

VILNIAUS UNIVERSITETAS  
MATEMATIKOS IR INFORMATIKOS FAKULTETAS  
PROGRAMŲ SISTEMŲ KATEDRA

**Priklausomybių valdymo sistemų analizė**  
**Analysis of Application-level Dependency Management Systems**

Kursinis darbas

Atliko:	3 kurso 2 grupės studentė Greta Piliponytė	(parašas)
Darbo vadovas:	partn. doc. Vaidas Jusevičius	(parašas)

Vilnius – 2019

## TURINYS

ĮVADAS .....	3
1. GO KALBOS PRIKLAUSOMYBIŲ VALDYMO SISTEMA .....	4
1.1. Priklausomybių valdymo Go kalboje istorija.....	4
1.2. Priklausomybių versijavimas Go sistemoje.....	4
1.3. Priklausomybių versijų pasirinkimas Go sistemoje .....	5
2. NPM PRIKLAUSOMYBIŲ VALDYMO SISTEMA .....	7
2.1. Priklausomybių versijavimas NPM sistemoje .....	7
2.2. Priklausomybių versijų pasirinkimas NPM sistemoje.....	7
3. MAVEN PRIKLAUSOMYBIŲ VALDYMO SISTEMA .....	9
3.1. Tranzityvių priklausomybių valdymas Maven sistemoje .....	9
3.1.1. Priklausomybių mediacija.....	9
3.1.2. Priklausomybių valdymas .....	9
3.1.3. Pašalintos bei pasirenkamosios priklausomybės .....	9
REZULTATAI IR IŠVADOS .....	11
ŠALTINIAI .....	12

# **Įvadas**

Čia kažkada bus įvadas (tikėkimės) ...

# 1. Go kalbos priklausomybių valdymo sistema

Go programavimo kalboje (dar žinomoje kaip Golang), numatytasis priklausomybių valdymo įrankis yra „go get“ komanda. Naudodamasis šia komanda, vartotojas gali atsisiųsti vystomam projektui reikalingas priklausomybes. Ateinančiuose skyriuose aptariami naujausi „go get“ pokyčiai ir šių pokyčių atnešami privalumai.

## 1.1. Priklausomybių valdymo Go kalboje istorija

Nuo pat Go kalbos išleidimo 2009 metais, jos naudotojų tarpe atsirado poreikis dalintis savo, bei naudoti kitų vartotojų sukurtus paketus. Šiam tikslui Go inžinieriai sukūrė „GOINSTALL“ komandą, leidžiančią atsisiųsti norimus paketus iš tokių programinio kodo saugyklų kaip Github ar Bitbucket. Neilgai trukus „GOINSTALL“ pakeitė „go get“ komanda, tačiau abi šios komandos turėjo didžiulį trūkumą – jose nebuvo paketo versijos sąvokos. Tai reiškė, jog naudojant „go get“ vartotojas visada gaus naujausią šio paketo kopiją <sup>[COX18a]</sup>.

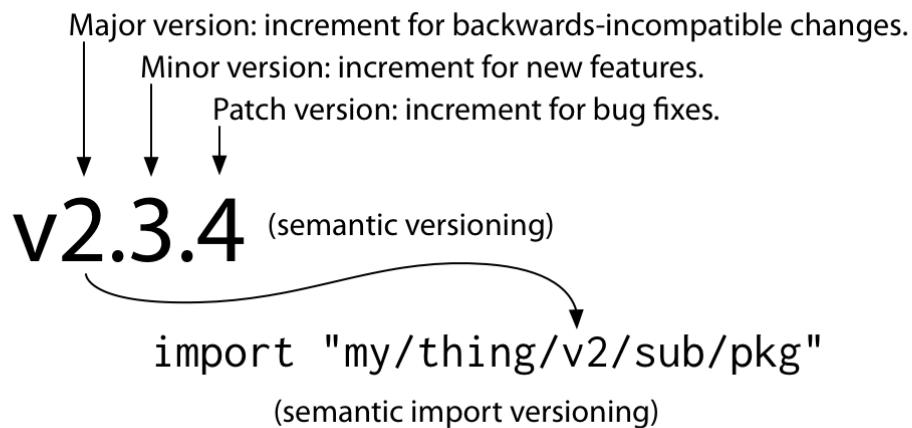
Negalėjimas pasirinkti paketo versijos kelia dvi pagrindines problemas. Pirmoji problema yra negalėjimas užtikrinti stabilaus programos surinkimo (ang. stable build), antroji – nėra galimybių užtikrinti, kad pokyčiai naujoje paketo versijoje bus atgaliai suderinami (ang. backwards-compatible).

Go priklausomybių valdymo trūkumai bandyti spręsti kopijuojant atsiųstus paketus ir laikant juos lokaliai – šį procesą automatizuoti sukurta daug įrankių, tokių kaip goven, godep, gb. Lokalus paketų laikymas išsprendė tik stabilaus programos surinkimo (ang. stable build) problemą. Priklausomybių versijų atgalinio suderinamumo problema vis dar nebuvo išspręsta <sup>[COX18a]</sup>.

Go kalboje buvo poreikis įvesti tiesioginį paketų versijų valdymą pačioje „go get“ komandoje ir nebepasikliauti trečiųjų šalių įrankiais. Taip Go inžinieriai pradėjo dirbti prie Vgo – pirmojo apie priklausomybių versijas žinančios (ang. version-aware) Go komandos prototipo <sup>[COX18b]</sup>. Go kalbos 1.11 versijoje pradėtas pirminis Vgo prototipe pristatytų idėjų palaikymas <sup>[GOLANG19]</sup>.

## 1.2. Priklausomybių versijavimas Go sistemoje

Go inžinierių pasiūlymas dėl version-aware Go komandos implementacijos apsprendžia, kaip naujoje Go bus versijuojamos priklausomybės. Trečiasis šio pasiūlymo punktas nurodo, kad atnaujintoje Go bus naudojamas semantinis importų versijavimas (ang. semantic import versioning) <sup>[COX18c]</sup>. Semantiniu importų versijavimu siekiama, jog su kiekvienam atgaliai nesuderinamui paketo pakeitimui bus priskiriamas skirtingas importavimo kelias (ang. import path) su specifikuota pagrindine (ang. major) versija, pavyzdžiui, „github.com/greta/foo/v2“.



1 pav. Semantinis importų versijavimas [COX18d]

Ketvirtasis Go pasiūlymo punktas, importų suderinamumo taisyklė, papildė prieš tai pasiūlyme pristatytą semantinio importų versijavimo idėją. Ši taisyklė teigia: jei senas paketas ir naujas paketas turi tą patį importavimo kelią, tuomet naujas paketas privalo būti atgaliai suderinamas su senuoju [COX18c]. Importų suderinamumo taisyklė paketų autoriams nustato griežtas ribas, kokie pakeitimai leidžiami nekeičiant paketo importavimo kelio ir kokius pakeitimus įvykdžius būtina kurti naują importavimo kelią.

Naudojant semantinį paketų versijavimą bei laikantis importų suderinamumo taisyklės tikimasi išspręsti prieš tai „go get“ komandoje buvusią nestabilaus API problemą – paketų naudotojams suteikiama garantija, kad atnaujinant priklausomybes jų naudojamų paketų metodai nesikeis.

### 1.3. Priklausomybių versijų pasirinkimas Go sistemoje

Nuo pat „go get“ pristatymo, viena didžiausių šios komandos problemų buvo nežinojimas apie valdomų paketų versijas. Senoji „go get“ komanda turėjo du priklausomybių versijų pasirinkimo algoritmus. Pirmasis, Go numatytasis algoritmas, „go get B“ metu atsiųsdavo naujausią paketo B versiją bei naujausias B priklausomybes, kurių nebuvo turima lokaliai. Antrasis algoritmas įvykdžius „go get -u B“ atsiųsdavo naujausią B, bei visas naujausias jos tranzityvių priklausomybių versijas [COX18d].

Abu šie algoritmai netenkino vartotojų bei kėlė daug klaidų. Naudojant pirmąjį algoritmą, kilo grėsmė, jog lokaliai turimos priklausomybės bus per senos ir neveiks su naujai atsiųstomis priklausomybėmis. Antrasis algoritmas taip pat nebuvo visiškai saugus, nes buvo galimybė, jog naujausios priklausomybių versijos nebus tarpusavyje sutapatinamos (ang. compatible) [COX18e].

Suprasdami „go get“ priklausomybių versijų pasirinkimo algoritmų keliamas problemas, Go inžinieriai į pasiūlymą dėl version-aware Go komandos įtraukė ir naują algoritmą priklausomybių versijų pasirinkimui. Šis algoritmas vadinasi „minimal version selection“ ir siūlo lyg šiol mažai naudotą priklausomybių versijų pasirinkimo mechanizmą – pasirinkti seniausią leidžiamą paketo versiją. Dauguma šiuolaikinių priklausomybių valdymo sistemų, tokių kaip dep ar cargo, naudoja

priešingą algoritmą – renkasi naujausią leidžiamą priklausomybės versiją [COX18a, COX18f].

Russ Cox, vienas pagrindinių Go kūrėjų, teigia, cargo bei dep naudojamas algoritmas yra klaidingas dėl dviejų priežasčių. Pirmoji priežastis yra tai, jog naujausia leidžiama versija gali nuolat kisti bei būti nestabili, antroji – klaidos atveju vartotojui reikia skirti papildomo laiko uždrausti naudoti specifinių versijų priklausomybes [COX18a].

Go inžinierių pasirinktas „minimal version selection“ algoritmas turi kelis pranašumus. Šis algoritmas užtikrina, jog visada su ta pačia „go get“ komanda bus gaunamos tų pačių versijų priklausomybės. Garantija, jog projekto priklausomybės nesikeis, leidžia užtikrinti, jog programos surinkimo rezultatas visada bus toks pats, tiek programų sistemos kūrimo metu, tiek sistemos produkcinėje aplinkoje [COX18a]. „Minimal version selection“ taip pat leidžia apsisaugoti nuo naujausiose paketų versijose galinčių būti klaidų – jei paketo A naujausiose versijoje yra klaida, tiek A paketo autorius, tiek kitų paketų, naudojančių A, autoriai turi laiko ištaisyti klaidą bei uždrausti naudoti šią triką turinčią versiją [COX18e].

## 2. NPM priklausomybių valdymo sistema

NPM yra priklausomybių valdymo sistema, naudojama JavaScript aplikacijose. Ši sistema turi didžiausią priklausomybių registrą pasaulyje, ja naudojantis galima atsisiųsti paketus arba Node modulius.<sup>[NPMa]</sup>

### 2.1. Priklausomybių versijavimas NPM sistemoje

NPM priklausomybėms versijuoti naudojama semantinio versijavimo sistema. Ši sistema versijuojamui vienetui suteikia X.Y.Z formos versiją, kurioje X reiškia pagrindinę (ang. major) versiją, Y – antraeilę (ang. minor) versiją, Z – pataisymų (ang. patch) versiją. Pataisymų (ang. patch) ir antraeilių (ang. minor) versijų pasikeitimai yra atgaliai suderinami ir yra saugūs naudoti projektuose su ta pačia pagrindine (ang. major) versija<sup>[NPMb]</sup>. Tuo tarpu pagrindinės versijos pasikeitimas reiškia atgaliai nesuderinamus pokyčius. Semantinis versijavimas leidžia vartotojui susidaryti lūkesčius naujoms priklausomybių versijoms bei package.json faile apibrėžti kurios priklausomybių versijos ir kurie jų atnaujinimai yra leidžiami projekte. Ši versijavimo sistema išsprendžia nestabilaus API problemą.

### 2.2. Priklausomybių versijų pasirinkimas NPM sistemoje

Norint naudotis NPM priklausomybėmis, projekte būtina turėti package.json failą. Šį failą sudaro projekto meta-data, tarp kurios nurodomos ir reikiamos priklausomybės bei jų versijos. Package.json faile išskirtos dvi priklausomybių grupės – „dependencies“ (projekto priklausomybės) bei „devDependencies“ (projekto kūrimui ir testavimui reikalingos priklausomybės). Kiekvienai šiame faile nurodytai priklausomybei nurodoma ir jos versija<sup>[GAU18]</sup>. NPM leidžia nurodyti ne tik tikslias priklausomybių versijas, bet ir leidžiamų versijų diapazoną. Tildės ženklas ( ) leidžia naujesnes pataisymų (ang. patch) versijas, stogelio ženklas (^) leidžia naujesnes antraeiles (ang. minor) arba pataisymų (ang. patch) versijas<sup>[PIT15]</sup>.

Galimybė package.json faile esančioms priklausomybėms suteikti leistinų versijų diapazoną turi privalumų ir trūkumų. Šis metodas leidžia vartotojui patogiau atnaujinti turimas priklausomybes – įvykdžius „npm update“ komandą automatiškai gaunamos naujausios leistinos priklausomybių versijos vartotojui net nekeitus package.json<sup>[PIT15]</sup>. Šio metodo trūkumas yra, jog nurodant leistinų priklausomybių versijų ribas, dirbant komandoje nėra galimybės užtikrinti, jog įvykdžius „npm install“ su tuo pačiu package.json bus gautos tų pačių versijų priklausomybės<sup>[PIT15]</sup>. Išlaikyti stabilias priklausomybių versijas projekte yra svarbu norint užtikrinti stabilų programos surinkimą (ang. stable build) – jei naujausioje priklausomybės versijoje būtų klaida, projektas veiktų klaidingai arba visai neveiktų.

Norint išspręsti nestabilaus programos surinkimo problemą, NPM 5 versijoje pradedamas naudoti package-lock.json failas. Šis failas nurodo tikslų priklausomybių medį – visų naudojamų priklausomybių, bei šių priklausomybių priklausomybių tikslias versijas<sup>[NPMc]</sup>. Package-lock.json generuojamas automatiškai pakeitus priklausomybių medį arba package.json failą<sup>[NPMc]</sup>. Atsiradus package-lock.json failui, NPM sistema vadovaujasi šiuo failu siųsdama priklausomybes. Kadangi

package-lock.json įrašomos tikslios priklausomybių versijos, išsprendžiama problema, jog kartu dirbantiems kolegoms ar CI/CD serveriui atsiunčiamos netinkamų versijų priklausomybės.



### 3. Maven priklausomybių valdymo sistema

Apache Maven yra daugiausiai Java projektams naudojama priklausomybių valdymo sistema <sup>[MAVENa]</sup>. Naudojant Maven projekte būtinas pom.xml failas, kuriame nurodomos norimos priklausomybės ir jų versijos. Maven priklausomybės yra vadinamos artefaktais – tai dažniausiai .jar tipo failai, įkelti į Maven repozitoriją. Pom.xml norimi artefaktai apibūdinami nurodant artefakto groupId, artifactId, version, kartu vadinamus artefakto koordinatėmis <sup>[MAVENb]</sup>. Maven Central repozitorija yra numatytoji vieta ieškoti priklausomybėms, tačiau pom.xml galima pridėti ir kitų repozitorijų (pavyzdžiui, kompanijų privačias repozitorijas), iš kurių bus siunčiamos priklausomybės <sup>[MAVENc]</sup>.

Maven taip pat palaiko daugiamodulininius (ang. multi-module) projektus, turinčius tėvinį modulį su šakniniu (ang. root) pom.xml, bei vaikinį modulius su savo pom.xml paveldinčiais priklausomybėmis iš tėvinio modulio <sup>[MAVENb]</sup>. Tai palengvina pom.xml failų palaikymą.

#### 3.1. Tranzityvių priklausomybių valdymas Maven sistemoje

Vartotojo patogumui pom.xml faile būtina nurodyti tik tiesiogines jo projekto priklausomybes – įrašant reikalingas tiesiogines priklausomybes automatiškai įrašomos ir jų priklausomybės, kitaip žinomos kaip tranzityvios (ang. transitive) priklausomybės. Maven turi daug ypatybių leidžiančių patogiau valdyti projekto tranzityvias priklausomybes.

##### 3.1.1. Priklausomybių mediacija

Priklausomybių mediacija (ang. dependency mediation) yra vienas iš Maven siūlomų funkcionalumų valdyti projekto tranzityvias priklausomybes. Tai algoritmas, nustatantis, kuri tranzityvios priklausomybės versija turi būti atsiųsta, jeigu priklausomybių medyje aptikta keletas skirtingų to paties artefakto versijų. Priklausomybių mediacija artefakto versiją pasirenka pagal tai, kuri artefakto versija priklausomybių medyje yra aukščiausia. Jei tame pačiame priklausomybių medžio lygyje sutinkamos kelios artefakto versijos, pasirenkama pirmoji paskelbta priklausomybės versija <sup>[MAVENd]</sup>.

##### 3.1.2. Priklausomybių valdymas

Maven siūlo ir kitus priklausomybių valdymo mechanizmus. Priklausomybių valdymas (ang. dependency management) leidžia konkrečiai specifiikuoti, kokios priklausomybių versijos bus naudojamos, jei tos priklausomybės atsidurs tarp projekto tranzityvių priklausomybių. Tai itin paranku, jei priklausomybių mediacijos (ang. dependency mediation) metu buvo atsiųstos ne tos versijos priklausomybės <sup>[MAVENd]</sup>.

##### 3.1.3. Pašalintos bei pasirenkamosios priklausomybės

Pašalintos priklausomybės (ang. excluded dependencies) suteikia galimybę uždrausti projektui atsisiųsti tam tikrų nurodytų priklausomybių, net jei tos priklausomybės priklauso projekto tranzityvioms priklausomybėms. Pasirenkamosios priklausomybės (ang. optional dependencies) –

leidžia ignoruoti tam tikras nurodytas tranzityvias priklausomybes iki kol projekto autorius aiškiai nenurodo naudoti šias pasirenkamas priklausomybes. Pašalintų bei pasirenkamųjų priklausomybių mechanizmai leidžia vartotojui sumažinti projekto priklausomybių medį, taip pagreitinant projekto programinio kodo surinkimą (ang. build) <sup>[MAVENd]</sup>.

## **Rezultatai ir išvados**

Čia kažkada bus rezultatai (tikėkimės)

## Šaltiniai

- COX18a - Russ Cox, *Go += Package Versioning*, <https://research.swtch.com/vgo-intro>, vasario 20, 2018
- COX18b - Russ Cox, *cmd/go: add package version support to Go toolchain*, <https://github.com/golang/go/issues/24301>, kovo 7, 2018
- GOLANG19 - *Go 1.11 Release Notes*, <https://golang.org/doc/go1.11>
- COX18c - Russ Cox, *Proposal: Versioned Go Modules*, <https://go.googlesource.com/proposal/+/master/design/versioned-go.md>, kovo 20, 2018
- COX18d - Russ Cox, *Semantic Import Versioning*, <https://research.swtch.com/vgo-import>, vasario 21, 2018
- COX18f - Russ Cox, *Cargo newest*, h
- <https://research.swtch.com/cargo-newest.html>
- NPMa - *About npm*, <https://docs.npmjs.com/about-npm/index.html>
- NPMb - *About semantic versioning*, <https://docs.npmjs.com/about-semantic-versioning>
- GAU18 - Shardendu Kumar Gautam, *Why package.json? | npm basics*, <https://medium.com/beginners-guide-to-mobile-web-development/why-package-json-npm-basics-cab3e8cd150>, kovo 12, 2018
- NPMc - *npm-package-lock.json*, <https://docs.npmjs.com/files/package-lock.json>
- MAVENa - *What is maven*, <https://maven.apache.org/what-is-maven.html>
- MAVENb - *Introduction to the POM*, <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>
- MAVENc - *Introduction to repositories*, <https://maven.apache.org/guides/introduction/introduction-to-repositories.html>
- MAVENd - *Introduction to the Dependency Mechanism*, <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>