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. PRIKLAUSOMYBIŲ VALDYMO SISTEMOS 4
   1.1. Priklausomybių valdymo sistemų poreikis 4
   1.2. Nagrinėti pasirinktos priklausomybių sistemos 4

2. GO KALBOS PRIKLAUSOMYBIŲ VALDYMO SISTEMA 5
   2.1. Priklausomybių valdymo Go kalboje istorija 5
   2.2. Priklausomybių versijavimas Go sistemoje 5
   2.3. Priklausomybių versijų pasirinkimas Go sistemoje 6

3. NPM PRIKLAUSOMYBIŲ VALDYMO SISTEMA 8
   3.1. Priklausomybių versijavimas NPM sistemoje 8
   3.2. Priklausomybių versijų pasirinkimas NPM sistemoje 8

4. MAVEN PRIKLAUSOMYBIŲ VALDYMO SISTEMA 10
   4.1. Tranzityvių priklausomybių valdymas Maven sistemoje 10
      4.1.1. Priklausomybių mediacija 10
      4.1.2. Priklausomybių valdymas 11
      4.1.3. Pašalintos bei pasirenkamosios priklausomybės 11
   4.2. Priklausomybių versijavimas Maven sistemoje 12

5. APTARTŲ PRIKLAUSOMYBIŲ VALDYMO SISTEMŲ PALYGIMAS 13
   5.1. Priklausomybių versijų pasirinkimas 13
   5.2. Stabilių priklausomybių versijų užtikrimas 13

REZULTATAI IR IŠVADOS 15

ŠALTINIAI 16

Įvadas

Čia bus įvadas

1. Priklausomybių valdymo sistemos

1.1. Priklausomybių valdymo sistemų poreikis

Programų sistemos dažnai susideda iš mažesnių, vieną paskirtį turinčių (ang. single purpose) modulių. Populiarėjant atvirojo programinio kodo (ang. open source) naudojimui, sistemos neretai priklauso nuo išorinių, kitų autorių sukurtų modulių [Pad17]. Toks programinio kodo daugkartinis panaudojimas turi daug privalumų - paspartinamas programų sistemų kūrimo procesas, daugiau laiko skiriama fokusuojantis į unikalias kuriamos sistemos užduotis.

Priklausomybių valdymo sistemos yra įrankiai, palengvinantys išorinių modulių naudojimą programų sistemose. Šios sistemos leidžia atsisiųsti tiesiogines projekto priklausomybes, dauguma jų taip pat turi mechanizmus gauti ir projekto tranzityvias priklausomybes (tiesioginių priklausomybių priklausomybes) [Pad17]. Naudojant priklausomybių valdymo sistemas vartotojui būtina nurodyti tik norimų tiesioginių priklausomybių sąrašą, tinkamos tranzityvių priklausomybių versijos bei iš kokių repozitorijos jos bus siunčiamos nustatoma automatiškai. Priklausomybių valdymo sistemos taip pat dažnai suteikia galimybę patogiai atnaujinti individualias priklausomybes arba visą projekto priklausomybių medį [Mat17].

Priklausomybių valdymo sistemos yra svarbios kuriant programų sistemas komandoje bei kuriamos programų sistemos nuolatinės integracijos bei nuolatinio tiekimo (ang. continuous integration and continuous delivery, trump. CI/CD) procesui. Modernios priklausomybių valdymo sistemos leidžia užtikrinti, jog visi komandos inžinieriai dirba su tų pačių versijų priklausomybėmis. Šios sistemos taip pat išlaiko vienodas priklausomybes tiek vystymo, tiek testinėse bei produkcinėje aplinkose [Mat17]. Taip sumažinama su netinkamomis priklausomybėmis susijusių programų sistemos trikių tikimybė.

1.2. Nagrinėti pasirinktos priklausomybių sistemos

Šiuo metu yra sukurta dešimtys skirtingų priklausomybių valdymo sistemų. Tolesniuose skyriuose bus nagrinėjamos Go, NPM bei Maven sistemos. Šios sistemos pasirinktos dėl kelių priežasčių. NPM turi didžiausią priklausomybių registrą iš visų priklausomybių valdymo sistemų ir yra vienas dominuojančių įrankių front-end sistemose. Tuo tarpu Go priklausomybių valdymo mechanizmas įdomus sistemos pertvarkymo metu priimtais unikaliais, iki šiol nenaudotais priklausomybių valdymo problemų sprendimais. Maven sistema yra šiek tiek senesnė nei Go ar NPM (išleista 2002 metais), tačiau siūlo daug mechanizmų efektyviau valdyti priklausomybes [Pad17]. Kiekviena iš pasirinktų sistemų turi savitų priklausomybių valdymo metodų, pavyzdžiui, visos trys iš šių sistemų turi skirtingus algoritmus tranzityvių priklausomybių versijoms gauti. Svarbus faktorius renkantis sistemas buvo ir tai, jog kiekviena iš minėtų sistemų yra aktyviai naudojama ir šiomis dienomis. Taigi, visos trys sistemos yra vis dar aktualios ir turi savų ypatumų, leidžiančių juos palyginti kursinio darbo eigoje.

2. 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.

2.1. 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 [Cox18b].

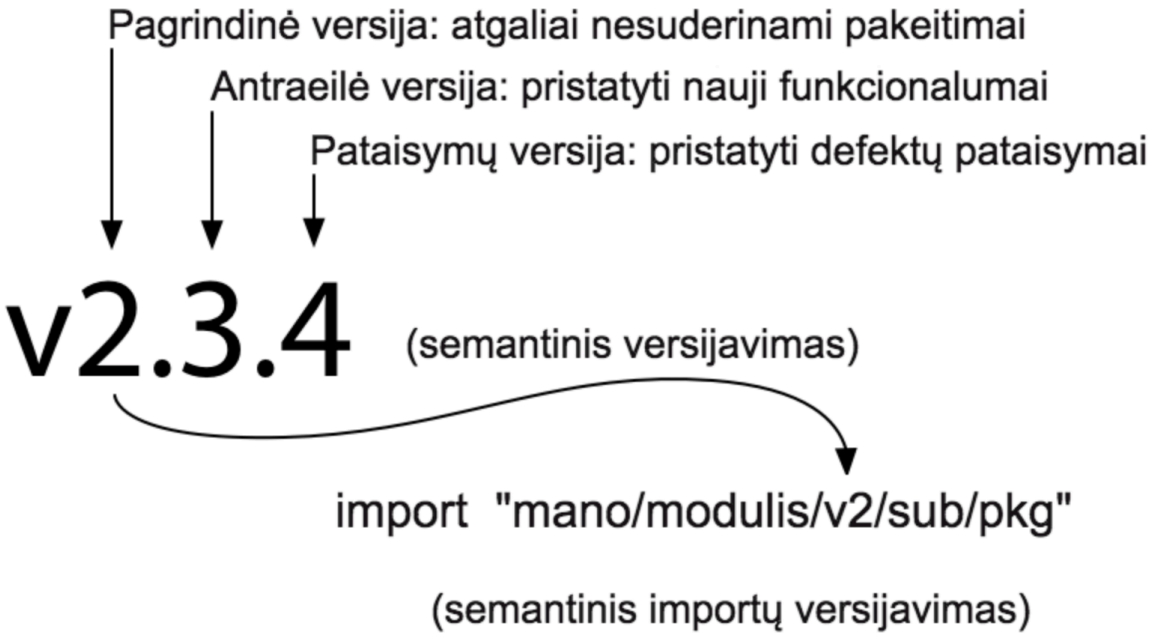
Negalėjimas pasirinkti paketo versijos kelia dvi pagrindines problemas. Pirmoji problema yra negalėjimas užtikrinti stabilaus programos surinkimo (ang. stable build), antroji - nėra galimybių užtikrinti, kad pokyčiai naujoje paketo versijoje bus atgaliai suderinami (ang. backwards-compatible).

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 [Cox18b].

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 [Cox19]. Go kalbos 1.11 versijoje pradėtas pirminis Vgo prototipe pristatytų idėjų palaikymas [Gol19].

2.2. 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) [Cox18d]. 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 [Cox18e]

Ketvirtasis Go pasiūlymo punktas, importų suderinamumo taisyklė, papildo 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 [Cox18d]. 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.

2.3. 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 [Cox18e].

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) [Cox18c].

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 [Cox18b] [Cox18a].

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 [Cox18b].

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 priklausomybes. Garantija, jog projekto priklausomybės nesikeis, leidžia užtikrinti, jog programos surinkimo rezultatas visada bus toks pats, tiek programų sistemos kūrimo metu, tiek sistemos produkcinėje aplinkoje [Cox18b]. „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 trikį turinčią versiją [Cox18c].

3. 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.[Npma].

3.1. 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 [Npmb]. 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.

3.2. 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 [Gau18]. 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 [Pit15].

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 [Pit15]. 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 [Pit15]. 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 [Npmc]. Package-lock.json generuojamas automatiškai pakeitus priklausomybių medį arba package.json failą [Npmc]. 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.

4. Maven priklausomybių valdymo sistema

Apache Maven yra daugiausiai Java projektams naudojama priklausomybių valdymo sistema [Mavd]. 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 [Mavc]. 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 [Mava].

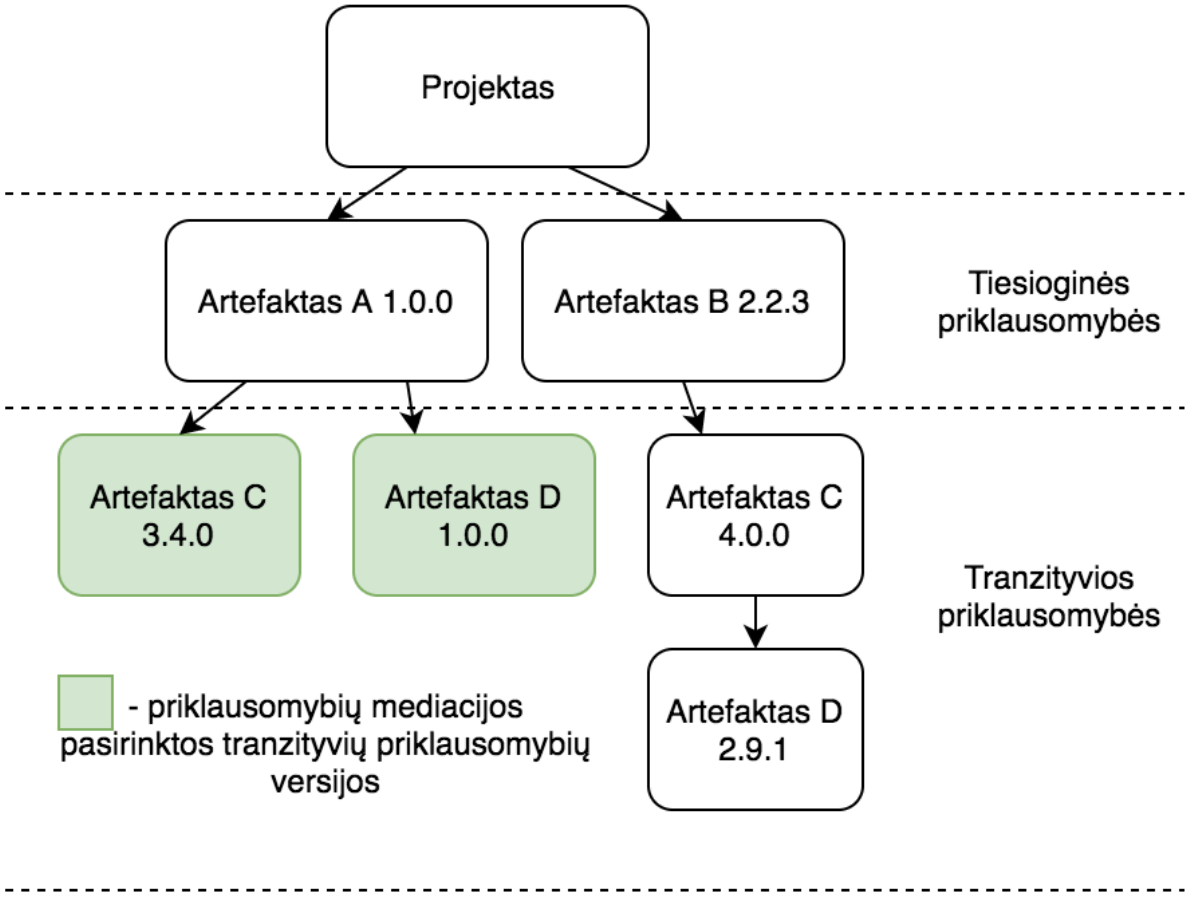
Maven taip pat palaiko daugiamodulininius (ang. multi-module) projektus, turinčius tėvinį modulį su šakniniu (ang. root) pom.xml, bei vaikinius modulius su savo pom.xml paveldinčiais priklausomybes iš tėvinio modulio [Mavc]. Tai palengvina pom.xml failų palaikymą.

4.1. 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 [Mavb]. Maven turi daug ypatybių leidžiančių patogiau valdyti projekto tranzityvias priklausomybes.

4.1.1. 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 [Mavb].



2 pav. Priklausomybių mediacija

4.1.2. Priklausomybių valdymas

Maven siūlo ir kitus priklausomybių valdymo mechanizmus. Priklausomybių valdymas (ang. dependency management) leidžia konkrečiai specifikuoti, kokios priklausomybių versijos bus naudojamos, jei tos priklausomybės atsidurs tarp projekto tranzityvių priklausomybių. Tai itin paranku, jei priklausomybių mediavimo (ang. dependency mediation) metu buvo atsiųstos ne tos versijos priklausomybės [Mavb].

4.1.3. 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. Pasirenkamos 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) [Mavb].

4.2. Priklausomybių versijavimas Maven sistemoje

Maven sistemoje dažniausiai laikomasi priklausomybių semantinio versijavimo modelio - artefakto versiją sudaro Major Version, Minor Version, Incremental Version (taip pat gali būti ir Build Number, Qualifier). Toks semantinis versijavimas priklausomybių naudotojui leidžia susidaryti lūkesčius, kurios priklausomybės versijos yra atgaliai suderinamos, o kurios - ne. Semantiniame versijavime tik pagrindinės (ang. major) versijos pokyčiai reiškia atgaliai nesuderinami, todėl vartotojas privalo atsargiai atlikti pagrindinių versijų atnaujinimus. Semantinis versijavimas leidžia išvengti nestabilaus API problemos, kuri buvo matoma senojoje Go priklausomybių valdymo sistemoje [Ora].

Maven pom.xml faile taip pat galima nurodyti leidžiamas artefaktų versijų ribas, iš kurių bus pasirenkama tuo metu didžiausia. Pavyzdžiui, (, 1.0] reiškia, jog artefakto versija bus automatiškai atnaujinama be vartotojo įsikišimo iki 1.0 versijos, vėliau jau reikės pačio vartotojo įsikišimo. Ši sintaksė buvo labiau naudojama Maven 2, tačiau vis dar sutinkama ir šiomis dienomis. Galimybė turėti automatinius versijų atnaujinimus reiškia, jog bet kada gali pasikeisti programinio kodo surinkimo rezultatas ar įvykti klaida. Dėl šios priežasties geriau vengti automatinio versijų atnaujinimo. Maven 3 taip pat dėl stabilus programinio kodo surinkimo problemų atsisakė LATEST bei RELEASE artefaktų versijų [Lig18].

5. Aptartų priklausomybių valdymo sistemų palygimas

5.1. Priklausomybių versijų pasirinkimas

Kiekviena iš darbe aptartų priklausomybių valdymo sistemų turi skirtingus priklausomybių versijų pasirinkimo metodus.

Pirmoji kursiniame darbe aprašyta Go priklausomybių valdymo sistema renkasi mažiausią leidžiamą priklausomybės versiją - šis algoritmas vadinamas „minimal version selection“. „Minimal version selection” prioritetizuojamas projekto priklausomybių versijų stabilumas ir nuspėjamumas, o ne jų naujumas.

Priešingai nei Go, NPM priklausomybių valdymo sistemos strategija yra (neturint package-lock.json failo) rinktis naujausias leidžiamas priklausomybių versijas. NPM teikia pirmenybę naujausių priklausomybių versijų naudojimui, net jei tai reiškia, jog sistemos vartotojas gali lengvai gauti priklausomybes su dar nežinomais trikiais.

Maven priklausomybių valdymo sistema siūlo keletą strategijų priklausomybių versijoms nustatyti. Tiesioginėms priklausomybėms, nurodytoms projekto pom.xml faile, kaip ir NPM sistemoje naudojamas didžiausios leidžiamos versijos algoritmas. Tranzityvių priklausomybių versijų pasirinkimo numatytasis metodas yra priklausomybių mediacija (ang. dependency mediation), kuri renkasi pirmąją priklausomybės versiją, esančią priklausomybių medyje. Naudojantis priklausomybių mediacija sunku nuspėti, kokių versijų tranzityvios priklausomybės bus gautos, šis metodas nėra toks nuspėjamas kaip Go ar NPM sistemų. Priklausomybių mediacijos metu gali būti gaunamos ir per senų, ir gerokai per naujų versijų priklausomybės, todėl Maven naudojamas ir priklausomybių valdymo (ang. dependency management) mechanizmas, leidžiantis specifikuoti konkrečias norimų tranzityvių priklausomybių versijas. Šis metodas nėra patogus, nes reikalauja žinoti konkrečias norimų trazityvių priklausomybių versijas, tačiau jis leidžia pataisyti priklausomybių mediacijos metu padarytas klaidas. Maven taip pat turi mechanizmų uždrausti atsiųsti nurodytas tranzityvias priklausomybes ar jas ignoruoti iki kol bus išreikštai nurodoma jas naudoti.

Apibendrinus, Go priklausomybių versijų pasirinkimo algoritmas yra saugus ir nuspėjamas, bet nėra įrašomos pačios naujausios priklausomybių versijos. NPM sistema leidžia patogiai gauti naujausias priklausomybių versijas, tačiau paaukojama dalis nuspėjamumo ir saugumo. Maven numatytasis trazityvių priklausomybių mediacijos algoritmas nėra nuspėjamas, todėl sistema siūlo ir priklausomybių valdymo mechanizmą, kurio metu atsiunčiamos specifikuotų versijų tranzityvios priklausomybės. Maven, kitaip nei NPM ir Go sistemos, suteikia galimybę vartotojui nenaudoti nurodytų tranzityvių priklausomybių, taip sumažinamas priklausomybių medis.

5.2. Stabilių priklausomybių versijų užtikrimas

Stabilių priklausomybių išlaikymas skirtingose aplinkose yra dar viena svarbi priklausomybių valdymo sistemų funkcija, kurią kursiniame darbe aptartos sistemos vykdo naudodamos skirtingais metodais.

Go priklausomybių valdymo sistema palaiko stabilias priklausomybes naudodama „minimal selection“ algoritmą, kurio metu pasirenkamos mažiausių leidžiamų versijų priklausomybės.

Šis metodas užtikrina, jog bus gaunamos tos pačios priklausomybės, nes seniausios leidžiamos priklausomybių versijos nekinta.

NPM pasirenka kitą, daugelyje šiuolaikinių priklausomybių valdymo sistemų populiarų metodą - priklausomybių rakinimą (ang. dependency locking). Naudojant priklausomybių rakinimą, projekto "lock" faile (package-lock.json) užfiksuojamas visas projekto priklausomybių medis bei užtikrina, jog instaliuojant priklausomybes bus replikuojamas būtent package-lock.json užfiksuotas priklausomybių medis.

Maven priklausomybių valdymo sistema neturi tvirtų mechanizmų, leidžiančių užtikrinti priklausomybių stabilumą. Sistemos vieninteliame manifest faile (pom.xml) leidžiama nurodyti ne tik konkrečias priklausomybių versijas, bet ir priklausomybių galimų versijų diapazoną, iš kurio naudojama naujausia. Tai reiškia, jog projekto priklausomybių medis, tiek vystymo, tiek produkcinėje gali būti nestabilus neatsargiam vartotojui nenurodžius konkrečių priklausomybių versijų. Vystant Maven projektą sunku išlaikyti stabilias priklausomybių versijas ir dėl priklausomybių mediacijos algoritmo, naudojamo gauti tranzityvioms projekto priklausomybėms, išreikštai nenurodytoms pom.xml failo „dependencyManagement" mazge. Priklausomybių mediacija naudoja priklausomybių medyje pirmą rastą priklausomybės versiją, kas reiškia, jog pridėjus naujas tiesines priklausomybes, gali netikėtai projekto tranzityvių priklausomybių versijos.

Taigi, Go ir NPM turi savitus, tačiau patikimus stabilių priklausomybių užtikrinimo metodus, ir šiuo aspektu yra pranašesni už Maven. Maven sistemoje nėra saugiklių priklausomybių versijoms užtikrinti (tokių kaip NPM priklausomybių rakinimas), todėl neatsargūs vartotojai gali susidurti su nestabilių priklausomybių sukeltais trikiais.

Rezultatai ir išvados

Čia bus rezultatai

Šaltiniai

[Cox18a] Russ Cox. Cargo newest, https://research.swtch.com/cargo-newest.html. 2018.

[Cox18b] Russ Cox. Go += package versioning, https://research.swtch.com/vgo-intro. 2018.

[Cox18c] Russ Cox. Opening keynote: go with versions, https://www.youtube.com/watch?v=F8nrpe0XWRg&t=1663s. 2018.

[Cox18d] Russ Cox. Proposal: versioned go modules, https://go.googlesource.com/proposal/+/master/design/24301-versioned-go.md. 2018.

[Cox18e] Russ Cox. Semantic import versioning, https://research.swtch.com/vgo-import. 2018.

[Cox19] Russ Cox. Cmd/go: add package version support to go toolchain, https://github.com/golang/go/issues/24301. 2019.

[Gau18] Shardendu Kumar Gautam. Why package.json? | npm basics, https://medium.com/beginners-guide-to-mobile-web-development/why-package-json-npm-basics-cab3e8cd150. 2018.

[Gol19] Golang. Go 1.11 release notes, https://golang.org/doc/go1.11. 2019.

[Lig18] Andrea Ligios. Use the latest version of a dependency in maven, https://www.baeldung.com/maven-dependency-latest-version. 2018.

[Mat17] Seun Matt. What are dependency managers?, https://medium.com/prodsters/what-are-dependency-managers-26d7d907deb8. 2017.

[Mava] Maven. Introduction to repositories, https://maven.apache.org/guides/introduction/introduction-to-repositories.html.

[Mavb] Maven. Introduction to the dependency mechanism, https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[Mavc] Maven. Introduction to the pom, https://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

[Mavd] Maven. What is maven, https://maven.apache.org/what-is-maven.html.

[Npma] Npm. About npm, https://docs.npmjs.com/about-npm/index.html.

[Npmb] Npm. About semantic versioning, https://docs.npmjs.com/about-semantic-versioning.

[Npmc] Npm. Npm-package-lock.json, https://docs.npmjs.com/files/package-lock.json.

[Ora] Oracle. Understanding maven version numbers, https://docs.oracle.com/middleware/1212/core/MAVEN/maven\protect_version.htm.

[Pad17] Arvind Padmanabhan. Dependency manager, https://devopedia.org/dependency-manager. 2017.

[Pit15] Panu Pitkämäki. Semver explained - why is there a caret (^) in my package.json?, https://bytearcher.com/articles/semver-explained-why-theres-a-caret-in-my-package-json. 2015.