Tallinna Ülikool

Digitehnoloogiate Instituut

Go programmeerimiskeele rakendused

Seminaritöö

Autor: Richard Aasa

Juhendaja: Jaagup Kippar

Autor: ..... „ ..... „2017

Juhendaja: ..... „ ..... „2017

Instituudi direktor: ..... „ ..... „2017

Tallinna 2017

Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

.....	.....
(kuupäev)	(autor)

Sisukord

Sissejuhatus

Käesolev teema on valitud autori isiklikust huvist. Antud keele vastu on globaalne huvi märgatavalt tõusnud viimaste aastate jooksul (lisa 1), aina enam mainitakse antud keelt foorumites ja blogipostitustes programmeerimise valla ekspertide poolt.

Keel on loodud Google tiimi poolt 2007, avalikustatud 2009 ja versioon 1.0 avalikustatud 2012. Algses tiimis olid: Robert Griesemer, Rob Pike ja Ken Thompson (Go (programming language), 2009).

Seminaritöö eesmärk on tutvustada keelt, leida põhjused, miks mainekamad ettevõtted ja eksperdid valisid projektides just Go keele ja läbi viia ühe programmiliidese (API) loomise dokumenteerides protsessi ja läbides mitu iteratsiooni. Autor toob välja ka erinevused ja sarnasused Java keelega, kuna on seda keelt kõige enam kasutanud tööalaselt.

Autori eesmärk on tekitada huvi lugejas Go keele vastu ja anda ka praktilisi näiteid loodava programmiliidese näol. Töö on väga sisutihe tehniliste terminitega ja vajab teadmisi kuidas programmeerimiskeeled töötavad süsteemi- ja projektitasemel, et täielikult kaasa mõelda.

1. Mis on Go(lang) programmeerimiskeel?

Keel loodi eesmärgiga vähendada sõnarohkust ja vältida kasutamast moodsamate keelte võimalusi nagu meetodi ja operaatori ülekirjutamist (*overloading*), *generics*, mäluviite aritmeetika. Keel on kiidetud oma prügikorjaja (*garbage collector*) ja mäluhalduse (*memory management*) poolest. Keelel on kaks nimetust tekkinud. Ametlik nimetus on Go, aga kuna seda on raske otsida, hakati laialdaselt viitavama keelele nimetusega Golang.

1.1. Kompileerimine

Keel vaikimisi kasutab kompileerimiseks *GNU Compiler Collection*. GCC toetab x86(32-bit ja 64-bit) ja ARM protsessoreid. Go pakub ka oma loodud GCC liidest GoGCC mis võimaldab kompileerida kõikidele kiibistike arhitektuuridele (SPARC, MIPS, PowerPC) (Taylor, 2012). GoGCC võimaldab kompileerimist ka Solaris operatsioonisüsteemil, GCC seda ei võimalda. Siin peaks mainima, et Go kompileerib otse masinkoodi, erinevalt näiteks Java keelest mis kompileerib Java baitkoodi, mis seejärel käivitatakse selle käivitamiseks optimeeritud JVM (Java Virtual Machine) peal. Sellepärast on Java rakenduste puhul käivitamiseks vaja rakendusi nagu näiteks JRE (Java Runtime Environment). Go puhul see ei ole vajalik. Lisaks veel tänu GCC kasutusele on võimalik kasutada C keele teeke Go koodis (Krennmair).

1.2. Loodud rakenduse käivitamine

Üks binaarfail kompileerimisel. Niikaua kui kompileeritakse suunatud platvormi peal (näiteks Windows) saab rakendust käivitada ühe binaarfaili kaudu (Windows puhul .exe). Java tavaliselt käivitatakse *source-deploy* metoodika alusel, kuigi on ka võimalik ühe faili kaudu pakkides kõik kompileeritud klassid .jar faili. Selle käivitamiseks on aga ikkagi vaja JVM olemasolu.

1.3. Tagasiulatuv toetus

Kood loodud Go versioon 1.0 (aastal 2012) alusel töötab ja kompileerub ka Go versioon 1.9 (aastal 2017) kasutades. See väike pisiasi võib säästa ettevõttel meeletult resursse. Java puhul kui võtta vastu projekt mis hõlmab kaasajastumist on mõttekoht alati Java JDK(*Java Development Kit*) versioon. Kuigi väidetakse, et rakendused on kergesti uuendatavad näiteks versioon 1.6 pealt 1.8 peale, pole see täielikult tõene. JDK 1.6 kompileeritud kood jookseb JDK 1.8 peal, aga algkood ise ei pruugi kompileeruda JDK 1.8 peal. Näiteks 1.8 JDK versioon nõuab, et kui klass A viitab klass B peale, mis omakorda implementeerib liidest C, siis nii klass B kui ka C peavad kajastuma *class path* sees. 1.6 puhul ei olnud sellist nõuet, see tähendab, et kõik klass B sarnased klassid mille liides ei kajastu *class path* sees ei kompileeru edukalt. See on vaid üks näide, selliseid ebakõlasid on veel (JDK 1.8/1.7 Compatibility Gotcha) ning kui koodibaas on märgatavalt suur (üle miljoni rea) või hõlmab endas keerulist arhitektuuri tähendab see märgatavat ajakulu.

1.4. Standardteekide kogu

Go pakub väga hästi kirjutatud standardteeke ja dokumentatsiooni. Java puhul on tihti raske aru saada kuidas mõnda meetodit või klassi implementeerida. Terved teegid on loodud Java maailmas sellepärast, et standardteek on kas üleehitatud või ei paku elementaarset eeldatud funktsionaalsust. Näiteks HTTP GET päring standardteegiga hõlmab endas kolme klassi kasutust (URLConnection, OutputStream, InputStream) ja ~10 rida koodi ilma vea haldamiseta. Go puhul piisab neljast reast (vt. koodinäide 1).

```
// Loome kliendi päringu tegemiseks
client := &http.Client{}

// Loome päringu ja lisame veel lisaks mis tüüpi vastust aktsepteerida.
// Kasutan siin Star Wars programmeerimisliidest näite eesmärgil.
// Iga funktsioon tagastab alati vastuse JA vea.
// Kuna meid vea olemasolu ei huvita
// muudame teise muutuja "_" peale.

request, _ := http.NewRequest("GET", "https://swapi.co/api/starships", nil)
request.Header.Add("Accept", "application/json")

// := operaator määrab automaatselt vastuse tüübi vasakul pool olevale
// muutujale.
// Kuna parem pool on String tüüpi (päringu vastus) on ka response String
// tüüpi.
// response muutuja omab nüüd terve päringu vastust.
response, _ := client.Do(request)
```

Koodinäide 1. Go HTTP GET päring

1.5. Prügikorjaja

Prügikorjaja (*garbage collector*) on teisisõnu automaatne mäluhaldus. Tema ülesanne on programmeerija käest mäluhaldus ära võtta. C keele puhul tuntud malloc() ja free() kasutamist võib mõelda kui manuaalset mäluhaldust.

Süsteemikeelte puhul on prügikorjaja implementeerimine ebatavaline (näiteks C ja C++ ei kasutata). Java puhul asub prügikorjaja JVM peal ja see on läbi aastakümnete optimeeritud. Go 1.9 versioon (aasta 2017) on tunduvalt oma prügikorjajat optimeerinud ja potensiaalne kiirus ületab Java oma, aga praeguse seisuga on Go mingite ülesannete täitmisel aeglasem kui Java – põhjus seisneb selles, et GoGCC (*Go GNU Compiler Collection*) ei ole veel implementeerinud kõiki võimalikke optimisatsioone.

Algselt oli kavas tuua näiteid kiiruse osas, kuid kiiresti tuli ilmsiks, et kõige suuremad faktorid on implementatsioon ja riistvara (Why is Go so slow (compared to Java)?). Näiteks C keeles on võimalik rakendus implementeerida täielikult ainult *Assembly* funktsionaalsust kasutades, aga mitte ükski mõistlik programmeerija ei tee seda praktikas, kuna saadud võit ei õigusta ajakulu.

Java puhul on paljud rakendused ülesse ehitatud raamistikel (Spring, GWT, Struts, Vaadin, JSF, Grail jne.). Go rakendused väldivad raamistikke kasutust ja sellest tulenev erinevus praktikas avaldab tunduvalt suuremat mõju kompileerimisele ja käivitamisele. Samuti oleneb millised vahendid on saadaval standardteegis ja kui suur on minimaalne aeg nende importimiseks. Sellegipoolest on tehtud sellel teemal juba uurimus (Go programs versus Java).

1.6. Keele põhikomponendid

Go on ülesehitatud tüüpidel ja funktsioonidel. Liides (*interface*) on tüüp milles on meetodite signatuurid välja kirjeldatud. Java puhul on klass *interface* selleks ja *abstract* mis hõlmab endas peale signatuuride ka implementeeritud funktsioone, lisaks veel on liidese implementeerimine täiesti erinev.

Java puhul peab klass implementeerima liidest, Go puhul peab implementeerima ainult liidese meetodeid – kusjuures seda ei pea tegema tüübi defineerimise ajal, piisab kui võrdluse/määramise momendil on tüübil kõik vajalikud meetodid küljes. NB! paketisiseselt saab ainult juurde lisada kõikjal, võõraste pakettide kasutamisel ei saa nende tüüpe täiendada enda meetoditega.

Struct on tüüp mille sees on kasutatavad muutujad ja nende tüübid välja kirjeldatud – Java puhul võib seda mõelda kui mudeli klassi.

Meetod on eritüüpi funktsioon millel on olemas *receiver* parameeter mis viitab *struct* tüübile. Java puhul tehakse nii objekti mudeli muutujate ja mudeli meetodite defineerimine tavaliselt OOP(*object oriented programming*) puhul ühes klassis, Go puhul defineeritakse *struct* ning peale defineerimist luuakse meetodid mis on sellega seotud. See lisab kirjutamist, kuna iga meetodi kirjutamisel tuleb ka näidata, mis *struct* kohta see käib. Hea töövoog (näiteks kasutades hästi konfigureeritud VIM/Atom/Sublime/Notepad/Brackets tekstiredaktoreid) elimimineerib selle probleemi.

1.7. Kompositsioon üle pärandamise

Java laialtlevinud klasside loomise muster on pärandamine (*inheritance*) ja Go kasutab kompositsiooni (*composition*) mustrit. Tüüp ja klass on suuresti väga sarnased mustrite seisukohalt. Pärinevus eeldab, et laienev klass on alamklass ja täidab „*is-a*“ semantilist argumenti. Näiteks klass „Kass“ oleks „Loom“ superklassi alamklass – tegu on pärinevus-suhtel.

Kompositsioon aga ei nõua seda. Go puhul saab tüübi A sisse manustada teist tüüpi B. Ehk mõtlemine peab toimuma vastassuunaliselt. Go puhul looksime alguses liidesed kirjeldamaks mingit tegevust (*behaviour*).

Loome liidesed mis kirjeldavad looma liidest. Näiteks loom hingab, liigub ja sööb – oleme loonud kolm liidest kirjeldamaks looma liidest. Kassi tüübi loomisel implementeeriksime kõik kolm vastavat liidest ja lisaksime veel pealekauba Kassi kirjeldava *käppadeArv* muutuja. Loome lisaks tüübi „Traktor“. Traktor ainult liigub ja sööb kütust, seega ta implementeerib ainult kaks looma meetodit. Kui kass hingab, liigub ja sööb siis järelikult on tegu loomaga ja seega on võimalik kõikjal kus kasutatakse liidest Loom kasutada hoopis Kassi. Näiteks kui meil on funktsioon *lisaLoomadeRegistrisse* mis võtab sisendis vastu liidese Loom, siis saame kutsuda seda andes sisendiks hoopis Kassi. Kuna Traktor aga ei hinga, siis ei saa seda lisada loomade registrisse.

Kassi kompositsioon kasutab kolme liidest ja tal on Traktoriga kaks ühist liidest. Kui me oleks aga pärandanud Java keeles Loom klassi, siis Traktori puhul rikuksime semantilist kuuluvust.

Pärinevus-suhte rikkumise probleem esineb isegi Java standardteekides. Näiteks klass *Stack* pikendab klassi *Vector*. Semantiliselt pinu ja vektor on kaks täiesti erinevat asja. Peale selle veel, kui keegi peaks muutma *Vector* klassi, on olemas tõenäosus, et tekivad kõrvalnähud klassi *Stack* implementeerimises. Kompositsiooni puhul jagaksid nad endiselt *add()* ja *remove()* meetodeid ja muudatused klassis *Vector* ei tekitaks mingil moel kõrvalnähtusi klassis *Stack*. See ei tähenda, et kompositsioon on parem kui pärinevus. Pärinevusmustril on eelis kui tekib suur loogiline pärinevuse puu. Kompositsiooni eelis on tihti see, et muudatusi on koodi tunduvalt kergem sisse viia.

1.8. Goroutine

Goroutine on väga saranane Java lõimudele. Põhierinevus on mälu kulu *goroutine* loomiseks. (Sundarram). Vaikimisi Javas võtab 1024KB 64-bitisel virtuaalmasinal, et üks lõim luua. *Jetty* on Javas kirjutatud veebiserver (ja servlet), mis on vägagi populaarne. *Jetty* on tuntud

ka selle poolest, et teatud tingimustes hõivab suures koguses mälu. *Goroutine* puhul hõivatakse vaid 2KB ja suurendatakse vajalikku pinu (*stack*) automaatselt vastavalt vajadusele. See annab meeletu eelise just veebiserverite puhul, kus igale kliendile määratakse eraldi *goroutine*/lõim. Peale selle ei ole vaja muretseda ka füüsilise protsessori lõimu blokeerimise peale.

Kuna lõimud on kallid ülesse seada ja maha võtta, siis tihti suurtemates süsteemides Java puhul implementeeritakse strateegiat nimega „thread pooling“. Seda võib ette kujutada kui pinu lõime. Kui protsess lõpetab lõimul töötamise, siis selle asemel, et see kustutada ja mälu vabastada, tagastatakse see tahtlikult pinusse järgmise protsessi jaoks. Selliste probleemide peale ei pea mõtlema Go kirjutamisel.

Lisaks on veel *goroutine* juurde ehitatud primitiivne tüüp *channel*. Selle abil saab kahe *goroutine* vahel vahetada infot. Java puhul info vahetamine kahe lõimu vahel vajab põhjalikke teadmisi lõimu haldamises. Miinus Go puhul on see, et *goroutine* haldus ei ole tasuta, see ohverdab protsessimisvõimekusest, kuna dünaamiliselt pinu suurendamine ja *scheduling* on usaldatu täielikult kompilaatori ja *runtime* kätte. Sellegipoolest see teeb reaalajas või suure koormustaluvusel töötavate rakenduste ehitamise väga kulu-efektiivseks.

2. Kes ja miks kasutavad?

Enamus antud nimekirjas koonduvad põhjustele mis on seotud jõudluse ja kiirusega. Go kogukonna poolt on loodud mahukas nimekiri autoritest, kes on kasutanud Go keelt projektide loomisel (Companies currently using Go throughout the world). Eelmainitud resurss on väga kasulik, aga see ei ole täielik ning nimekiri ei ole mingil pole agregeeritud.

2.1. Caddy

Tegu on staatilise veebiserveri rakendusega. *Caddy* on käivitatav läbi ühe binaarfaili (*single binary deploy*) ilma mingi konfigureerimiseta. Vaikimisi käivitub HTTP/2 protokolli automaatse HTTPS-ga ja tänu sellele kui odav on *goroutine* mälu kontekstis, on väga hea jõudlusega. Kasulik kui on vaja kiiresti ülesseada staatiline veebiserver või kiiresti implementeerida HTTPS. *Caddy* pole veel üldtuntud, aga arvav see võib muutuda programmeerimise valdkonnas laialdikasutatavaks vahendiks, seega lisasin esimeseks.

2.2. Google Kubernetes

Kubernetes võimaldab konteinerite (virtuaalmasinad) näol hallata rakendusi. Konteinereid saab skaleerida vastavalt vajadusele ja analüüsida nende jõudlust. Teised sarnased konkurendid on *Docker* ja *Heroku*. Sarnast turunišši täitvaid firmasid on veel, kes kasutavad Go keelt. Näiteks *Wercker* ja *Codeship*.

Kubernetese tiim kiitis Go KISS(*Keep It Stupid Simple*) kultuuri, häid kiireid tööriistu, teekidekogusid, prügikorjajat ja lõimuprotsesside kokkulangevust (*concurrency*) (Kubernetes + Go = Crazy Delicious).

2.3. Google allalaadimisserver

Google on monoliitne ettevõtte. Google Earth, Android SDK, Chrome, YouTube ja paljud teised kõik kasutavad ühist allalaadimise serverit. Server on ehitatud ülesse Go najal mitmel põhjusel. Esiteks, eelnev kood oli kirjutatud C++ peal. Koodibaas oli käest ära läinud ja liigselt üleehitatuks. Otsustati kasutada midagi, mis on süsteemitaseme keel, aga omab vahendeid kergesti loetava koodi jaoks. Kuna Go kompilaatori katte all peitub GCC, siis oli võimalik C++ pealt inkrementaalselt migreeruda Go peale (Fitzpatrick).

2.4. CloudFlare

Pakuvad teenuseid mis tõstavad rakenduste turvalisust ja kaitset liigse koormuse eest. Pika staažiga ettevõtte kelle käsutuses on globaalselt suur kogus andmekeskuseid. Peamised kasutamise põhjused on jälle *goroutine* ja *channel* loogika ja olemasolu. Peale seda kiidetakse ka standardteeki, mainides lisaks, et HTTP, TLS, räsi loomine, *string* manipulatsioonid, logimine ja ajaga seonduvad funktsionaalsused on väga teretulnud standardteegis (Go at CloudFlare).

2.5. DigitalOcean

Kiidetakse Go liideste lahendust, tööriistu ja välise koodiga integreerimist. Kritiseeritakse Go paketihaldust. Nimelt installeerides Go, tuleb kaasa käsk "*go get* [paketi asukoht, tavaliselt github]". Käsk tõmbab alati alla viimase paketi seisu. See on väga halb kui tahetakse luu jätkusuutlikut koodi. Kuigi Go paketid on tavaliselt väga kõrge kvaliteediga, ei saa eeldada, et kõik paketid peavad lugu tagasiulatuvuse suhtes. Sellele probleemile on Go kogukonna poolt loodud lahendus *godep* rakenduse näol (Get Your Development Team Started With Go).

3. Dokumentatsioon rakenduse loomisest

Rakendus on loodud iteratsioonidega peegeldamaks tegelikku töövoogu ja mõtteviisi mida kasutati rakenduse loomisel. Lisade vaatamisel pöörata tähelepanu iteratsiooni numbrile, kuna igal iteratsioonil mängib ümberkirjutamine rolli. Lõpptulemuse kriteeriumiteks on järgnevad punktid:

- Peab kasutama kolmanda osapoole programmiliidest
- Peab salvestama/lugema andmeid
- Peab sisaldama erinevaid tüüpi objekte
- Peab nähtavaks tegema erinevate tüüpide REST lõpp-punktid
- Peab kasutama vähemalt ühte *interface* tüüpi

3.1. Seadistus

Installeerimiseks tuleb järgida juhendeid https://golang.org/doc/install aadressilt. Igal juhendis mainitud platvormil (FreeBSD, MacOS, Windows ja Linux) toimub installeerimine erinevalt. Installatsiooni lõpus kindlasti veenduda, et PATH keskkonna muutuja on seadistatud korrektselt ja kui on valitud tavalisest erinev töökaust, veenduda, et GOPATH keskkonna muutuja on seadistatud korrektselt. Vastav Bash *shell* käsk kontrollimaks:

```
echo $PATH $GOPATH
```

Töökeskkond Go puhul näeb välja järgnevalt:

`$GOPATH/go/src` – algkoodi kaust

`$GOPATH/go/build` – kompileeritud koodi kaust

Antud töö puhul on kasutatud neli käsku:

`go get github.com/nanobox-io/golang-scribble` – moodul mis sisaldab JSON andmebaasi

`go run [faili nimi]` – kompileerib ja käivitab ühe faili

`go build` – projekti kaustas käivitades kompileerib kausta `$GOPATH/go/build`

Nimekiri kõikidest võimalikest käskudest saadaval: https://golang.org/cmd/go/

Luua uus fail `hello.go` uue projekti kausta sees `$GOPATH/go/src/hello` ja täita järgnevaga:

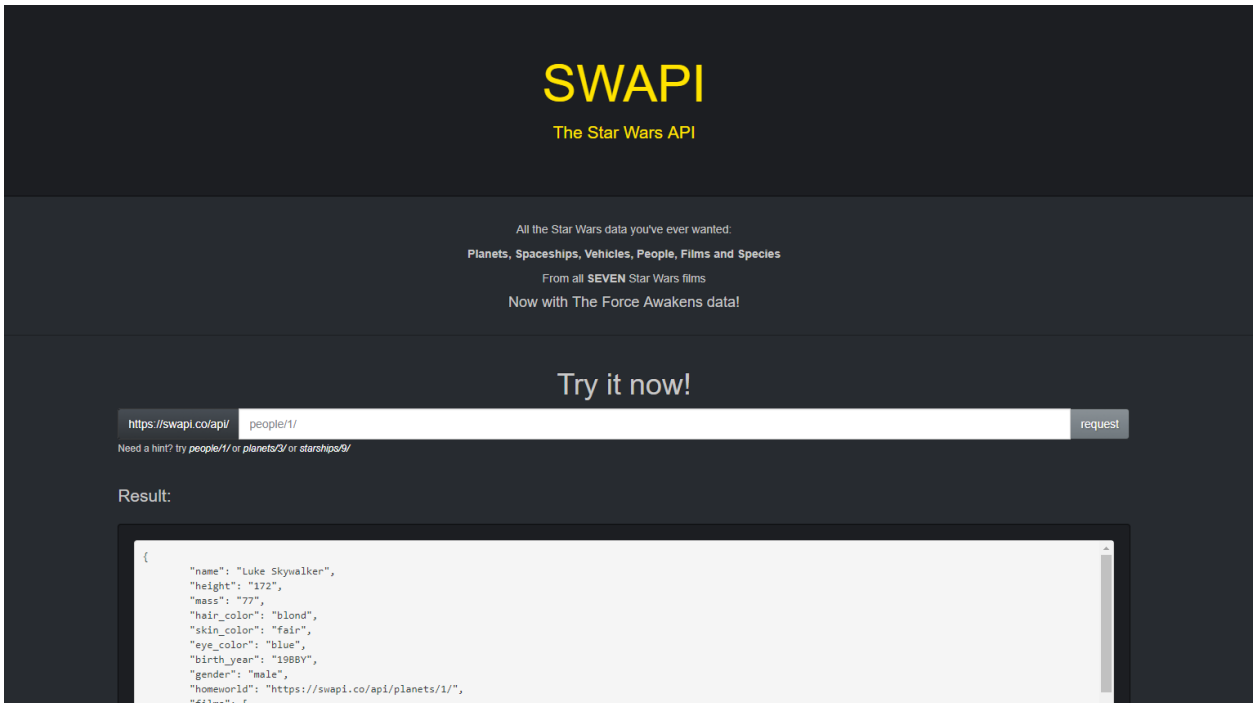
```
package main

import "fmt"

func main() {

    fmt.Printf("Hello world!\n")

}
```

Seejärel samas kaustas käivitada kompileermise käsk: `go build`. Nüüd binaarfaili käivitamisel algkoodi kaustas (Windows puhul `hello.exe`) prindib rakendus konsooli: *Hello World!*

3.2. 1. Iteratsioon

Iteratsiooni eesmärk oli leida töödeldav kolmanda osapoole programmiliides, implementeerida see ja proovida ka testimise võimalusi. Valitud programmiliideseks osutus: https://swapi.co/ (Star Wars API) (vt. Joonis 1). Peamiselt sellepärast, et programmiliides on hästi esitatud ja tundus kergesti töödeldavana. Mis eesmärgil seda kasutada veel ei olnud selge. Alustuseks kasutati liidese lõpp-puntki `/starships/`. Kuna tegu on JSON formaadis vastusega, siis Go keeles on kerge kaardistada vastus vastu kindlat tüüpi. Loodud tüüpideks said `Starships` ja `Starship`. Kõikide objektide saamiseks pidi jälitama `Next` parameetrit, vaikimisi tagastab

esimese lehe. Sellepärast on loodud ka kaks tüüpi ühe asmel. `Starships` tüübi küljes on meetod `fill()` mis rekursiivselt täidab enda küljes olevat `Starship` objektide massiivi andes sisendiks järgmise lehe URLi. Iteratsiooni tulemuseks oli kaks faili `main.go` ja `main_test.go (lisa 2).` Käivitamisel tagastab kõikide lehtede tähelaevade kirjelduse.



Joonis 1

3.3. 2. Iteratsioon

Iteratsiooni eesmärk oli salvestada/lugeda informatsiooni, teha *interface* ja parendada struktuuri. Algul loodi uus fail `database.go` mis hakkab haldama salvestamist, kustutamist ja lugemist. Selleks kasutati Scribble moodulit mis kujutab endas JSON andmebaasi failide kujul. Implementeeriti seejärel salvestamine tähelaevade (mille me täidame eelnevalt programmiliidese vastusega) jaoks `insertStarships()` meetodiga. Peale rakenduse käivitamist salvestatud tähelaevad kajastuvad kaustas starship (vt. Joonis 2). Siinkohal eraldati tähelaevade mudel ja selle meetodid eraldi faili `starship.go` – sisuliselt oleme loonud nüüd Java klassi analoogi. Nüüd alustati tööd *interface* `tüübiga.` Valitut tüübiks sai *`Fighter`* `main.go` failis. Seejärel loodi kaks tüüpi mis omavad kõiki *Fighter* meetodeid: *Rebel* ja *Chewbacca.* Lisaks veel tüüp *Palpatine* mis erineb teistest selle poolest, et ei oma `canFight()` meetodit. Siinkohal on mõeldud interaktsioon nende tüüpide vahel järgmine:

- *Palpatine* tüübil on parameeter *strength* mis peegeldab "elusi"
- *Fighter* on tüüp mis peegeldab võitlejaid, kes võitlevad *Palpatine* tüübiga
- Võitlejateks on kaht sorti tüüpe *Chewbacca* ja *Rebel*
- *Chewbacca* saab võidelda (`canFight())` siis kui ta `hunger` parameeter on piisavalt suur
- *Rebel* saab võidelda kui tal on piisavalt "raha" enda tähelaeva kütuse jaoks
- *Palpatine* tüübil on meetod `fightPalpatine()` mis võtab sisendiks *Fighter* tüübi. Meetodi eesmärgiks on vähendada *Palpatine* tüübi elusi ja tagastada tõeväärtuse mis peegeldab kas võitleja võitis või kaotas.

Iteratsiooni tulemuseks oli kolm uut faili `database.go`, `starship.go` ja `starship_test.go` ning `main.go` ümberkirjutatud (lisa 3).


richard@cassini:~/Projects/go/src/github.com/richardaasa/starship$ ls
'AA-9 Coruscant freighter.json'
arc-170.json
A-wing.json
'Banking clan frigate.json'
'Belbullab-22 starfighter.json'
B-wing.json
'Calamari Cruiser.json'
'CR90 corvette.json'
'Death Star.json'
'Droid control ship.json'
'EF76 Nebulon-B escort frigate.json'
Executor.json
'H-type Nubian yacht.json'
'Imperial shuttle.json'
'Jedi Interceptor.json'
'Jedi starfighter.json'
'J-type diplomatic barge.json'
'Millennium Falcon.json'
'Naboo fighter.json'
'Naboo Royal Starship.json'
'Naboo star skiff.json'
'Rebel transport.json'
'Republic Assault ship.json'
'Republic attack cruiser.json'
'Republic Cruiser.json'
Scimitar.json
'Sentinel-class landing craft.json'
'Slave 1.json'
'Solar Sailer.json'
'Star Destroyer.json'
'T-70 X-wing fighter.json'
'Theta-class T-2c shuttle.json'
'TIE Advanced x1.json'
'Trade Federation cruiser.json'
V-wing.json
X-wing.json
Y-wing.json
richard@cassini:~/Projects/go/src/github.com/richardaasa/starship$

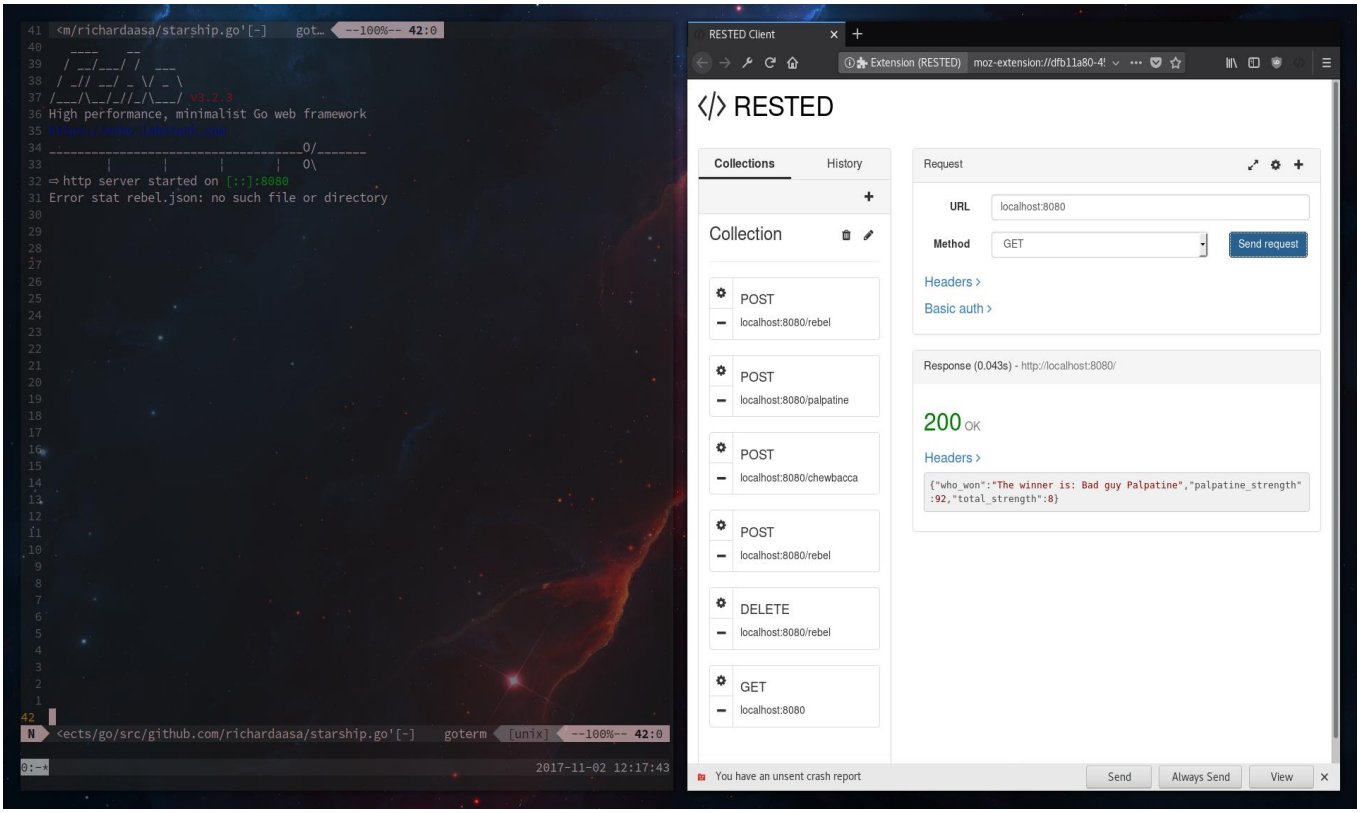
Joonis 2

3.4. 3. Iteratsioon

Iteratsiooni eesmärk oli implementeerida eelneva iteratsiooni meetodid ja luua salvestamise ja lugemise võimalus uutele loodud tüüpidele (*Palpatine*, *Rebel*, *Chewbacca*). Igale tüübile loodi analoogselt tähelaeva salvestamisele andmebaasi salvestamise funktsioonid. Lisaks loodi lugemiseks andmebaasi funktsioonid. Nüüd kus kõik "ehitusklotsid" on olemas, loome testimise ahela `main()` funktsioonis. Ahelas sisestame andmebaasi mõlemat sorti võitlejaid (*Chewbacca* ja *Rebel*). *Rebel* sisestamisel küsitakse andmebaasis suvalise tähelaeva ja määratakse see võitleja külge. Seejärel sisestame *Palpatine* tüübi ja määrame ta *strength* parameetri. Nüüd kus vajalik andmestik on andmebaasis olemas, pärime andmebaasist kõik võitlejad ja salvestatud *Palpatine* tüübi. Nüüd saame nende vahel lahingut pidada. Iteratiivselt võideldakse *Palpatine* tüübiga senikaua kuni *strength* parameeter on väiksem-võrdne nulliga (mis juhul kuvame võitjana viimase võitleja nime) või peale viimast iteratsiooni kui keegi pole võitnud kuvatakse *Palpatine* võitjana. Selle iteratsiooni käigus uusi faile ei loodud, aga kirjutati umber järgmised failid: `main.go, starship.go` ja `database.go (lisa 4)`.

3.5. 4. iteratsioon

Iteratsiooni eesmärk oli viimased vajalikud parandused ja veebiserveri seadistamine, et eelnev tehtud mäng saaks läbi mängida HTTP POST ja GET päringutega. Selleks implementeeriti Echo moodul, mis võimaldab seda kergesti ja loetavalt teha. Standardteek võimaldab seda samuti, aga standardteegis puuduvad abifunktsioonid mis teevad lõpp-punktide loomise väga kergeks ja loetavaks. Uue veebiserveri lõpp-punktide haldus toimub `router.go` failis. Lõpptulemuseks on loodud rakendus mis täidab kõiki alguses seatud kriteeriume (vt. Joonis 3). Alguses päritakse kolmandalt osapoolelt tähelaevade kohta informatsiooni, mis salvestatakse andmebaasi. Seejärel initsialiseeritakse veebiserver mis avab kõik vajalikud lõpp-punktid tüüpidele *Rebel*, *Chewbacca* ja *Palpatine*. Nüüd on võimalik 3. iteratsioonis tehtud mäng läbi mängida HTTP päringutega. Selle iteratsiooni käigus loodi üks uus fail `router.go` ja kirjutati ümber `main.go`, `starship.go`, `database.go` (lisa 4).



Joonis 3

Kokkuvõte

Antud töö eesmärk oli analüüsida Go programmeerimiskeelt. Uurida miks seda kasutatakse ja dokumenteerida ühe Go rakenduse loomist.

Go on keel mille omadustel leidub palju analooge teiste keeltega nagu Java ja C aga erinevused esinevad detailides. Märkimisväärsed erinevused väljenduvad järgmistes omadustes: *goroutine* (analoogne lõimuga), *interface* ja kompileerimine.

Mainekamad ettevõtted programmeerimise vallas kiitsid kõige enam koodi kerget loetavust, väga head koormustaluvust, kiirust, standardteekide sisu, C keeltelt kerget migratsiooni ja tööriistu.

Rakenduse loomine oli tehtud iteratiivse protsessiga. Iga abstrakse verstaposti tagant salvestati koodi algkood ja kirjeldati iteratsiooni eesmärki ning tulemust. Loodav programmiliides andis autorile sügavamaid teadmisi nii programmeerimise kui ka Go keele kohta. Tulemuseks oli Star Wars temaatikaga programmiliides kus saab läbi mängida lahinguid HTTP päringute vahendusel.

Kasutatud kirjandus

Go (programming language). (2009, November 11). Retrieved Oktoober 28, 2017, from Wikipedia: https://en.wikipedia.org/wiki/Go_(programming_language)

Companies currently using Go throughout the world. (n.d.). Retrieved oktoober 28, 2017, from GitHub: https://github.com/golang/go/wiki/GoUsers

Fitzpatrick, B. (n.d.). *dl.google.com: Powered by Go*. Retrieved oktoober 28, 2017, from Golang Talks: https://talks.golang.org/2013/oscon-dl.slide#1

Get Your Development Team Started With Go. (n.d.). Retrieved november 2, 2017, from DigitalOcean: https://blog.digitalocean.com/get-your-development-team-started-with-go/

Go at CloudFlare. (n.d.). Retrieved oktoober 29, 2017, from Cloudflare: https://blog.cloudflare.com/go-at-cloudflare/

Go programs versus Java. (n.d.). Retrieved oktoober 28, 2017, from The Computer Language Benchmarks Game: http://benchmarksgame.alioth.debian.org/u64q/go.html

JDK 1.8/1.7 Compatibility Gotcha. (n.d.). Retrieved november 2, 2017, from Draconian Overlord: http://www.draconianoverlord.com/2014/04/01/jdk-compatibility.html

Krennmair, A. (n.d.). *Go & cgo: integrating existing C code with Go*. Retrieved oktoober 28, 2017, from Go & cgo: integrating existing C code with Go: http://akrennmair.github.io/golang-cgo-slides/#1

Kubernetes + Go = Crazy Delicious. (n.d.). Retrieved oktoober 29, 2017, from Gopher Academy Blog: https://blog.gopheracademy.com/birthday-bash-2014/kubernetes-go-crazy-delicious/

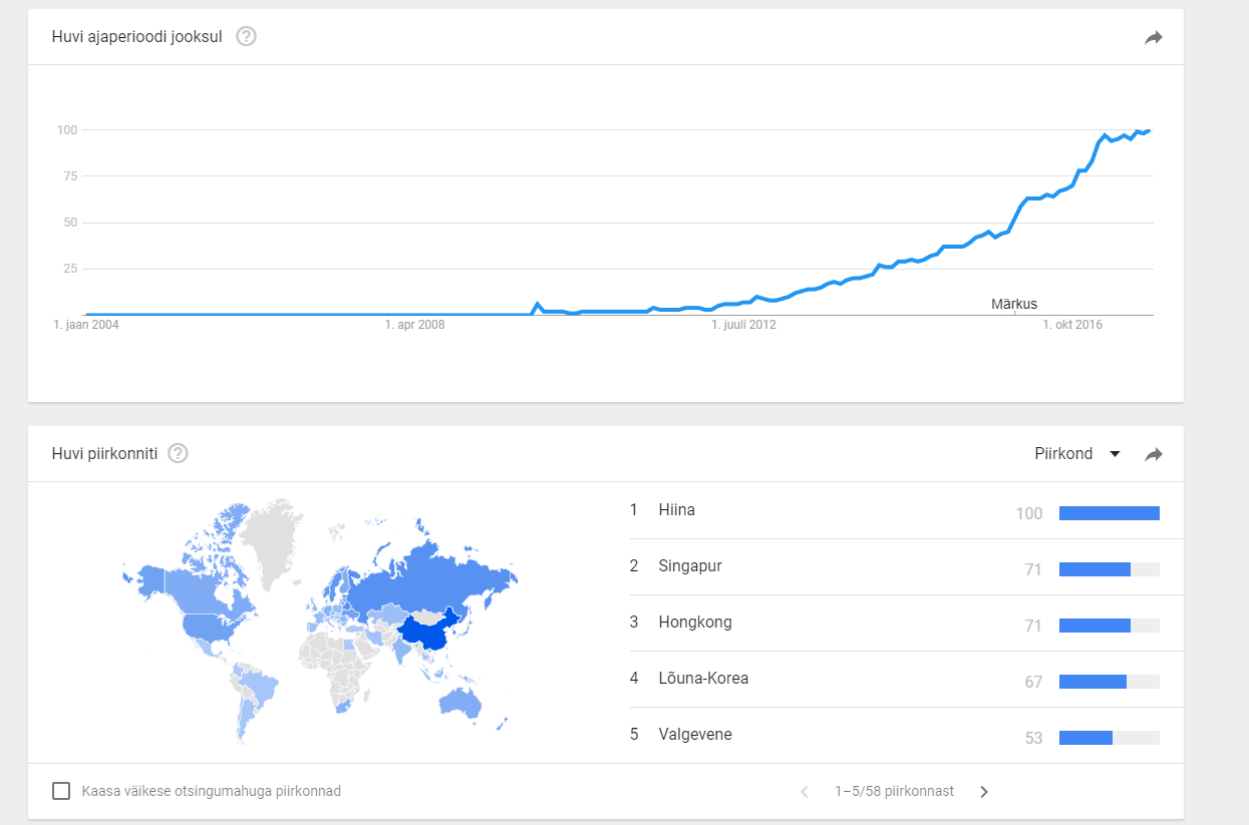
Sundarram, K. (n.d.). *How Goroutines Work*. Retrieved oktoober 28, 2017, from Krishna's blog: http://blog.nindalf.com/how-goroutines-work/

Taylor, I. L. (2012, juuli 11). *Gccgo in GCC 4.7.1*. Retrieved from https://blog.golang.org/gccgo-in-gcc-471

Why is Go so slow (compared to Java)? (n.d.). Retrieved oktoober 28, 2017, from StackOverflow: https://stackoverflow.com/questions/2704417/why-is-go-so-slow-compared-to-java

Lisad

Lisa 1 Google Trends



Lisa 2 1. Iteratsiooni algkood

main.go

```go
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "time"
)

// Esmane API URL kust infot küsida
const baseURL string = "https://swapi.co/api/starships/?page=1"

// Loome uue HTTP klienti
// NB! & ja * operaatorid käituvad täpselt samasuguselt nagu C keele viited
// Kui räägitakse, et Go keeles puudub reference tüüpi viide, siis seda peetakse silmas.
// Erinevus seisneb selles, et ei pea ise manuaalselt mälu allokeerima.
var httpClient = &http.Client{Timeout: 10 * time.Second}

// Objekt mille vastu me hakkame API vastust kaardistama.
// Tahame kaardistada ainult kahte välja.
// Kuna API serveerib kosmoselaevu üle mitme lehe, on meil vaja antud
// mudelit. Next viitab järgmisele lehele.
type Starships struct {
    Next    string     `json:"next"`
    Results []Starship `json:"results"`
}
// API vastus on mitme tasemeline. See mudel hoiab infot mida meil tegelikult vaja.
type Starship struct {
    Name          string `json:"name"`
    Model         string `json:"model"`
    CostInCredits string `json:"cost_in_credits"`
    Length        string `json:"length"`
}

// JSON paketi Decode sisendit
// ootab tüüpi mille vastu kaardistada, antud puhul Starships tüüp.
// interface{} on alternaative lahendus paljudes kohtades Java
// generics võimalustele. Erinevalt Java generic tüübist, ei vaja
// interface{} implementatsiooni. Isiklikult leian see teeb koodi
// kergemini loetavaks ja paremini arusaadavaks.

// func name(params...) [return type]
func getJSON(url string, target interface{}) error {
```

```
    // Go keeles puudub exception. Selle asemel funktsioonid
    // on võimelised tagastama kaks väärtust, millest teine on alati
    // viga. Vea sisu kontrollitakse kohe selle määramise all.
    // Seda lähenemist on laialt kritiseeritud, aga minu jaoks teeb
    // see nii testimise kui navigeerimise kergemaks.
    r, err := httpClient.Get(url)
    if err != nil {
        return err
    }

    // Body on tüüpi io.ReadCloser
    // Javas me tavaliselt sulgeme IO tegevusi finally blokis
    // defer teeb sisuliselt sama, paneb IO sulgemise pinu lõppu
    defer r.Body.Close()
    return json.NewDecoder(r.Body).Decode(target)
}

// Lisame rekursiivselt kosmoselaevu Starships.Result massiivi
// Seekord tegu meetodiga, kuna esineb receiver

// func [receiver] name(params...) [return type]
func (ships *Starships) fill(pageUrl string) error {
    // Loome vaheobjekti salvestamaks uue API lehe sisu
    tempShips := Starships{}

    // Pärime API käest alguses leht 1 (baseURL sees määratud)
    // seejärel rekursiivelt küsime järgmist lehte Next abil.
    // &tempShips tähendab me täiendame algset objekti.
    // Ilma & operaatorita, annaksime me funktsioonile getJSON
    // edasi koopia mudelist.
    err := getJSON(pageUrl, &tempShips)
    if err != nil {
        return err
    }

    // Lisame kõik vastuse kosmoselaevad algsesse massiivi
    // [] on tegelikult slice. Erinevus seisneb tavalise C
    // massiivi vahel selles, et tal on lisafunktsionaalsusi
    // mis võimaldab kergemini massiive hallata. Näiteks append()
    for i := 0; i < len(tempShips.Results); i++ {
        ships.Results = append(ships.Results, tempShips.Results[i])
    }

    // Küsime järgmist lehte kui on olemas ja lõpetame
    // töö kui enam ei ole.
    ships.Next = tempShips.Next

    if ships.Next != "" {
        err = ships.fill(ships.Next)
    }

    return err
```

```
}

// Nagu Java keeleski, main() on alati esimene funktsioon mida vaikimisi
käivitatakse
func main() {
    starShips := Starships{}

    err := starShips.fill(baseURL)
    if err != nil {
        panic(err)
    }

    resp, _ := json.Marshal(starShips)
    fmt.Printf("starShips = %+v\n", string(resp))
}
```

main_test.go

```
package main

import (
    "encoding/json"
    "testing"
)

func TestGetJSON(t *testing.T) {
    ships := Starships{}

    err := getJSON(baseURL, &ships)
    if err != nil {
        t.Errorf("Failed to retreive data from %s err: %s", baseURL, err)
    }

    result, err := json.Marshal(ships)
    if err != nil {
        t.Errorf("Failed to marshal JSON")
    }

    if string(result) == "" {
        t.Errorf("Expecting anything in result, got %s", result)
    }
}

func TestStarshipsFill(t *testing.T) {
    ships := Starships{}

    err := ships.fill(baseURL)
    if err != nil {
       t.Errorf("Failed to fill() Starships err: %s", err)
    }
```

```
    result, err := json.Marshal(ships)
    if err != nil {
        t.Errorf("Failed to marshal JSON")
    }

    if string(result) == "" {
        t.Errorf("Expecting anything in result, got %s", result)
    }
}
```

Lisa 3 2. Iteratsioon algkood

main.go

```
package main

import (
    "strconv"
)

type Fighter interface {
    CanFight() bool
    Name() string
    GetStrength() int
}

type Chewbacca struct {
    Hunger   int `json:"hunger"`
    Strength int `json:"strength"`
}

func (c *Chewbacca) CanFight() bool {
    return c.Hunger > 20
}

func (c *Chewbacca) Name() string {
    return "Chewie"
}

func (c *Chewbacca) GetStrength() int {
    if c.CanFight() {
       return c.Strength
    }
    return 0
}

type Rebel struct {
    ID       int      `json:"-"`
    Credits  int      `json:"credits"`
    Starship Starship `json:"starship"`
    Strength int      `json:"strength"`
}

func (r *Rebel) CanFight() bool {
    cost, err := r.Starship.fuelCost()
    if err != nil {
        panic(err)
    }

    return r.Credits > cost
```

```
}

func (r *Rebel) Name() string {
    return "Rebel nr. " + strconv.Itoa(r.ID) + ", owner of " +
r.Starship.Name
}

func (r *Rebel) GetStrength() int {

    if r.CanFight() {
        return r.Strength
    }

    return 0
}

type Palpatine struct {
    Strength int `json:"strength"`
}

func (p *Palpatine) Name() string {
    return "Bad guy Palpatine"
}

func (p *Palpatine) FightPalpatine(f Fighter) bool {
    p.Strength -= f.GetStrength()
    return p.Strength <= 0
}

// Nagu Java keeleski, main() on alati esimene funktsioon mida vaikimisi
käivitatakse
func main() {
    starShips := Starships{}

    err := starShips.fill(baseURL)
    if err != nil {
        panic(err)
    }

    // Salvestame andmebaasi kosmoselaevad
    insertStarships(starShips.Results)

}
```

starship.go

```go
package main

import (
    "encoding/json"
    "net/http"
    "strconv"
    "time"
)

// Esmane API URL kust infot küsida
const baseURL string = "https://swapi.co/api/starships/?page=1"

// Loome uue HTTP klienti
// NB! & ja * operaatorid käituvad täpselt samasuguselt nagu C keele
viited
// Kui räägitakse, et Go keeles puudub reference tüüpi viide, siis seda
peetakse silmas.
// Erinevus seisneb selles, et ei pea ise manuaalselt mälu allokeerima.
var httpClient = &http.Client{Timeout: 10 * time.Second}

// Objekt mille vastu me hakkame API vastust kaardistama.
// Tahame kaardistada ainult kahte välja.
// Kuna API serveerib kosmoselaevu üle mitme lehe, on meil vaja antud
// mudelit. Next viitab järgmisele lehele.
type Starships struct {
    Next    string     `json:"next"`
    Results []Starship `json:"results"`
}

// API vastus on mitme tasemeline. See mudel hoiab infot mida meil
tegelikult vaja.
type Starship struct {
    Name          string `json:"name"`
    CostInCredits string `json:"cost_in_credits"`
    Length        string `json:"length"`
}

// Täiendame mudelit uue meetodiga.
// Hinna ja laevapikkuse jagatis ei peegelda muidugi
// reaaleluliselt üldse kütusehinda, aga API oli kahjuks
// selle koha pealt puudulik.
func (s *Starship) fuelCost() (int, error) {
    num1, err := strconv.Atoi(s.CostInCredits)
    if err != nil {
        return 0, err
    }

    num2, err := strconv.Atoi(s.Length)
    if err != nil {
        return 0, err
```

```
    }

    return num1 / num2, nil
}

// JSON paketi Decode sisendit
// ootab tüüpi mille vastu kaardistada, antud puhul Starships tüüp.
// interface{} on alternaative lahendus paljudes kohtades Java
// generics võimalustele. Erinevalt Java generic tüübist, ei vaja
// interface{} implementatsiooni. Isiklikult leian see teeb koodi
// kergemini loetavaks ja paremini arusaadavaks.

// func name(params...) [return type]
func getJSON(url string, target interface{}) error {
    // Go keeles puudub exception. Selle asemel funktsioonid
    // on võimelised tagastama kaks väärtust, millest teine on alati
    // viga. Vea sisu kontrollitakse kohe selle määramise all.
    // Seda lähenemist on laialt kritiseeritud, aga minu jaoks teeb
    // see nii testimise kui navigeerimise kergemaks.
    r, err := httpClient.Get(url)
    if err != nil {
        return err
    }

    // Body on tüüpi io.ReadCloser
    // Javas me tavaliselt sulgeme IO tegevusi finally blokis
    // defer teeb sisuliselt sama, paneb IO sulgemise pinu lõppu
    defer r.Body.Close()
    return json.NewDecoder(r.Body).Decode(target)
}

// Lisame rekursiivselt kosmoselaevu Starships.Result massiivi
// Seekord tegu meetodiga, kuna esineb receiver

// func [receiver] name(params...) [return type]
func (ships *Starships) fill(pageUrl string) error {
    // Loome vaheobjekti salvestamaks uue API lehe sisu
    tempShips := Starships{}

    // Pärime API käest alguses leht 1 (baseURL sees määratud)
    // seejärel rekursiivelt küsime järgmist lehte Next abil.
    // &tempShips tähendab me täiendame algset objekti.
    // Ilma & operaatorita, annaksime me funktsioonile getJSON
    // edasi koopia mudelist.
    err := getJSON(pageUrl, &tempShips)
    if err != nil {
        return err
    }

    // Lisame kõik vastuse kosmoselaevad algsesse massiivi
    // [] on tegelikult slice. Erinevus seisneb tavalise C
    // massiivi vahel selles, et tal on lisafunktsionaalsusi
```

```
    // mis võimaldab kergemini massiive hallata. Näiteks append()
    for i := 0; i < len(tempShips.Results); i++ {
        ships.Results = append(ships.Results, tempShips.Results[i])
    }

    // Küsime järgmist lehte kui on olemas ja lõpetame
    // töö kui enam ei ole.
    ships.Next = tempShips.Next

    if ships.Next != "" {
        err = ships.fill(ships.Next)
    }

    return err
}
```

starship_test.go

```
package main

import (
    "encoding/json"
    "testing"
)

// Siin täiendame jooksvalt funtktsioonide teste.
// VIM tekstiredaktor pakub väga häid tööriistu testide
// haldamiseks: 'faith/vim-go'

func TestFuelCost(t *testing.T) {
    ship := Starship{Length: "300", CostInCredits: "8500000"}
    cost, err := ship.fuelCost()

    if err != nil {
       t.Errorf("Failed to calculate fuelCost(): %d err: %s", cost, err)
   }

    if cost < 28300 {
        t.Errorf("Too small of a number: %d", cost)
    }

}

func TestGetJSON(t *testing.T) {
    ships := Starships{}

    err := getJSON(baseURL, &ships)
    if err != nil {
        t.Errorf("Failed to retreive data from %s err: %s", baseURL, err)
    }

    result, err := json.Marshal(ships)
    if err != nil {
        t.Errorf("Failed to marshal JSON")
    }

    if string(result) == "" {
        t.Errorf("Expecting anything in result, got %s", result)
    }
}

func TestStarshipsFill(t *testing.T) {
    ships := Starships{}

    err := ships.fill(baseURL)
    if err != nil {
        t.Errorf("Failed to fill() Starships err: %s", err)
    }
```

```
    result, err := json.Marshal(ships)
    if err != nil {
        t.Errorf("Failed to marshal JSON")
    }

    if string(result) == "" {
        t.Errorf("Expecting anything in result, got %s", result)
    }
}
```

database.go

```
package main

import (
    "fmt"

    scribble "github.com/nanobox-io/golang-scribble"
)

// Kirjuta kosmoselaevad andmebaasi
func insertStarships(ships []Starship) error {

    // Loome uue andmebaasi.
    // Scribble on väga mugav JSON andmebaas prototüüpimiseks.
    // Tabelid kujutavad endas kaustu ja faili nimi on ID.
    // Rakenduse käivitamisel luuakse algkoodi kausta "starship" kaust
    // mille sissu kirjutatakse kõik päritud kosmoselaevad.
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    for i := 0; i < len(ships); i++ {
        // Alternatiivne käekiri veahalduseks
        if err := database.Write("starship", ships[i].Name, ships[i]); err
!= nil {
            fmt.Println("Database error", err)
            return err
        }
    }
    return err
}
```

Lisa 4 3. Iteratsiooni algkood

main.go

```
package main

import (
    "fmt"
    "strconv"
)

const chewieThreshold = 20

type Fighter interface {
    CanFight() bool
    Name() string
    GetStrength() int
}

type Chewbacca struct {
    Hunger   int `json:"hunger"`
    Strength int `json:"strength"`
}

func (c *Chewbacca) CanFight() bool {
    return c.Hunger > chewieThreshold
}

func (c *Chewbacca) Name() string {
    return "Chewie"
}

func (c *Chewbacca) GetStrength() int {
    if c.CanFight() {
        return c.Strength
    }
    return 0
}

type Rebel struct {
    // Kui ID on tühi näiteks POST päringul, siis jäetakse
    // väli täitmata ja määratakse autmaatselt süsteemi poolt.
    ID       int      `json:"id, omitempty"`
    Credits  int      `json:"credits"`
    Starship Starship `json:"starship"`
    Strength int      `json:"strength"`
}

func (r *Rebel) CanFight() bool {
    cost, err := r.Starship.fuelCost()
```

```
    if err != nil {
        fmt.Println("Cost: %f", cost)
        panic(err)
    }

    return float32(r.Credits) > cost
}

func (r *Rebel) Name() string {
    return "Rebel nr. " + strconv.Itoa(r.ID) + ", owner of " +
r.Starship.Name
}

func (r *Rebel) GetStrength() int {

    if r.CanFight() {
        return r.Strength
    }

    return 0
}

type Palpatine struct {
    Strength int `json:"strength"`
}

func (p *Palpatine) Name() string {
    return "Bad guy Palpatine"
}

func (p *Palpatine) FightPalpatine(f Fighter) bool {
    p.Strength -= f.GetStrength()
    return p.Strength <= 0
}

// Nagu Java keeleski, main() on alati esimene funktsioon mida vaikimisi
käivitatakse
func main() {
    starShips := Starships{}

    err := starShips.fill(baseURL)
    if err != nil {
        // panic prindib detailse stacktracei
        panic(err)
    }

    // Salvestame andmebaasi kosmoselaevad
    insertStarships(starShips.Results)

    // Uute objektide loomine ja salvestamine andmebaasi
    insertRebel(Rebel{Credits: 300000000, Strength: 1})
    insertRebel(Rebel{Credits: 300000000, Strength: 1})
```

```
    insertRebel(Rebel{Credits: 300000000, Strength: 1})
    insertRebel(Rebel{Credits: 300000000, Strength: 1})
    insertChewbacca(Chewbacca{Hunger: 30, Strength: 100})
    insertPalpatine(Palpatine{Strength: 105})

    rebels := getRebels()
    chewie := getChewbacca()
    palpatine := getPalpatine()

    winner := palpatine.Name()

    if palpatine.FightPalpatine(&chewie) {
        winner = chewie.Name()
    }

    for _, rebel := range rebels {
        if palpatine.FightPalpatine(&rebel) {
            winner = rebel.Name()
        }
    }

    fmt.Println("The winner is: " + winner)
    fmt.Printf("Palpatines remaining strength: %d", palpatine.Strength)
}
```

starship.go

```go
package main

import (
    "encoding/json"
    "net/http"
    "strconv"
    "time"
)

// Esmane API URL kust infot küsida
const baseURL string = "https://swapi.co/api/starships/?page=1"

// Loome uue HTTP klienti
// NB! & ja * operaatorid käituvad täpselt samasuguselt nagu C keele
viited
// Kui räägitakse, et Go keeles puudub reference tüüpi viide, siis seda
peetakse silmas.
// Erinevus seisneb selles, et ei pea ise manuaalselt mälu allokeerima.
var httpClient = &http.Client{Timeout: 10 * time.Second}

// Objekt mille vastu me hakkame API vastust kaardistama.
// Tahame kaardistada ainult kahte välja.
// Kuna API serveerib kosmoselaevu üle mitme lehe, on meil vaja antud
// mudelit. Next viitab järgmisele lehele.
type Starships struct {
    Next    string     `json:"next"`
    Results []Starship `json:"results"`
}

// API vastus on mitme tasemeline. See mudel hoiab infot mida meil
tegelikult vaja.
type Starship struct {
    Name          string `json:"name"`
    CostInCredits string `json:"cost_in_credits"`
    Length        string `json:"length"`
}

// Täiendame mudelit uue meetodiga.
// Hinna ja laevapikkuse jagatis ei peegelda muidugi
// reaaleluliselt üldse kütusehinda, aga API oli kahjuks
// selle koha pealt puudulik.
func (s *Starship) fuelCost() (float32, error) {
    num1, err := strconv.ParseFloat(s.CostInCredits, 32)
    if err != nil {
        return 0, err
    }

    num2, err := strconv.ParseFloat(s.Length, 32)
    if err != nil {
        return 0, err
```

```
    }

    return float32(num1 / num2), nil
}

// JSON paketi Decode sisendit
// ootab tüüpi mille vastu kaardistada, antud puhul Starships tüüp.
// interface{} on alternaative lahendus paljudes kohtades Java
// generics võimalustele. Erinevalt Java generic tüübist, ei vaja
// interface{} implementatsiooni. Isiklikult leian see teeb koodi
// kergemini loetavaks ja paremini arusaadavaks.

// func name(params...) [return type]
func getJSON(url string, target interface{}) error {
    // Go keeles puudub exception. Selle asemel funktsioonid
    // on võimelised tagastama kaks väärtust, millest teine on alati
    // viga. Vea sisu kontrollitakse kohe selle määramise all.
    // Seda lähenemist on laialt kritiseeritud, aga minu jaoks teeb
    // see nii testimise kui navigeerimise kergemaks.
    r, err := httpClient.Get(url)
    if err != nil {
        return err
    }

    // Body on tüüpi io.ReadCloser
    // Javas me tavaliselt sulgeme IO tegevusi finally blokis
    // defer teeb sisuliselt sama, paneb IO sulgemise pinu lõppu
    defer r.Body.Close()
    return json.NewDecoder(r.Body).Decode(target)
}

// Lisame rekursiivselt kosmoselaevu Starships.Result massiivi
// Seekord tegu meetodiga, kuna esineb receiver

// func [receiver] name(params...) [return type]
func (ships *Starships) fill(pageUrl string) error {
    // Loome vaheobjekti salvestamaks uue API lehe sisu
    tempShips := Starships{}

    // Pärime API käest alguses leht 1 (baseURL sees määratud)
    // seejärel rekursiivelt küsime järgmist lehte Next abil.
    // &tempShips tähendab me täiendame algset objekti.
    // Ilma & operaatorita, annaksime me funktsioonile getJSON
    // edasi koopia mudelist.
    err := getJSON(pageUrl, &tempShips)
    if err != nil {
        return err
    }

    // Lisame kõik vastuse kosmoselaevad algsesse massiivi
    // [] on tegelikult slice. Erinevus seisneb tavalise C
    // massiivi vahel selles, et tal on lisafunktsionaalsusi
```

```
    // mis võimaldab kergemini massiive hallata. Näiteks append()
    for i := 0; i < len(tempShips.Results); i++ {
        ships.Results = append(ships.Results, tempShips.Results[i])
    }

    // Küsime järgmist lehte kui on olemas ja lõpetame
    // töö kui enam ei ole.
    ships.Next = tempShips.Next

    if ships.Next != "" {
        err = ships.fill(ships.Next)
    }

    return err
}
```

database.go

```
package main

import (
    "encoding/json"
    "fmt"
    "strconv"
    "strings"

    scribble "github.com/nanobox-io/golang-scribble"
)

// Kirjuta kosmoselaevad andmebaasi
func insertStarships(ships []Starship) error {
    // Loome uue andmebaasi.
    // Scribble on väga mugav JSON andmebaas prototüüpimiseks.
    // Tabelid kujutavad endas kaustu ja faili nimi on ID.
    // Rakenduse käivitamisel luuakse algkoodi kausta "starship" kaust
    // mille sissu kirjutatakse kõik päritud kosmoselaevad.
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    for i := 0; i < len(ships); i++ {
        // Alternatiivne käekiri veahalduseks
        if err := database.Write("starship", ships[i].Name, ships[i]); err
!= nil {
            fmt.Println("Database error", err)
            return err
        }
    }
    return err
```

```
}

func insertRebel(rebel Rebel) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if rebel.ID == 0 {
        // Kui tabelit pole loodud ignoreerime
        rebels, _ := database.ReadAll("rebel")
        count := len(rebels)
        count++
        rebel.ID = count
    }

    if rebel.Starship == (Starship{}) {
        starships, err := database.ReadAll("starship")
        if err != nil {
            fmt.Println("Error", err)
        }

        // Filtreerime välja kõik laevad mille hind v]i pikkus on
"unknown".
        // Kirjutame i üle viimase elemendiga.
        // Seejärel võtame "lõigu" mis lõpeb eelviimase elemendi juures.
        for i := 0; i < len(starships); i++ {
           if strings.Contains(starships[i], "unknown") {
                starships[i] = starships[len(starships)-1]
                starships = starships[:len(starships)-1]
            }
        }

        starship := starships[rebel.ID%len(starships)]
        rebel.Starship = Starship{}

        if err := json.Unmarshal([]byte(starship), &rebel.Starship); err
!= nil {
            fmt.Println("Error", err)
        }
    }

    if err := database.Write("rebel", strconv.Itoa(rebel.ID), rebel); err
!= nil {
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func getRebels() []Rebel {
```

```
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    records, err := database.ReadAll("rebel")
    if err != nil {
        fmt.Println("Error", err)
    }

    rebels := []Rebel{}
    for _, r := range records {
        rebel := Rebel{}
        if err := json.Unmarshal([]byte(r), &rebel); err != nil {
            fmt.Println("Error", err)
        }
        rebels = append(rebels, rebel)
    }

    return rebels
}

func deleteRebels() {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    if err := database.Delete("rebel", ""); err != nil {
        fmt.Println("Error", err)
    }
}

func insertChewbacca(chewie Chewbacca) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if err := database.Write("chewbacca", "Chewie", chewie); err != nil {
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func getChewbacca() Chewbacca {
   database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }
```

```
    chewie := Chewbacca{}
    if err := database.Read("chewbacca", "Chewie", &chewie); err != nil {
        fmt.Println("Error", err)
    }
    return chewie
}

func insertPalpatine(palpatine Palpatine) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if err := database.Write("palpatine", "Palpatine", palpatine); err !=
nil {
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func getPalpatine() Palpatine {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    palpatine := Palpatine{}
    if err := database.Read("palpatine", "Palpatine", &palpatine); err !=
nil {
        fmt.Println("Error", err)
    }
    return palpatine
}
```

Lisa 5 4. Iteratsiooni algkood

main.go

```
package main

import (
    "encoding/json"
    "fmt"
    "strconv"
)

const chewieThreshold = 20

type Fighter interface {
    CanFight() bool
    Name() string
    GetStrength() int
}

type Chewbacca struct {
    Hunger   int `json:"hunger"`
    Strength int `json:"strength"`
}

func (c *Chewbacca) CanFight() bool {
    return c.Hunger > chewieThreshold
}

func (c *Chewbacca) Name() string {
    return "Chewie"
}

func (c *Chewbacca) GetStrength() int {
    if c.CanFight() {
        return c.Strength
    }
    return 0
}

type Rebel struct {
    // Kui ID on tühi näiteks POST päringul, siis jäetakse
    // väli täitmata ja määratakse autmaatselt süsteemi poolt.
    ID       int      `json:"id, omitempty"`
    Credits  int      `json:"credits"`
    Starship Starship `json:"starship"`
    Strength int      `json:"strength"`
}

func (r *Rebel) CanFight() bool {
```

```
    cost, err := r.Starship.FuelCost()
    if err != nil {
        fmt.Println("Cost: %f", cost)
        panic(err)
    }

    return float32(r.Credits) > cost
}

func (r *Rebel) Name() string {
    return "Rebel nr. " + strconv.Itoa(r.ID) + ", owner of " +
r.Starship.Name
}

func (r *Rebel) GetStrength() int {

    if r.CanFight() {
        return r.Strength
    }

    return 0
}

type Palpatine struct {
    Strength int `json:"strength"`
}

func (p *Palpatine) Name() string {
    return "Bad guy Palpatine"
}

func (p *Palpatine) FightPalpatine(f Fighter) bool {
    p.Strength -= f.GetStrength()
    return p.Strength <= 0
}

func FightResult() string {
    rebels := GetRebels()
    chewie := GetChewbacca()
    palpatine := GetPalpatine()

    winner := palpatine.Name()
    totalStrength := chewie.GetStrength()

    if palpatine.FightPalpatine(&chewie) {
        winner = chewie.Name()
    }

    for _, rebel := range rebels {
        totalStrength += rebel.GetStrength()
        if palpatine.FightPalpatine(&rebel) {
            winner = rebel.Name()
```

```
        }
    }
    bytes, _ := json.Marshal(struct {
        WhoWon            string `json:"who_won"`
        PalpatineStrength int    `json:"palpatine_strength"`
        TotalStrength     int    `json:"total_strength"`
    }{
        WhoWon:            "The winner is: " + winner,
        PalpatineStrength: palpatine.Strength,
        TotalStrength:     totalStrength,
    })

    return string(bytes)
}

// Nagu Java keeleski, main() on alati esimene funktsioon mida vaikimisi
käivitatakse
func main() {
    starShips := Starships{}

    // Pärime kõik kosmoselaevad kolmanda osapoole APIst
    err := starShips.Fill(baseURL)
    if err != nil {
        // panic prindib detailse stacktracei
        panic(err)
    }

    // Salvestame andmebaasi kosmoselaevad
    InsertStarships(starShips.Results)

    // Loome serveri vastavate lõpp-punktidega
    Router()
}
```

starship.go

```
package main

import (
    "encoding/json"
    "net/http"
    "strconv"
    "time"
)

// Esmane API URL kust infot küsida
const baseURL string = "https://swapi.co/api/starships/?page=1"

// Loome uue HTTP klienti
// NB! & ja * operaatorid käituvad täpselt samasuguselt nagu C keele
viited
// Kui räägitakse, et Go keeles puudub reference tüüpi viide, siis seda
peetakse silmas.
// Erinevus seisneb selles, et ei pea ise manuaalselt mälu allokeerima.
var httpClient = &http.Client{Timeout: 10 * time.Second}

// Objekt mille vastu me hakkame API vastust kaardistama.
// Tahame kaardistada ainult kahte välja.
// Kuna API serveerib kosmoselaevu üle mitme lehe, on meil vaja antud
// mudelit. Next viitab järgmisele lehele.
type Starships struct {
    Next    string     `json:"next"`
    Results []Starship `json:"results"`
}

// API vastus on mitme tasemeline. See mudel hoiab infot mida meil
tegelikult vaja.
type Starship struct {
    Name          string `json:"name"`
    CostInCredits string `json:"cost_in_credits"`
    Length        string `json:"length"`
}

// Täiendame mudelit uue meetodiga.
// Hinna ja laevapikkuse jagatis ei peegelda muidugi
// reaaleluliselt üldse kütusehinda, aga API oli kahjuks
// selle koha pealt puudulik.
func (s *Starship) FuelCost() (float32, error) {
    num1, err := strconv.ParseFloat(s.CostInCredits, 32)
    if err != nil {
        return 0, err
    }

    num2, err := strconv.ParseFloat(s.Length, 32)
    if err != nil {
        return 0, err
```

```
    }

    return float32(num1 / num2), nil
}

// JSON paketi Decode sisendit
// ootab tüüpi mille vastu kaardistada, antud puhul Starships tüüp.
// interface{} on alternaative lahendus paljudes kohtades Java
// generics võimalustele. Erinevalt Java generic tüübist, ei vaja
// interface{} implementatsiooni. Isiklikult leian see teeb koodi
// kergemini loetavaks ja paremini arusaadavaks.

// func name(params...) [return type]
func GetJSON(url string, target interface{}) error {
    // Go keeles puudub exception. Selle asemel funktsioonid
    // on võimelised tagastama kaks väärtust, millest teine on alati
    // viga. Vea sisu kontrollitakse kohe selle määramise all.
    // Seda lähenemist on laialt kritiseeritud, aga minu jaoks teeb
    // see nii testimise kui navigeerimise kergemaks.
    r, err := httpClient.Get(url)
    if err != nil {
        return err
    }

    // Body on tüüpi io.ReadCloser
    // Javas me tavaliselt sulgeme IO tegevusi finally blokis
    // defer teeb sisuliselt sama, paneb IO sulgemise pinu lõppu
    defer r.Body.Close()
    return json.NewDecoder(r.Body).Decode(target)
}

// Lisame rekursiivselt kosmoselaevu Starships.Result massiivi
// Seekord tegu meetodiga, kuna esineb receiver

// func [receiver] name(params...) [return type]
func (ships *Starships) Fill(pageUrl string) error {
    // Loome vaheobjekti salvestamaks uue API lehe sisu
    tempShips := Starships{}

    // Pärime API käest alguses leht 1 (baseURL sees määratud)
    // seejärel rekursiivelt küsime järgmist lehte Next abil.
    // &tempShips tähendab me täiendame algset objekti.
    // Ilma & operaatorita, annaksime me funktsioonile getJSON
    // edasi koopia mudelist.
    err := GetJSON(pageUrl, &tempShips)
    if err != nil {
        return err
    }

    // Lisame kõik vastuse kosmoselaevad algsesse massiivi
    // [] on tegelikult slice. Erinevus seisneb tavalise C
    // massiivi vahel selles, et tal on lisafunktsionaalsusi
```

```
    // mis võimaldab kergemini massiive hallata. Näiteks append()
    for i := 0; i < len(tempShips.Results); i++ {
        ships.Results = append(ships.Results, tempShips.Results[i])
    }

    // Küsime järgmist lehte kui on olemas ja lõpetame
    // töö kui enam ei ole.
    ships.Next = tempShips.Next

    if ships.Next != "" {
        err = ships.Fill(ships.Next)
    }

    return err
}
```

database.go

```
package main

import (
    "encoding/json"
    "fmt"
    "strconv"
    "strings"

    scribble "github.com/nanobox-io/golang-scribble"
)

// Kirjuta kosmoselaevad andmebaasi
func InsertStarships(ships []Starship) error {
    // Loome uue andmebaasi.
    // Scribble on väga mugav JSON andmebaas prototüüpimiseks.
    // Tabelid kujutavad endas kaustu ja faili nimi on ID.
    // Rakenduse käivitamisel luuakse algkoodi kausta "starship" kaust
    // mille sissu kirjutatakse kõik päritud kosmoselaevad.
   database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    for i := 0; i < len(ships); i++ {
        // Alternatiivne käekiri veahalduseks
        if err := database.Write("starship", ships[i].Name, ships[i]); err
!= nil {
            fmt.Println("Database error", err)
            return err
        }
    }
    return err
}

func InsertRebel(rebel *Rebel) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if rebel.ID == 0 {
        // Kui tabelit pole loodud ignoreerime
        rebels, _ := database.ReadAll("rebel")
        count := len(rebels)
        count++
        rebel.ID = count
    }
```

```
    if rebel.Starship == (Starship{}) {
        starships, err := database.ReadAll("starship")
        if err != nil {
            fmt.Println("Error", err)
        }

        // Filtreerime välja kõik laevad mille hind v]i pikkus on
"unknown".
        // Kirjutame i üle viimase elemendiga.
        // Seejärel võtame "lõigu" mis lõpeb eelviimase elemendi juures.
        for i := 0; i < len(starships); i++ {
            if strings.Contains(starships[i], "unknown") {
                starships[i] = starships[len(starships)-1]
                starships = starships[:len(starships)-1]
            }
        }

        starship := starships[rebel.ID%len(starships)]
        rebel.Starship = Starship{}

        if err := json.Unmarshal([]byte(starship), &rebel.Starship); err
!= nil {
            fmt.Println("Error", err)
        }
    }

    if err := database.Write("rebel", strconv.Itoa(rebel.ID), rebel); err
!= nil {
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func GetRebels() []Rebel {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    records, err := database.ReadAll("rebel")
    if err != nil {
        fmt.Println("Error", err)
    }

    rebels := []Rebel{}
    for _, r := range records {
        rebel := Rebel{}
        if err := json.Unmarshal([]byte(r), &rebel); err != nil {
            fmt.Println("Error", err)
        }
        rebels = append(rebels, rebel)
```

```
    }

    return rebels
}

func DeleteRebels() {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    if err := database.Delete("rebel", ""); err != nil {
        fmt.Println("Error", err)
    }
}

func InsertChewbacca(chewie *Chewbacca) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if err := database.Write("chewbacca", "Chewie", chewie); err != nil {
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func GetChewbacca() Chewbacca {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    chewie := Chewbacca{}
    if err := database.Read("chewbacca", "Chewie", &chewie); err != nil {
        fmt.Println("Error", err)
    }
    return chewie
}

func InsertPalpatine(palpatine *Palpatine) error {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
        return err
    }

    if err := database.Write("palpatine", "Palpatine", palpatine); err !=
nil {
```

```
        fmt.Println("Database error", err)
        return err
    }
    return err
}

func GetPalpatine() Palpatine {
    database, err := scribble.New("./", nil)
    if err != nil {
        fmt.Println("Database error", err)
    }

    palpatine := Palpatine{}
    if err := database.Read("palpatine", "Palpatine", &palpatine); err !=
nil {
        fmt.Println("Error", err)
    }
    return palpatine
}
```

router.go

```
package main

import (
    "net/http"

    "github.com/labstack/echo"
)

// Kaardistame 3 mudelit lõpp-punktideks
func Router() {
    e := echo.New()
    e.GET("/", func(c echo.Context) error {
        return c.String(http.StatusOK, FightResult())
    })

    e.POST("/rebel", func(c echo.Context) error {
        r := new(Rebel)
        if err := c.Bind(r); err != nil {
            return err
        }
        InsertRebel(r)
        return c.JSON(http.StatusCreated, r)
    })

    e.DELETE("/rebel", func(c echo.Context) error {
        DeleteRebels()
        return c.String(http.StatusOK, "Succesfully culled incursion")
    })

    e.POST("/chewbacca", func(c echo.Context) error {
        chewie := new(Chewbacca)
        if err := c.Bind(chewie); err != nil {
            return err
        }
        InsertChewbacca(chewie)
        return c.JSON(http.StatusCreated, chewie)
    })

    e.POST("/palpatine", func(c echo.Context) error {
        p := new(Palpatine)
        if err := c.Bind(p); err != nil {
            return err
        }
        InsertPalpatine(p)
        return c.JSON(http.StatusCreated, p)
    })

    e.Logger.Fatal(e.Start(":8080"))
}
```