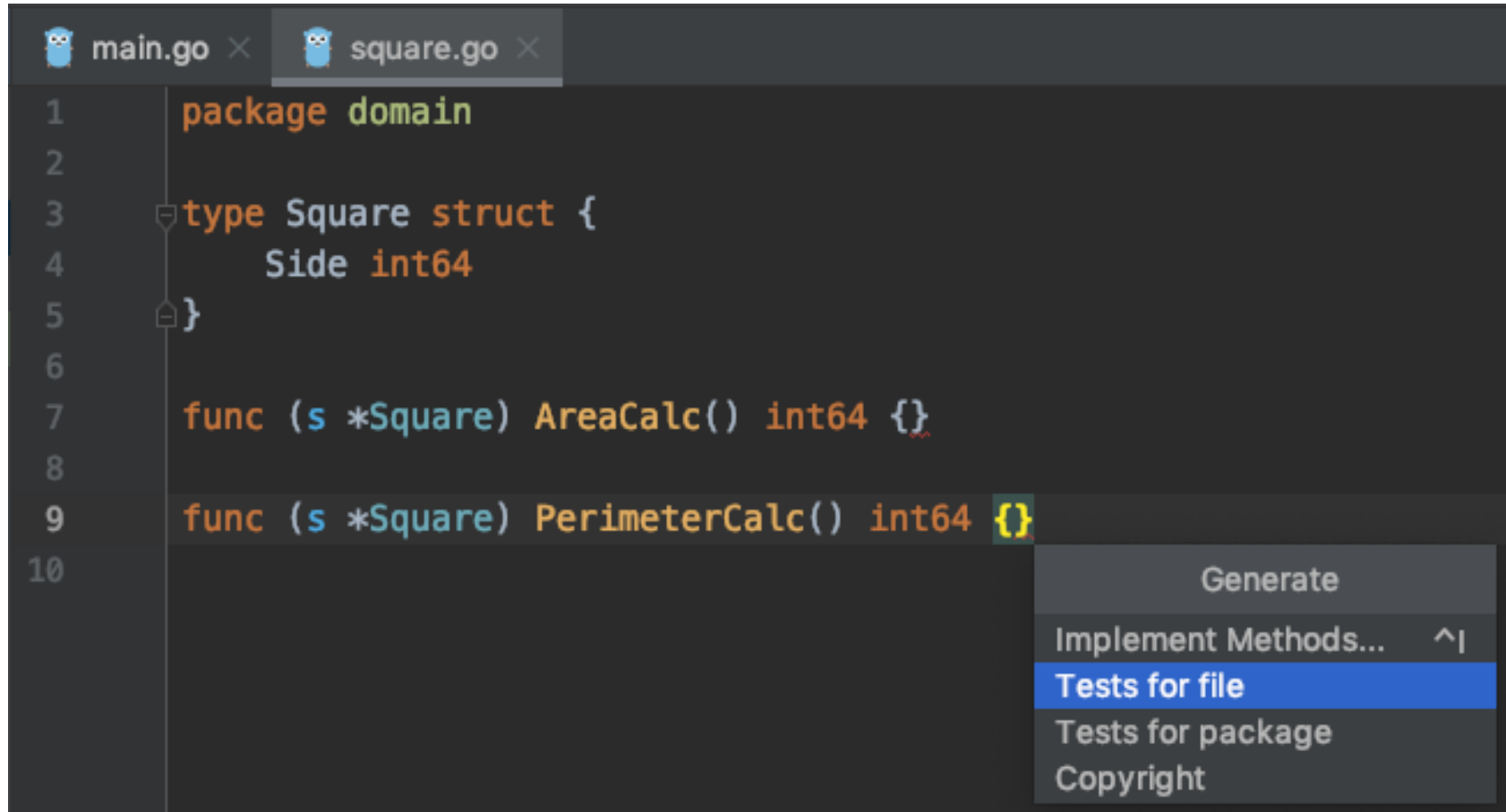
The image shows a code editor with two tabs: `main.go` and `square.go`. The `square.go` tab is active, displaying the following Go code:

```
1 package domain
2
3 type Square struct {
4     Side int64
5 }
6
7 func (s *Square) AreaCalc() int64 {}
8
9 func (s *Square) PerimeterCalc() int64 {}
10
```

A context menu is open on the right side of the editor, listing various actions:

- Show Context Actions (⌘⇧⌘)
- Copy Reference (⌘⇧⌘C)
- Paste (⌘V)
- Paste from History... (⌘⇧⌘V)
- Paste without Formatting (⌘⇧⌘V)
- Column Selection Mode (⌘⇧⌘8)
- Refactor (▶)
- Folding (▶)
- Go To (▶)
- Generate... (⌘N)**
- Run 'go build square.go' (⌘⇧R)
- Debug 'go build square.go' (⌘⇧D)
- Create 'go build square.go'...

# Test-driven development



The image shows a code editor with two tabs: `main.go` and `square.go`. The `square.go` tab is active, displaying the following Go code:

```
1 package domain
2
3 type Square struct {
4     Side int64
5 }
6
7 func (s *Square) AreaCalc() int64 {}
8
9 func (s *Square) PerimeterCalc() int64 {}
10
```

A context menu is open over the `PerimeterCalc()` function, showing the following options:

- Generate
- Implement Methods... ^|
- Tests for file
- Tests for package
- Copyright

# Test-driven development

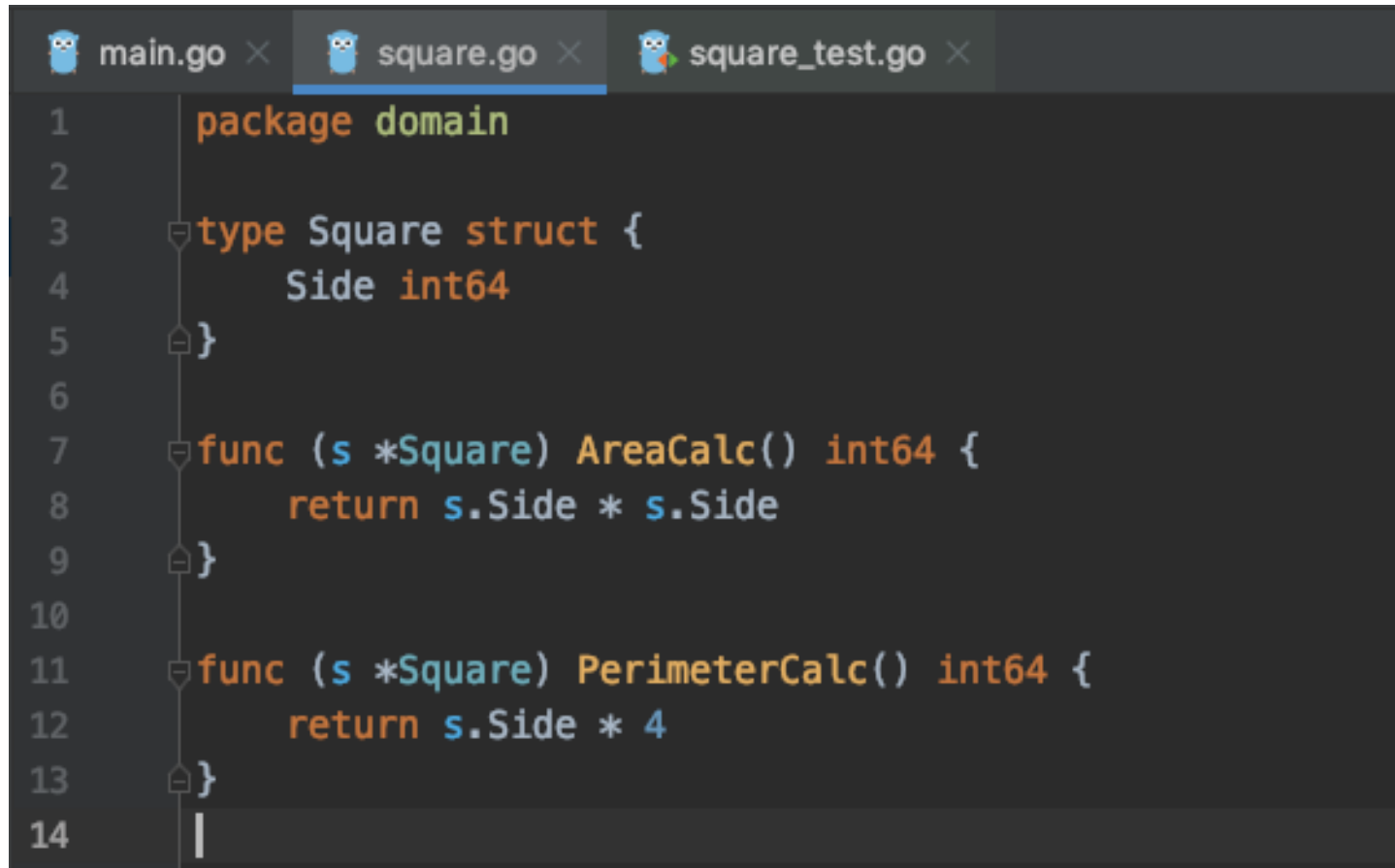
```
main.go x square.go x square_test.go x
1 package domain
2
3 import "testing"
4
5 func TestSquare_AreaCalc(t *testing.T) {
6     type fields struct {
7         Side int64
8     }
9     tests := []struct {
10         name string
11         fields fields
12         want int64
13     }{
14         // TODO: Add test cases.
15     }
16     for _, tt := range tests {
17         t.Run(tt.name, func(t *testing.T) {
18             s := &Square{
19                 Side: tt.fields.Side,
20             }
21             if got := s.AreaCalc(); got != tt.want {
22                 t.Errorf("AreaCalc() = %v, want %v", got, tt.want)
23             }
24         })
25     }
26 }
```

# Test-driven development

```
14      {
15          name:  "AreaTest1",
16          fields: fields{
17              Side:0,
18          },
19          want:  0,
20      },
21      {
22          name:  "AreaTest2",
23          fields: fields{
24              Side:10,
25          },
26          want:  100,
27      },
```



# Test-driven development



The image shows a code editor with three tabs: `main.go`, `square.go` (selected), and `square_test.go`. The `square.go` file contains the following Go code:

```
1 package domain
2
3 type Square struct {
4     Side int64
5 }
6
7 func (s *Square) AreaCalc() int64 {
8     return s.Side * s.Side
9 }
10
11 func (s *Square) PerimeterCalc() int64 {
12     return s.Side * 4
13 }
14 |
```

# Test-driven development

```
✓ Tests passed: 6 of 6 tests – 0 ms
[4] <4 go setup calls>
=== RUN   TestSquare_AreaCalc
--- PASS: TestSquare_AreaCalc (0.00s)
=== RUN   TestSquare_AreaCalc/AreaTest1
--- PASS: TestSquare_AreaCalc/AreaTest1 (0.00s)
=== RUN   TestSquare_AreaCalc/AreaTest2
--- PASS: TestSquare_AreaCalc/AreaTest2 (0.00s)
=== RUN   TestSquare_PerimeterCalc
--- PASS: TestSquare_PerimeterCalc (0.00s)
=== RUN   TestSquare_PerimeterCalc/PerimeterTest1
--- PASS: TestSquare_PerimeterCalc/PerimeterTest1 (0.00s)
=== RUN   TestSquare_PerimeterCalc/PerimeterTest2
--- PASS: TestSquare_PerimeterCalc/PerimeterTest2 (0.00s)
PASS
Process finished with exit code 0
```

**Примечание:** командой `go test ./...` можно запустить тестирование проекта с учетом тестов во всех вложенных директориях проекта.

# Профилирование

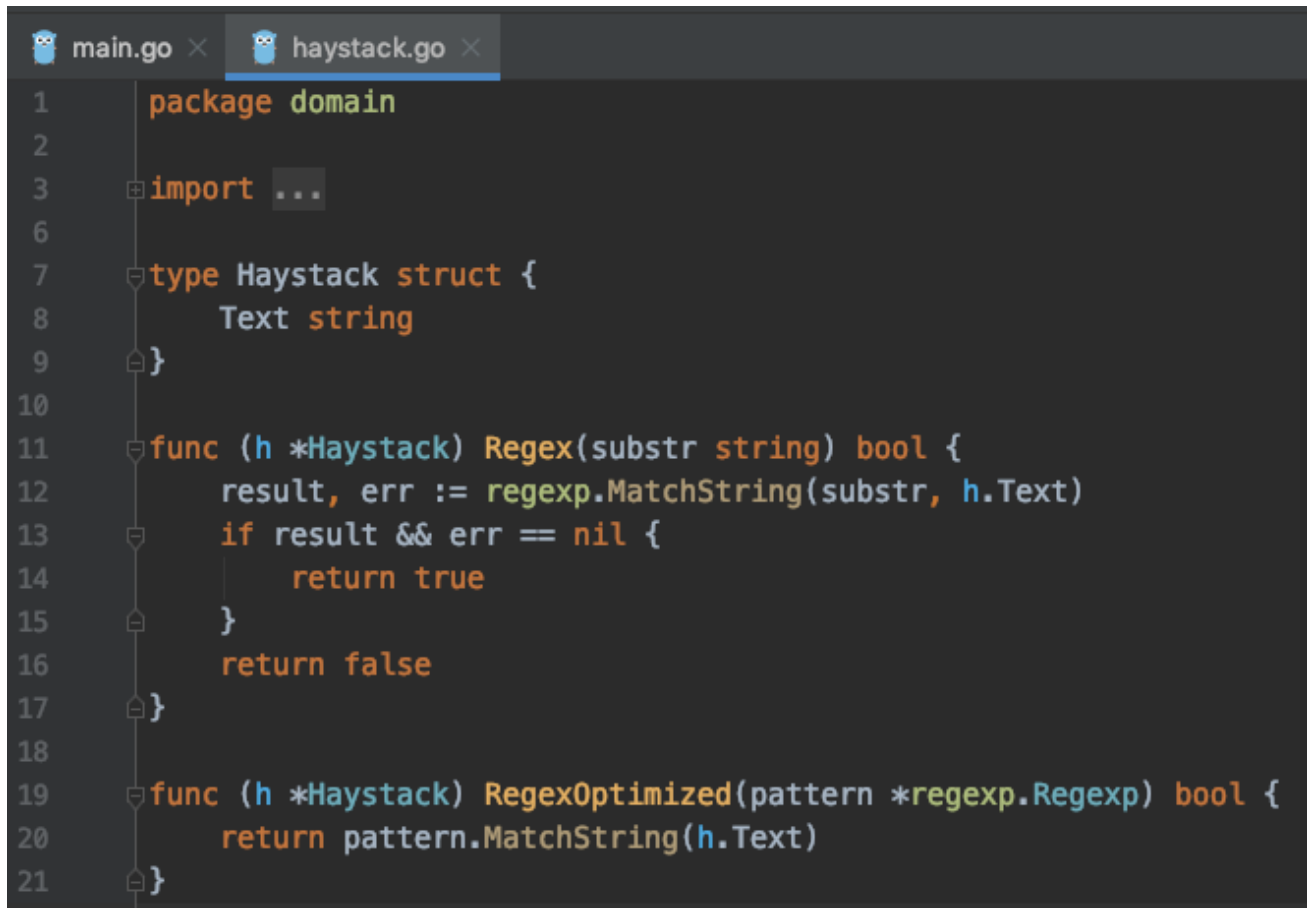
## Для чего нужен профайлинг?

Если ваша программа работает недостаточно быстро, использует слишком много памяти, неоптимально использует процессор, вы хотите понять, в чем дело, и исправить — это и есть профайлинг и оптимизация.

Для начала выполним профилирование работы процессора.



# Профилирование



```
1 package domain
2
3 import ...
4
5
6
7 type Haystack struct {
8     Text string
9 }
10
11 func (h *Haystack) Regex(substr string) bool {
12     result, err := regexp.MatchString(substr, h.Text)
13     if result && err == nil {
14         return true
15     }
16     return false
17 }
18
19 func (h *Haystack) RegexOptimized(pattern *regexp.Regexp) bool {
20     return pattern.MatchString(h.Text)
21 }
```

# Профилирование

```
main.go x haystack.go x haystack_bench_test.go x
1  package domain
2
3  import ...
7
8  var haystack = `Lorem ipsum dolor sit amet, consectetur adipiscing
9  Nullam maximus odio vitae augue fermentum laoreet eget scelerisque
10
11 func BenchmarkRegex(b *testing.B) {
12     haystack := &Haystack{Text:haystack}
13     for i := 0; i < b.N; i++ {
14         haystack.Regex( substr: "auctor")
15     }
16 }
17
18 func BenchmarkRegexOptimized(b *testing.B) {
19     haystack := &Haystack{Text:haystack}
20     pattern := regexp.MustCompile( str: "auctor")
21     for i := 0; i < b.N; i++ {
22         haystack.RegexOptimized(pattern)
23     }
24 }
```

# Профилирование

Далее выполняем команды (предварительно закомментируем функцию `RegexOptimized`):

```
// В директории src/domain
```

```
GOGC=off go test -bench=. -cpuprofile cpu.out  
go tool pprof domain.test cpu.out  
(pprof) png
```

Затем комментируем функцию `Regex` и раскомментируем `RegexOptimized`.

Повторяем команды и получим два png изображения от профилировщика.

