

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1385

**Implementacija kontinuirane
isporuke programske podrške za
operacijski sustav iOS**

Ivan Rep

Zagreb, lipanj 2017.

Htio bih se zahvaliti izv. prof. dr. sc. Borisu Vrdoljaku na kontinuiranoj potpori i vodstvu kroz diplomski studij

SADRŽAJ

1. Uvod	1
2. Kontinuirana integracija	3
2.1. Priprema	5
2.1.1. Xcode Server	5
2.1.2. Homebrew	8
2.2. Izgradnja	9
2.2.1. Verzioniranje	10
2.2.2. Priprema sustava	21
2.2.3. Upravljanje ovisnostima	22
2.2.4. Izgradnja	26
2.3. Testiranje	29
2.3.1. Automatizacija testiranja	31
2.4. Osiguranje kvalitete	32
3. Kontinuirana dostava	35
3.1. Potpisivanje koda	38
3.1.1. Automatizacija potpisivanja koda	41
3.2. Arhiviranje	42
3.2.1. Automatizacija arhiviranja	44
3.3. Objava	44
3.3.1. Crashlytics	45
3.3.2. App Store	47
4. Kontinuirana isporuka	51
4.1. Tipovi isporuke	53
4.2. Odabir tipa isporuke	54
4.3. Priprema projekta za automatsku isporuku	56

4.4. Implementacija kontinuirane isporuke	58
4.5. Pregled rezultata implementacije kontinuirane isporuke	60
5. Zaključak	63
Literatura	65
Dodaci	70
A. Usporedba alata za implementaciju kontinuirane integracije	71
A.1. Prvi upitnik	72
A.2. Drugi upitnik	72
A.3. Ispitani sustavi	73
A.3.1. Jenkins	74
A.3.2. Xcode Server	75
A.3.3. CircleCI	76
A.3.4. Travis CI	77
B. Alat xcodebuild	80
B.1. Testiranje	81
B.2. Osiguranje kvalitete	82
B.3. Arhiviranje	82
C. Fastlane	84
C.1. Dohvat ovisnosti	85
C.2. Izgradnja	86
C.3. Testiranje	86

1. Uvod

Profesionalna izrada programske potpore za iOS operacijski sustav zahtijeva čestu isporuku različitih verzija aplikacije. Osim produkcijske verzije programske potpore, čija učestalost isporuke može varirati od nekoliko puta tjedno do jednom u nekoliko mjeseci, u sklopu razvoja isporučuju se i druge verzije, na primjer verzija koja prikazuje trenutno stanje razvoja i verzija za testiranje. Učestalost isporuke navedenih tipova također značajno varira od tima do tima.

Kako je iOS operacijski sustav relativno mlad, isporuka se još uvijek uglavnom obavlja ručno. Ručna je isporuka programske potpore zahtjevna i podložna ljudskoj pogrešci što dovodi do njenog rijetkog obavljanja. Rijetko obavljanje isporuke uzrokuje zajedničku isporuku većeg broja promjena što dovodi do lošije kvalitete produkta i sporije isporuke novih funkcionalnosti.

Ovaj je problem moguće riješiti automatizacijom procesa isporuke programske potpore za iOS operacijski sustav. Prije isporuke programsku potporu potrebno je arhivirati, a zatim je dobivenu arhivu potrebno objaviti na željenoj platformi. Arhiviranje je proces pripreme iOS projekta za isporuku. Proces arhiviranja i proces objave značajno ovise o platformi na kojoj se objavljuje programska potpora.

Prije arhiviranja programske potpore potrebno je istu izgraditi te utvrditi zadovoljava li postavljene zahtjeve. Zahtjevi se najčešće izražavaju u obliku testova, ali mogu poprimiti i druge oblike.

Na kraju, potrebno je utvrditi pomoću kojeg je tipa isporuke potrebno isporučiti pojedinu verziju programske potpore.

Navedene se funkcionalnosti mogu svrstati u tri dobro prihvaćene prakse u sklopu računalnog inženjerstva: kontinuiranu integraciju, kontinuiranu dostavu i kontinuiranu isporuku.

Kontinuirana integracija automatizira procese izgradnje, testiranja i osiguranja kvalitete kako bi poboljšala kvalitetu programske potpore. Ovako automatizirani proces provodi se nad svakom novom verzijom programske potpore čime se osigurava njezina ispravnost. Također, kontinuirana integracija potiče učestalu integraciju radnih kopija

s glavnom kopijom kako bi se izbjegla pojava konflikta pri integraciji.

Kontinuirana dostava automatizira proces isporuke programske potpore. U sklopu razvoja programske potpore za iOS operacijski sustav, praksa uključuje automatizaciju potpisivanja koda, arhiviranja i objave. Praksa automatiziranjem navedenih procesa nastoji olakšati proces isporuke te tako povećati učestalost objave novih funkcionalnosti. Svaka funkcionalnost koja je razvijena, a nije objavljena, predstavlja gubitak timu. Dokle je god razvijena funkcionalnost neobjavljena, ona ne donosi korist zbog koje je razvijena.

Kontinuirana isporuka automatskom isporukom produkta nastoji što više smanjiti vrijeme proteklo od završetka razvoja do objave funkcionalnosti, pa čak i u potpunosti eliminirati ga. Promjene se ne isporučuju direktno u produkciju već prolaze više faza isporuke. Prvo se isporučuju razvojnom timu, zatim timu za osiguranje kvalitete, vanjskim testerima, a prema potrebi i ograničenom broju stvarnih korisnika. Tek se nakon svih ovih faza promjena isporučuje u produkciju. Kontinuirana integracija ovime nastoji povećati kvalitetu programske potpore, ubrzati isporuku promjena te smanjiti opterećenje razvojnog tima.

U sklopu ovog rada definiram i implementiram navedene prakse za iOS operacijski sustav.

Cilj rada nije samo u teoriji ispitati mogućnost implementacije sustave koji obavlja navedene procese, već ga iskoristiti na stvarnim projektima. Sustav mora biti jednostavan za implementaciju i korištenje te prenosiv na druge projekte. Zbog ovog ograničenja ne samo da nastojim automatizirati isporuku i integraciju, već i automatizirati te proces dodavanja automatskih procesa projektu.

Rad je strukturiran po praksama redoslijedom koji je korišten u ovom uvodu. Svaku od praksi prvo promatram iz opće perspektive i potom definiram i implementiram za iOS operacijski sustav. Rad završava pregledom ostvarenih funkcionalnosti i zaključkom.

2. Kontinuirana integracija

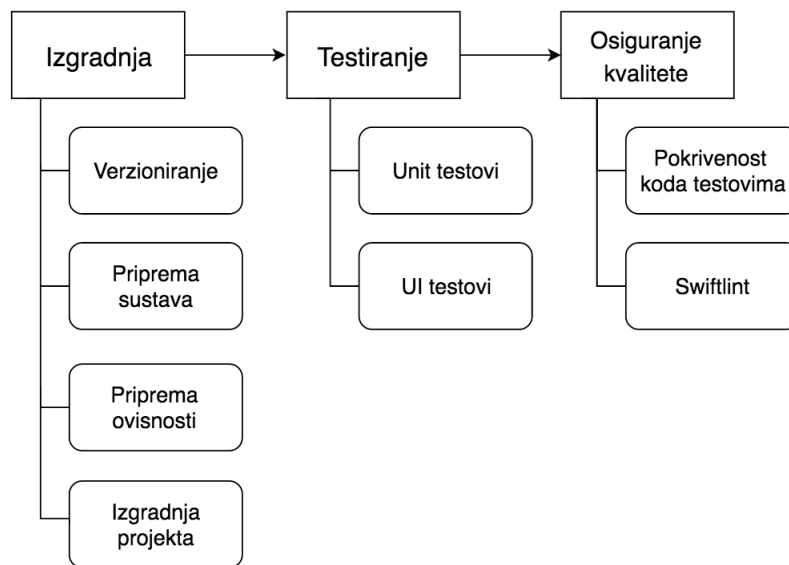
Kontinuirana integracija praksa je spajanja razvojnih kopija koda s glavnom kopijom nekoliko puta dnevno. Termin je prvi predložio i iskoristio Grady Booch 1991. godine tijekom opisa metode danas poznate kao Boochova metoda (engl. *Booch method*)[14].

Glavni cilj metode jest smanjivanje broja konflikata prilikom spajanja različitih verzija koda. Tijekom razvoja članovi tima preuzimaju zajedničku (engl. *master*) kopiju izvornog koda (engl. *source code*) te nad njom obavljaju promjene. Lokalna kopija izvornog koda naziva se *razvojnomo kopijom izvornog koda*. Članovi tima pomoću zajedničke kopije izvornog koda stvaraju razvojnu kopiju izvornog koda na vlastitom računalu.

Nakon implementacije željenih promjena programer vlastitu razvojnu kopiju spaja s izvornom kopijom. Ovaj postupak nazivamo integracija izvornog koda. Ako zajednička kopija izvornog koda nije bila mijenjana otkako ju je programer preuzeo, onda je promjene moguće jednostavno dodati zajedničkoj kopiji. Međutim, ako je zajednička kopija izvornog koda izmijenjena, onda je potrebno na neki način spojiti lokalne i promjene koje se već nalaze na glavnoj kopiji izvornog koda.

Što je duže programerova kopija izdvojena, to je veća vjerojatnost da je izvorna kopija u međuvremenu izmijenjena. Što se kopije više razlikuju, to je teže obaviti njihovo spajanje. Ujedno, spajanje često nije moguće obaviti automatski. Ova se pojava naziva konflikt i javlja se prilikom spajanja kopija koje su istovremeno modificirale isti dio istog dokumenta. Programer u takvom slučaju prvo mora preuzeti novu glavnu kopiju, ručno otkloniti konflikte koje prouzrokuju njegove promjene te nakon toga obaviti integraciju.

Nakon nekog vremena izvorna i radna kopija mogu postati toliko različite da je vrijeme potrebno za njihovo spajanje duže od vremena koje je uloženo za implementaciju promjena. Ovaj se problem tada naziva *pakao integracije*. Iako se navedena situacija čini teško mogućom, timovi mogu biti veliki, pritisak može biti visok i tempo naporan. Bez specificiranja postupka verzioniranja te automatizacije izgradnje i provjere ispravnosti projekti lako mogu završiti upravo u navedenom stanju.



Slika 2.1: Faze kontinuirane integracije

Danas je kontinuirana integracija standardna praksa u razvoju programske potpore. Međutim, ona se značajno razlikuje od prakse koju je 1991. godine predložio Grady Booch. Danas se uz kontinuiranu integraciju usko vežu procesi automatizacije izgradnje i testiranja programske potpore. Ovi su pojmovi postali toliko standardan dio kontinuirane integracije da mnogi upravo njih nazivaju kontinuiranom integracijom. Drugim riječima, pojam kontinuirane integracije danas podrazumijeva barem neku razinu automatizacije procesa izgradnje i testiranja. S druge strane, učestalom spajanju radnih kopija daje se malo pozornosti.

Kontinuiranu integraciju moguće je podijeliti na tri generalne faze: izgradnju, testiranje i osiguranje kvalitete. Nadalje, fazu izgradnje moguće je podijeliti na podfaze verzioniranja, pripreme sustava, pripreme ovisnosti i izgradnje projekta. Faze testiranja i osiguranja kvalitete uvelike ovise o tipu programske potpore koja se razvija. U sklopu ovog rada u fazi testiranja provodim unit i UI testove, dok u sklopu faze osiguranja kvalitete provodim provjeru pokrivenosti koda testovima te provjeru izvornog koda korištenjem alata Swiftlint. Podjela kontinuirane integracije na faze s podfazama prikazana je na slici 2.1.

Svaka je faza obrađena zasebnim odlomkom u nastavku poglavlja. Proces verzioniranja u praksi dio je procesa izgradnje, zbog čega je u sklopu odlomka 2.2 razmotren i problem učestalosti obavljanja integracije.

Za automatizaciju izgradnje koristim alat Xcode Server. Alat je spoj dvije aplikacije, Xcodea i macOS Servera, te implementira veliki broj funkcionalnosti korištenih u sklopu kontinuirane integracije. Alat se pokazao najboljim među nekoliko sličnih

ispitanih alata. Razlog odabira Xcode Server alata detaljnije je prikazan u dodatku A.

2.1. Priprema

Jedan od ciljeva implementacije kontinuirane integracije je i olakšanje cjelokupnog procesa izgradnje, testiranja i osiguranja kvalitete. Što je implementacija automatizacije zahtjevnija, to se više narušava navedeni cilj. Zbog navedenog nastojim automatizirati instalaciju i konfiguraciju što većeg broja alata i tako olakšati cjelokupan proces implementacije kontinuirane integracije. Također, automatizacija procesa instalacije i konfiguracije pruža veću kontrolu nad samim procesom. Na primjer, alat je moguće instalirati samo kad je potreban i ako je uopće potreban, dok se ručnim postupkom instalacija alata mora obaviti prije pokretanja automatizacije.

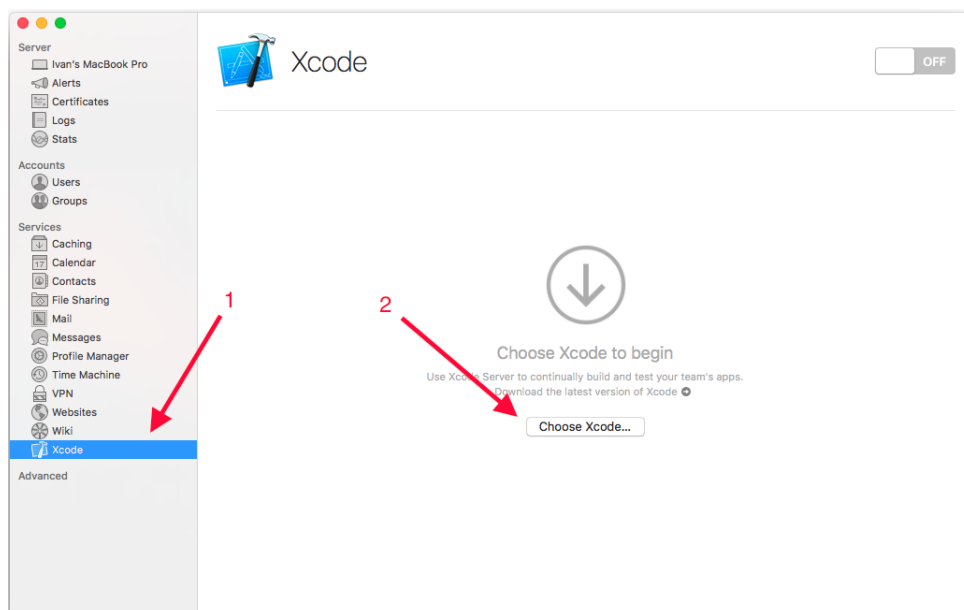
MacOS je vrlo siguran, a zbog toga i zatvoren operacijski sustav. Veliki broj alata zahtijeva korisničku interakciju zbog čega je njihovu instalaciju teško automatizirati. Dodatno, ako se instalacija obavlja za više računala operacijskog sustava ili alat pristupa osjetljivim datotekama, onda je instalaciju potrebno autorizirati lozinkom računa s administracijskim privilegijama. Postoji nekoliko načina za automatski upis lozinke, ali navedeni procesi narušavaju sigurnost operacijskog sustava te zbog toga nije moguće u potpunosti automatizirati proces dohvata i pripreme alata.

Ovaj odlomak prikazuje pripremu alata čiju instalaciju nije moguće automatizirati, dok odlomak 2.2.2 prikazuje pripremu ostalih alata. Primjeri su napisani za macOS operacijski sustav te su testirani na *Sierra 10.12.4* verziji. Minimalna preporučena verzija operacijskog sustava je *Yosemite 10.10*.

Za implementaciju kontinuirane integracije koristim brojne alate koji ne pružaju vizualno korisničko sučelje. Navedenim se alatima pristupa korištenjem naredbenog korisničkog sučelja - *ljuske*. U sklopu rada koristim *bash* ljusku. Pristup naredbenom korisničkom sučelju ostvaruje se korištenjem emulatora terminala - aplikacija s vizualnim sučeljem koje emuliraju terminal. U radu koristim aplikaciju *Terminal* koja je dostupna u sklopu instalacije macOS operacijskog sustava.

2.1.1. Xcode Server

Xcode Server spoj je dvije aplikacije - Xcodea i macOS Servera. Xcode je integrirani sustav za razvoj programske potpore za iOS, macOS, tvOS i watchOS operacijske sustave. MacOS Server je alat za automatizaciju procesa na macOS operacijskom sustavu. Prije implementacije kontinuirane integracije potrebno je preuzeti, instalirati



Slika 2.2: Povezivanje macOS Server i Xcode aplikacija

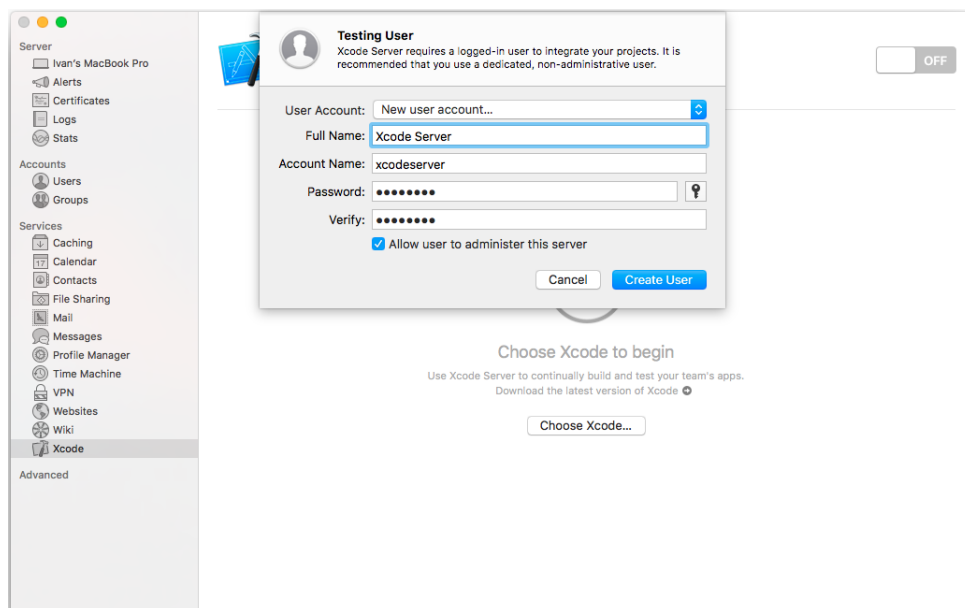
i konfigurirati obje aplikacije.

Moguće je oba alata preuzeti korištenjem App Store aplikacije koja je dostupna u sklopu svake instalacije macOS operacijskog sustava. Međutim, cijena macOS Server aplikacije u trenutku pisanja ovog rada iznosi \$25. Aplikacija je besplatna za korisnike s Apple Developer računom koji se koristi i u razvoju iOS programske potpore. Besplatnu verziju macOS Servera moguće je preuzeti na poveznici <https://developer.apple.com/download/>. Nakon preuzimanja potrebno je slijediti upute za instalaciju obje aplikacije.

Nakon instalacije alata potrebno je kreirati Xcode Server alat povezivanjem Xcode i macOS Server aplikacija. Treba pokrenuti macOS Server i u lijevom bočnom izborniku odabrati opciju Xcode → Choose Xcode... U novootvorenom izborniku potom je potrebno odabrati željenu verziju Xcode aplikacije. Slika 2.2 prikazuje proces povezivanja macOS Server i Xcode aplikacija.

Preporučeno je zbog sigurnosnih razloga macOS Server pokrenuti na zasebnom računu operacijskog sustava te mu omogućiti korištenje samo potrebnih alata i datoteka. Nakon povezivanja Xcode aplikacije s macOS Serverom, otvara se izbornik u kojem je moguće kreirati novi račun operacijskog sustava ili odabrati postojeći. U sklopu ovog rada kreiram novi račun predodređenog imena *xcodeserver*. Potrebno je slijediti upute nakon kreiranja računa za dovršetak spajanja aplikacija.

Proces potpisivanja koda osigurava autentičnost i neizmjenjenost kreirane programske potpore. Proces je detaljnije objašnjen u 3.1 odlomku. Za sada je dovoljno znati da

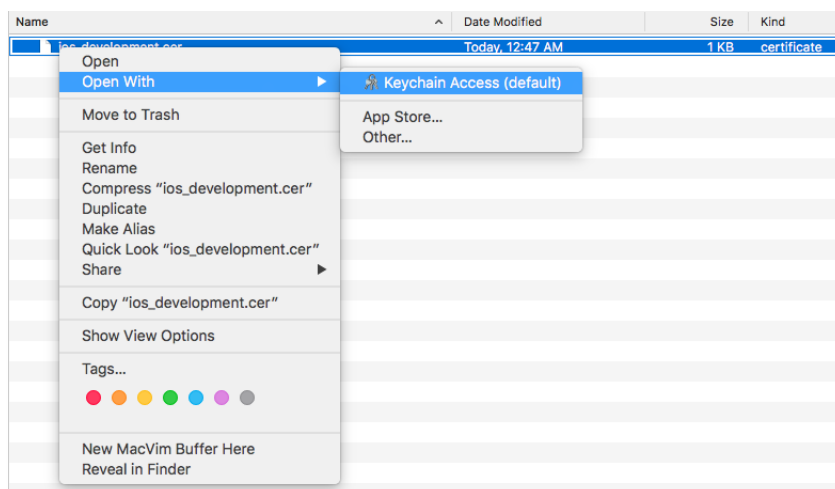


Slika 2.3: Kreiranje xcodeserver korisničkog računa

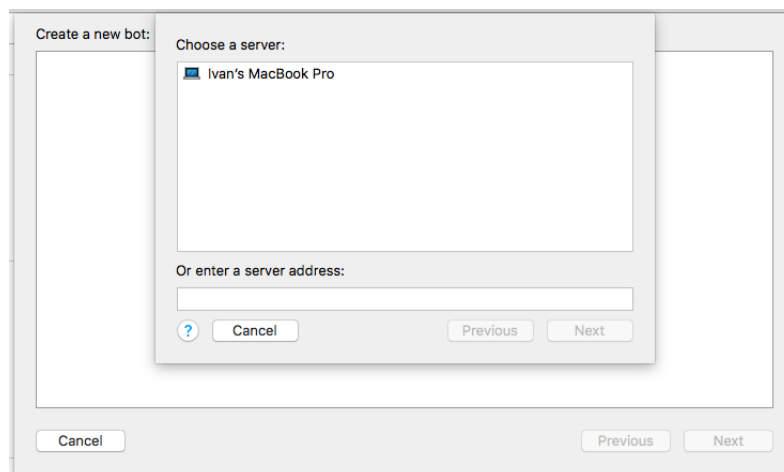
proces zahtijeva postojanje tri artefakta: certifikat člana tima, identifikator aplikacije i pripremni profil aplikacije za tip isporuke koji se koristi. Navedene artefakte moguće je kreirati i preuzeti s web-stranice <https://developer.apple.com/account>.

Certifikat i identifikator instaliraju se korištenjem Keychain Access aplikacije. Dovoljno ih je pokrenuti korištenjem navedene aplikacije. Pripremne profile potrebno je spremiti na lokaciji `~/Library/MobileDevice/Provisioning_Profiles` računa koji obavlja integraciju.

Xcode Server automatizaciju izgradnje, testiranja i isporuke ostvaruje korištenjem alata imena *bot*. Bot se kreira i konfigurira korištenjem Xcode aplikacije, a pokreće na



Slika 2.4: Dodavanje certifikata korištenjem Keychain Access aplikacije



Slika 2.5: Odabir macOS Server aplikacije za obavljanje kontinuirane integracije

macOS Server aplikaciji. Navedene aplikacije ne moraju se nalaziti na istom računalu, ali moraju biti povezane.

Treba pokrenuti željeni projekt korištenjem aplikacije Xcode. Prozor za kreiranje bota pokreće se odabirom opcije *Product -> Create bot*. U novootvorenom prozoru imenovati bot te odabrati macOS Server aplikaciju na kojoj će se bot izvršavati. Ako macOS Server nije vidljiv, potrebno je ponovno pokrenuti macOS Server aplikaciju i provjeriti povezanost s Xcode aplikacijom. Slika 2.5 prikazuje odabir macOS Server aplikacije kod kreiranja bota.

2.1.2. Homebrew

Homebrew je alat za dohvat i upravljanje alatima za macOS operacijski sustav[13]. Također, alat navedene funkcionalnosti pruža korištenjem naredbenog korisničkog sučelja zbog čega ih je jednostavno automatizirati. Instalacija alata zahtijeva administrativna prava (engl. *sudo user*), zbog čega istu nije moguće automatizirati. Skripta 2.1 instalira Homebrew alat korištenjem *ruby* alata. Nakon pokretanja naredbe slijediti upute instalacije.

Skripta 2.1: Instalacija Homebrew alata

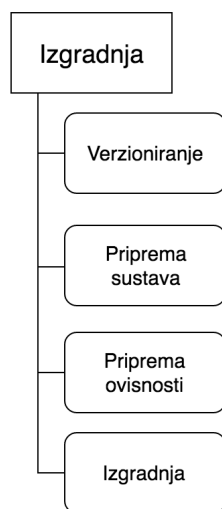
```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

2.2. Izgradnja

Povijesno, pojam izgradnja često se koristio kao sinonim za kompajliranje. Kompajliranje (engl. *compilation*) je proces prevođenja koda iz izvornog u ciljni jezik uz očuvanje funkcionalnosti. Kod se uz prevođenje često i optimizira. Najčešći razlog kompajliranja jest prevođenje koda u jezik koji procesor može razumjeti i tako ga izvršiti. Rezultat ovog tipa kompajliranja jest izvršni program, odnosno program koji se može izvršiti. Kompajliranje je složena funkcija koja se najčešće obavlja u više prolaza. Jezici koji se kompajliraju nazivaju se kompajlirani jezici (engl. *compiled languages*).

Interpretirani jezici (engl. *interpreted languages*) ne prevode se, već interpretiraju. Oni se izvršavaju na pomoćnom programu naziva interpreter koji naredbe izvornog jezika prevodi i izvršava. Danas gotovo niti jedan jezik nije u cijelosti kompajliran ili interpretiran, već koristi kombinaciju obje metode s ciljem poboljšanja performansi.

Danas se s pojmom izgradnje vežu svi procesi koji su dio pretvaranja izvornog koda u željeni artefakt. Ovisno o jeziku i alatima koji se koriste, proces izgradnje može značajno oscilirati u svojoj veličini i složenosti. Generalno, proces izgradnje možemo podijeliti na verzioniranje, pripremu sustava za izgradnju, dohvat i pripremu ovisnosti (engl. *dependencies*) te kompajliranje. Verzioniranjem odabiremo željenu verziju izvornog koda koju koristimo za izgradnju artefakta. Priprema za izgradnju dovodi računalu u stanje potrebno za obavljanje izgradnje. Izvorni kod često sadržava upute za pripremu sustava kao što su potrebni alati i postavke projekta. Dohvat i priprema ovisnosti osiguravaju postojanje ovisnosti koje zahtijeva izvorni kod. Ovisnosti



Slika 2.6: Podfaze procesa izgradnje

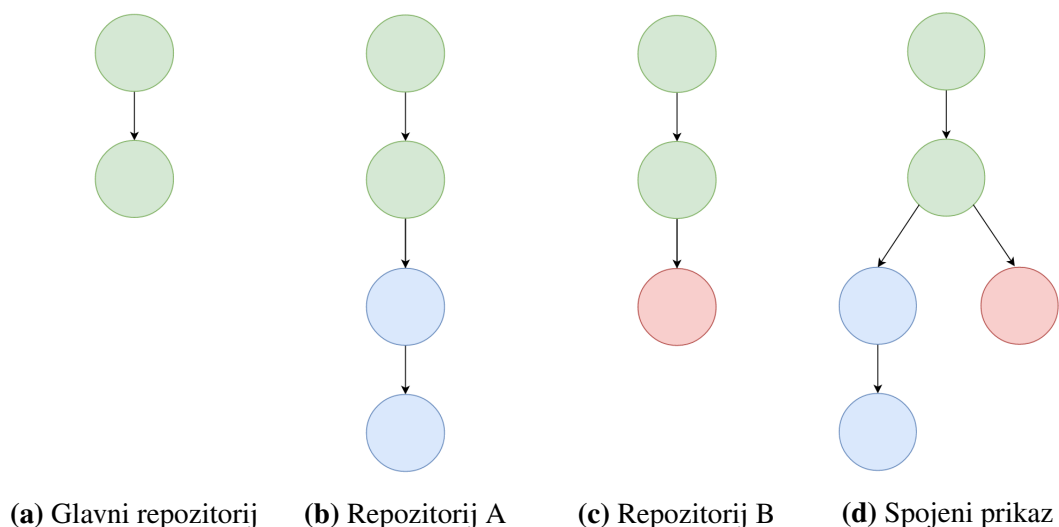
dijelimo na dva tipa: ovisnosti koje su dio razvojne okoline i vanjske (engl. *third party*) ovisnosti, koje uglavnom razvija zajednica. Kompajliranje prevodi izvorni kod u izvršivi artefakt. Kod interpretiranih jezika ovaj je proces često zamijenjen statičkom i dinamičkom provjerom izvedivosti programa. Zbog jednostavnijeg sporazumijevanja, oba procesa nazivam izgradnja projekta. Slika 2.6 prikazuje podjelu procesa izgradnje.

Osim kreiranja artefakta, izgradnja provjerava i je li verzija izvornog koda izgradiva. Kod je izgradiv ako u procesu izgradnje ne izaziva pogrešku, odnosno ako se kod ispravno izgradi. Pogrešku može izazvati neispravnost u izvornom kodu, neispravna konfiguracija sustava, nepostojanje potrebnog alata ili neki drugi nedostatak. Izgradivost sustava preduvjet je za testiranja i isporuku. Samim time automatizacija izgradnje preduvjet je za automatizaciju testiranja i automatizaciju isporuke.

2.2.1. Verzioniranje

Verzioniranje je proces dodjele jedinstvene oznake (engl. *id*) stanju repozitorija. Repozitorij je verzioniran direktorij i može sadržavati sve od izvornog koda do certifikata i izvršnog programa. Jedinstvena oznaka omogućava identifikaciju pojedinog stanja repozitorija i izgradnju stabla promjena (engl. *source tree*) koje povezivanjem stanja prikazuje povijest izmjena repozitorija. Verzionirano stanje repozitorija naziva se verzija (engl. *commit*)[20].

Uz jedinstvenu oznaku i stanje repozitorija, proces verzioniranja pohranjuje i dodatne podatke kao što su autor i datum kreiranja verzije te identifikator prijašnje verzije. Navedeni podaci omogućavaju izgradnju stabla promjena. Dodatno, ako se ver-



Slika 2.7: Stablo promjena

zije poredaju kronološki po datumu obavljanja izmjena, onda svaka pojedina verzija ne treba sadržavati cijelo stanje repozitorija. Dovoljno je samo navesti promjene obavljene nakon prijašnje verzije. Navedeni proces ne samo da značajno smanjuje veličinu cijele kopije, već olakšava i praćenje izmjena. Slika 2.7 prikazuje primjer stabla promjena. Glavni repozitorij sadrži dvije verzije. Repozitorij u navedenom stanju preuzimaju dva člana tima čime kreiraju lokalne repozitorije nad kojima obavljaju izmjene. Globalno stablo promjena kreira se zajedničkim prikazom stabla promjena svih članova.

Navedeni se tip verzioniranja naziva inkrementalno verzioniranje, jer se zbog lakšeg praćenja promjena provodi vrlo često. Identifikatori ovog tipa verzioniranja najčešće su generirani pseudo-slučajno. U praksi se često koriste i dodatne sheme verzioniranja koje olakšavaju praćenje stanja projekta. Navedene sheme nastoje olakšati praćenje projekta zbog čega se ovaj tip verzioniranja naziva vanjsko verzioniranje. Nove se verzije kreiraju dodavanjem posebnih oznaka postojećoj verziji inkrementalnog verzioniranja. Na primjer, u praksi je standardno označiti svaku verziju iz koje se kreira produkt posebnom oznakom koja se naziva verzija izgradnje (engl. *build number*). Kako vanjsko verzioniranje nosi neko značenje, proces dodjele identifikatora puno je složeniji i ovisi o svrsi koje se pokušava postići.

Unutarnje verzioniranje koda naziva se kontrola verzija[21]. Sustavi koji implementiraju proces kontrole verzija nazivaju se sustavi za kontrolu verzija. Kroz povijest je razvijen veliki broj sustava za kontrolu verzija, a danas je timski razvoj programske potpore gotovo nezamisliv bez korištenja jednog od njih.

Danas su u praksi najpopularnija dva alata: Apach Subversion i git.

Apache Subversion, poznat i pod skraćenicom svn, kreiran je 2000. godine u sklopu projekta koji je vodila *Apache Software Foundation* zajednica. Alat je centraliziran, siguran i jednostavan za korištenje te je danas objavljen kao alat otvorenog koda. Generalno, postoji jedan glavni repozitorij koji članovi tima kloniraju, uređuju te zatim lokalne promjene sinkroniziraju s njim.

Git je kreirao Linus Torvalds 2005. godine zbog nezadovoljstva tadašnjim sustavima za kontrolu verzija. Git je izdan kao alat otvorenog koda te je ubrzo okupio veliku podršku u zajednici. Za razliku od svn-a, git je distribuirani sustav. Repozitoriji istog projekta mogu postojati na proizvoljnom broju uređaja u proizvoljnom broju stanja. Navedeni se repozitoriji mogu klonirati, usklađivati i uređivati neovisno jedan o drugom. Zbog navedenog, pomoću gita moguće je implementirati proizvoljan pristup verzioniranju, bio to centralizirani repozitorij nalik na svn-ov pristup, pristup s osobama zaduženim za odobravanje promjena, distribuirani model ili drugo. Najvaž-

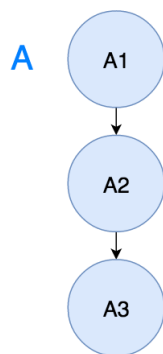
nije, git je jednostavan ali vrlo moćan alat. Implementacija osnovnih funkcionalnosti intuitivna je dok istovremeno postoji podrška za vrlo kompleksne pothvate.

Svn je stariji, međutim još uvijek široko korišten sustav. Koristi ga veliki broj starijih kompanija i projekata otvorenog koda. Git je značajno popularniji na novijim projektima, posebno onim otvorenog koda. Njegova jednostavnost i fleksibilnost čine ga lakšim za upoznavanje i korištenje. Zbog toga u ovom radu koristim git. Sve se funkcionalnosti mogu, uz manju modifikaciju, implementirati i korištenjem svn-a.

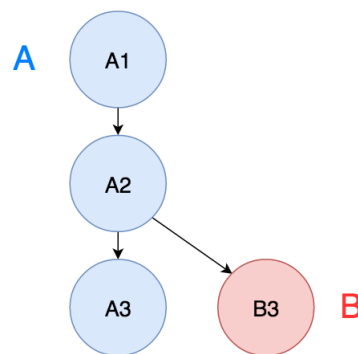
Uvod u git Temeljne funkcionalnosti git alata su repozitoriji i grane. Repozitorij je direktorij koji je verzioniran korištenjem git sustava za kontrolu verzija. Ovaj repozitorij sadrži direktorij `.git` koji specificira način na koji se verzionira direktorij te sadrži informacije o repozitoriju.

Repozitoriji se mogu klonirati na istom ili drugom uređaju. Klonirani repozitorij je novi repozitorij identičan izvornom repozitoriju. Promjene koje se obavljaju u kloniranom repozitoriju nemaju nikakvog utjecaja na izvorni repozitorij. Međutim, promjene obavljene u kloniranom repozitoriju mogu se, uz postojanje odgovarajuće autorizacije, prenijeti na izvorni repozitorij. Prijenos promjena ne mora se obavljati isključivo između izvornog i kloniranog repozitorija, već se može obaviti između bilo koja dva povezana repozitorija.

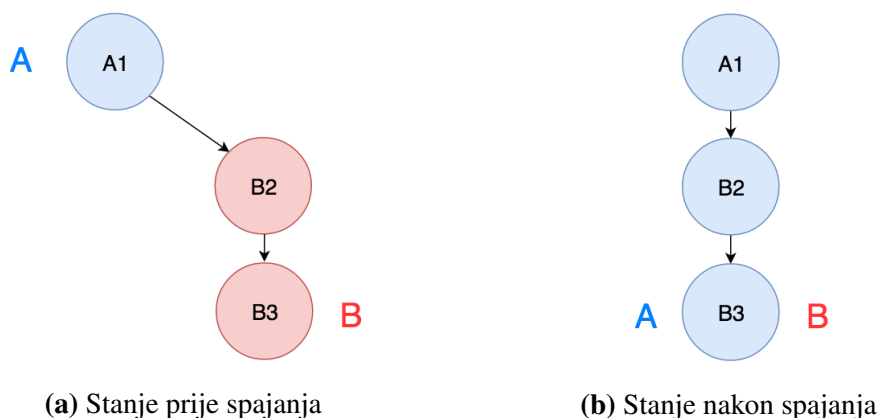
Prilikom kreiranja git repozitorija stvara se i glavna grana (engl. master branch) repozitorija. Grana je definirana slijedom verzija koje su obavljene na njoj. Stvaranje nove verzije na trenutnoj se grani ostvaruje potvrđivanjem promjena (engl. *commit*) koje su dodane repozitoriju. Potvrđivanje promjena prikazano je na slici 2.8. Repozitorij koda stvara se kreiranjem glavne grane i obavljanjem inicijalnog potvrđivanja (engl. *initial commit*). Glavna grana označena je slovom A. Inicijalno potvrđivanje označeno je identifikatorom A1, dok su naknadna potvrđivanja označena identifikatorima



Slika 2.8: Grana s tri potvrde



Slika 2.9: Grananje



Slika 2.10: Spajanje dodavanje promjena

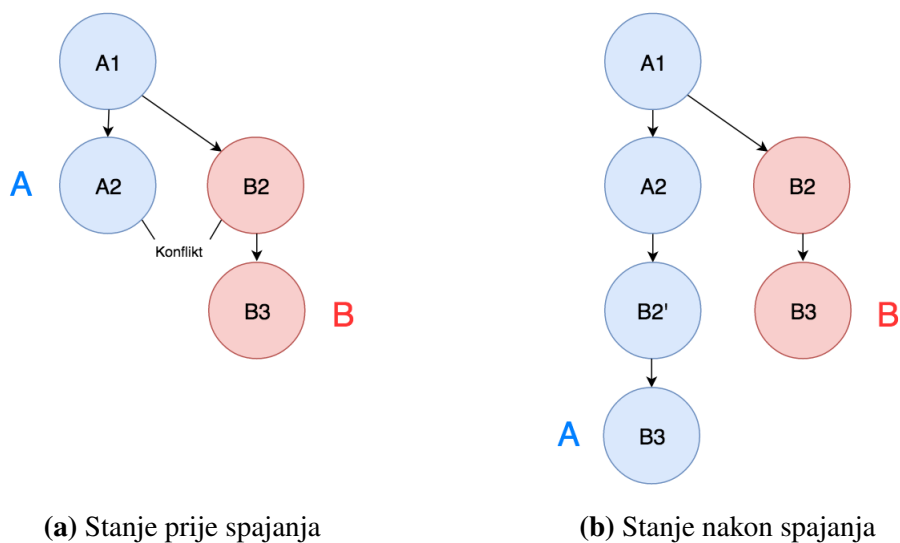
A2 i A3.

Nova se grana može kreirati iz bilo kojeg stanja postojeće grane. Ovaj se postupak naziva grananje (engl. *branching*). Izvorna i kreirana grana dijele zajedničku povijest do trenutka grananja. Daljnje promjene primjenjuju se samo na jednu od postojećih grana. Slika 2.9 prikazuje postupak grananja. Grana B se kreira iz stanja A2 grane A. Grane A i B dijele dva zajednička stanja A1 i A2. Ova stanja nazivamo zajednička povijest grana A i B. Nakon grananja na granu B dodaje se novo stanje B3.

Grane je također moguće spojiti. Spajanje grana dodaje promjene obavljene na izvornoj (engl. *source*) grani u odredišnu (engl. *destination*) granu. Spajanje je moguće obaviti na nekoliko načina ovisno o odnosu dviju grana koje se spajaju. Slika 2.10 prikazuje najjednostavniji odnos dviju grana kod spajanja. Nakon grananja grane B iz stanja A1 grane A na granu B dodaju se dva nova stanja, B2 i B3. U međuvremenu je grana A ostala nepromijenjena. Zbog navedenog spajanje grana moguće je obaviti jednostavno dodavanjem promjena B grane na vrh A grane, (engl. *fast forward merge*). Slike 2.10a prikazuje stanje prije spajanja dok slika 2.10b prikazuje stanje nakon spajanja. Također, samo je spajanje moguće označiti dodavanjem novog stanja na odredišnu granu.

Postupak se komplicira ako je odredišna grana modificirana nakon grananja. U navedenom slučaju nije moguće promjene obavljene u izvorišnoj gani samo dodati na vrh odredišne grane, nego je promjene potrebno spojiti. Proces spajanja ovisi o tome postoje li konflikti između promjena. Ako ne postoji, spajanje je moguće obaviti jednako kao na slici 2.10, jednostavno dodavanjem promjena na vrh odredišne grane.

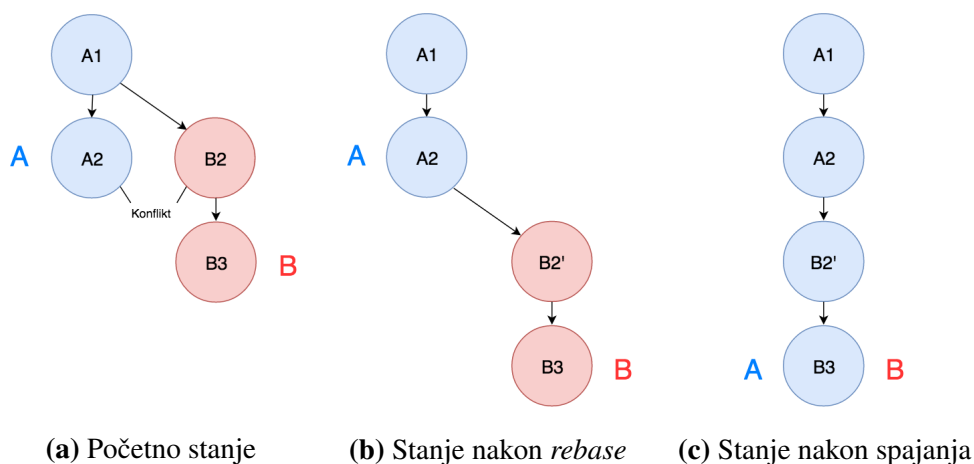
Međutim, ako promjene izazivaju konflikte, onda je te konflikte potrebno ručno razriješiti. Otklanjanje konflikata uzrokuje izmjenu verzija jedne ili obje grane. Proces otklanjanja konflikata najčešće se odrađuje dodavanjem jedne po jedne verzije izvorne



Slika 2.11: Spajanje otklananjem konflikta

grane na odredišnu granu. Ako izvorna verzija ne izaziva konflikt, ona se jednostavno dodaje na vrh odredišne grane. Međutim, ako verzija izaziva konflikt, tada se isti otklanja modificiranjem iste. Slika 2.11 prikazuje proces spajanja grana s konfliktom. Konflikt je nastao između verzija A2 i B2. Konflikt se otklanja dodavanjem verzije B2 na vrh A grane i njenim modificiranjem. Ovo je stanje označeno s B2'. Stanje B3 ne izaziva konflikt te se samo dodaje na vrh A grane. Rezultat spajanja su dvije grane A i B različitih povijesti.

Isti je slučaj moguće riješiti postupkom koji se naziva *rebase*. Postupak prije spajanja u povijest izvorišne grane dodaje sve verzije nastale u odredišnoj grani nakon grananja. Verzije se dodaju odmah nakon stare točke grananja čime se točka grananja



Slika 2.12: Spajanje *rebase* postupkom

pomiče na zadnju trenutnu verziju A grane. Slika 2.12b prikazuje stanje nakon obavljanja *rebase* postupka na grani B. Sada su grane u stanju jednakom onom na slici 2.10 te je spajanje moguće obaviti dodavanjem promjena na vrh odredišne grane. Stanje B2 još se uvijek mijenja, međutim, sada je povijest repozitorija linearna.

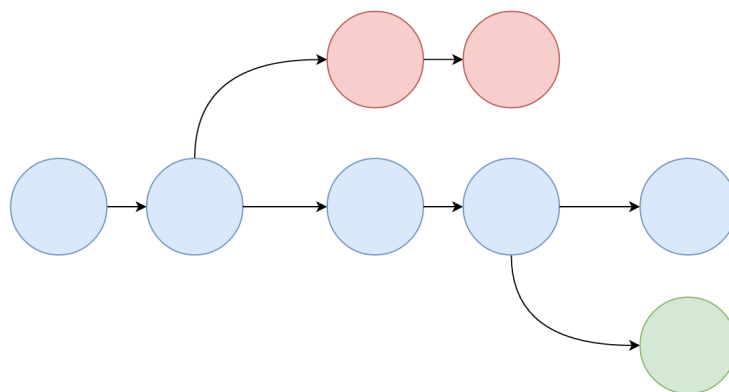
Tijek verzioniranja Ostaje otvoreno pitanje kako koristiti alat za kontrolu verzija. Koliko često kreirati novu verziju koda i koliko često promjene spajati sa zajedničkim repozitorijem? Kod korištenja gita javljaju se i pitanja kako organizirati repozitorije i sustav grananja.

Programer koji samostalno radi na projektu najčešće koristi jedan javni repozitorij s jednom granom na kojoj obavlja promjene i proizvoljno sinkronizira lokalni s glavnim repozitorijem. Međutim, ovaj je pristup vrlo teško održiv u timskom radu. Učestalo preplitanje različitih tokova razvoja na jednoj grani značajno otežava praćenje razvoja i čini teškim poništavanje neželjenih promjena.

Danas se u praksi koristi nekoliko različitih tijeka rada verzioniranja (engl. *versioning workflows*). Ovaj odlomak obrađuje centralizirani tijek rada (engl. *centralized workflow*), tijek rada grananja funkcionalnosti (engl. *feature branch workflow*), *gitflow* tijek rada (engl. *gitflow workflow*) i tijek rada izdvajanja promjena (engl. *forking workflow*). Svaki od navedenih pristupa ima svoje prednosti i mane te se koristi u različitim tipovima projekta[2].

Centralizirani tijek rada koristi jedan glavni i više lokalnih repozitorija. Najčešće se koristi samo jedna, glavna grana. Svaki programer kreira lokalnu kopiju glavnog repozitorija na kojoj obavlja promjene. Nakon obavljanja željenih promjena, spaja ih s glavnim granom centralnog repozitorija. Na pojedinom je programeru da vlastitu, lokalnu verziju repozitorija drži usklađenom s glavnim repozitorijem. Glavni repozitorij predstavlja službeno stanje projekta zbog čega treba posebnu pažnju obratiti na održavanje njegove povijesti. Izmjena povijesti glavnog repozitorija može dovesti lokalne repozitorije u nekonzistentno stanje, zbog čega se ona smatra lošom praksom. Zbog navedenog, ako lokalna kopija izaziva konflikt pri spajanju, konflikt je potrebno otkloniti na lokalnoj kopiji te promjene zatim spojiti s centralnim repozitorijem. Centralizirani proces vrlo je jednostavan te je sličan načinu rada svn-a.

Tijek rada grananja funkcionalnosti nastoji otkloniti glavni nedostatak centraliziranog tijeka rada - učestalo preplitanje različitih tokova razvoja. Grananje funkcionalnosti također ima jedan glavni i više lokalnih repozitorija. Razlika je u tome što se funkcionalnosti implementiraju u grani kreiranoj specifično za nju. Programer za novu funkcionalnost kreira novu granu u lokalnom repozitoriju te u nju dodaje promjene.



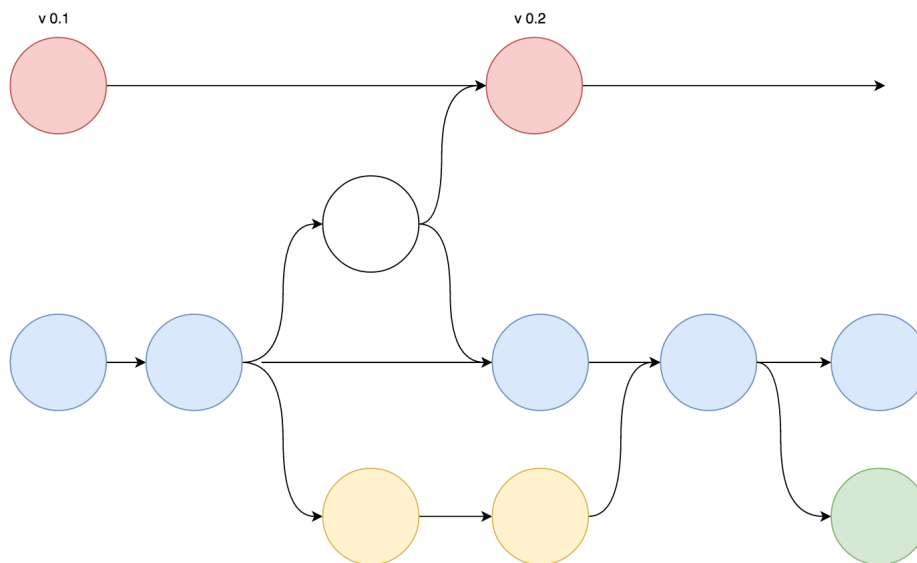
Slika 2.13: Primjer tijeka rada grananja funkcionalnosti

Pri završetku implementacije funkcionalnosti programer granu spaja s glavnom granom centralnog repozitorija. Ovaj proces daje jasniji uvid u napredak projekta i implementirane funkcionalnosti. Isto tako, ovaj proces timu daje priliku revizije obavljenih promjena. Umjesto direktnog spajanja grane moguće je kreirati zahtjev za spajanjem (engl. *merge request*). Zahtjev za spajanjem dodatno opisuje promjene ostvarene u sklopu grane te timu daje priliku za komunikaciju i reviziju obavljenih promjena.

Gitflow tijek rada također koristi jedan centralni i više lokalnih repozitorija. Za razliku od prijašnja dva tijeka rada, *gitflow* tijek rada povijest repozitorija prati kroz glavnu i razvojnu granu. Razvojna grana (engl. *develop branch*) vrlo je slična glavnoj grani u procesu grananja funkcionalnosti. Grana za novu funkcionalnost kreira se iz razvojne grane te se pri završetku implementacije u nju spaja. S druge strane, glavna grana sadrži samo produkcijske verzije izvornog koda, odnosno one verzije projekta koje su objavljene korisniku. Kad tim odluči objaviti novu verziju projekta, kreira se nova grana iz trenutnog stanja razvojne grane. Nakon završetka provjere ispravnosti grana se spaja s glavnom i, po potrebi, razvojnom granom. Nova se verzija glavne grane zatim objavljuje. Verzije na glavnoj grani označavaju se s objavljenom verzijom projekta.

Primjer korištenja *gitflow* procesa prikazan je na slici 2.14. Crvenom bojom prikazana je glavna grana, a plavom razvojna grana. Grane funkcionalnosti, prikazane zelenom i žutom bojom granaju se iz razvojne grane te u nju spajaju. Bijelom bojom označena je grana pripreme za objavu nove verzije projekta. Nakon obavljanja pripreme za objavu, grana se spaja s glavnom granom tima, kreirajući novu produkcijsku verziju, te s razvojnom granom kako bi promjene nastale pri pripremi za objavu bile dodane projektu.

Navedeni pristup olakšava upravljanje objave projekta. Buduće da je krucijalno



Slika 2.14: Primjer gitflow tijeka rada

objaviti ispravan produkt, sam proces objave treba biti kontroliran, a produkt temeljito testiran. Izdvajajući proces objave na pomoćnu granu omogućava istovremeno testiranje produkcijske verzije i nastavak rada na novim funkcionalnostima.

Za razliku od ostalih tijekova rada promatranih u ovom poglavlju, forking tijek rada nema centralni repozitorij, već svaki sudionik ima vlastiti javni i privatni repozitorij. Programer vlastiti javni repozitorij kreira kopiranjem drugog javnog repozitorija. Zatim iz vlastitog javnog repozitorija kreira vlastiti privatni repozitorij. Promjene obavlja na privatnom repozitoriju te ih proizvoljno spaja s javnim repozitorijem. Navedene promjene zatim može iskoristiti netko drugi kloniranjem repozitorija ili spajanjem promjena s postojećim repozitorijem. Ujedno, programer može predložiti dodavanje vlastitih promjena drugom repozitoriju. Navedeni se proces naziva zahtjev za povlačenjem promjena (engl. *pull request*).

Forking tijek rada najčešće se primjenjuje za projekte otvorenog koda. On omogućuje svakom članu zajednice kloniranje, modifikaciju i objavu promjena obavljenih na projektu. Dodatno, zahtjev za spajanje daje vrlo dobar uvid u obavljene promjene bez modifikacije izvornog repozitorija.

Osnova kontinuirane integracije je kontinuirano, odnosno učestalo spajanje radnih kopija s glavnom kopijom. Kad bi se vodio samo ovim principom, centralizirani repozitorij najbolje bi zadovoljavao postavljene zahtjeve. Međutim, centralizirani repozitorij u praksi se ne koristi ni za što osim najjednostavnijih projekata.

Iako drugi tijekovi rada rjeđe obavljaju integraciju radnih kopija, prednosti koje pružaju nadilaze navedeni nedostatak. Također, moguće je smanjiti vrijeme između

spajanja radnih kopija. Na primjer, *gitflow* tijekom rada spajanje radne kopije s glavnom kopijom obavlja pri završetku implementacije funkcionalnosti. Što je veća funkcionalnost koja se implementira, to će duže radna kopija ostati izdvojena. Zbog navedenog, posao je potrebno razdijeliti na manje dijelove. To se ne odražava pozitivno samo na proces kontinuirane integracije, već olakšava i praćenje projekta te je sastavni dio agilnog pristupa razvoja programske potpore. Prednosti koje pružaju napredniji pristupi verzioniranju su: lakše praćenje razvoja, zahtjevi za spajanjem i lakša objava projekta u produkciju te nadilaze nešto duže vrijeme izdvojenosti radnih kopija.

U praktičnom dijelu rada koristim *gitflow* tijekom rada. Ovaj tijekom rada najbolje odgovara zahtjevima i tipu projekta. Isto tako, *gitflow* tijekom rada omogućava jednostavniju implementaciju kontinuirane dostave i isporuke. Uz glavnu i radnu granu, repozitoriju ću po potrebi dodavati dodatne grane. Na primjer, isporuku verzija programske potpore za testiranje izdvojiti ću u zasebnu granu. Navedeni proces omogućava lako praćenje testnih verzija te olakšava implementaciju procesa isporuke testne verzije.

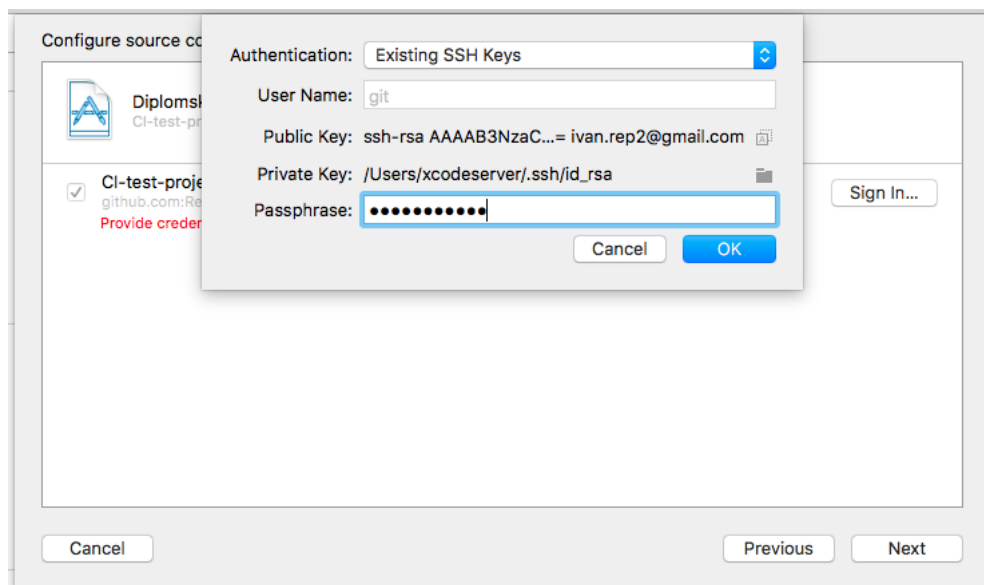
Verzioniranje u sklopu kontinuirane integracije Proces kontinuirane integracije započinje dohvaćanjem željene verzije repozitorija. Kako bih dohvaćao željenu verziju, proces kontinuirane integracije mora imati pristup repozitoriju. Xcode Server omogućava korištenje lokalnog ili udaljenog repozitorija verzioniranog *svn* ili *git* alatom. Ako se koristi udaljeni repozitorij, onda je isti potrebno zaštititi od neželjenog pristupa. Danas se u praksi koriste dva tipa zaštite: **HTTPS** i **SSH** autentifikacija.

HTTPS autentifikacija pristup kontrolira korištenjem jedinstvenog korisničkog imena i lozinke. Budući da se proces integracije odvija automatski, potrebno je spriječiti i automatizirati unošenje korisničkog imena i lozinke. Preporučeno je iste pohraniti korištenjem **Keychain Access** aplikacije.

SSH autentifikacija pristup kontrolira korištenjem javnog i privatnog ključa najčešće generiranog korištenjem **RSA** protokola. Ovaj je tip autentifikacije pogodniji za automatizaciju zbog čega ga koristim u sklopu ovog rada. Proces kreiranja i konfiguriranja **SSH** autentifikacije detaljnije je objašnjen u sljedećem odlomku.

Xcode Server automatski detektira alat kojim je repozitorij verzioniran. Nakon autentifikacije pristupa potrebno je odabrati granu za koju se kreira proces integracije. Pojedini bot integraciju obavlja za samo jednu granu pa je potrebno kreirati zaseban bot za svaku željenu granu. Slika 2.15 prikazuje autentifikaciju detektiranog *git* repozitorija korištenjem postojećeg **SSH** ključa.

Xcode Server omogućava automatsko pokretanje integracije nakon kreiranja nove verzije na promatranoj grani. Navedenu funkcionalnost Xcode Server ostvaruje konti-

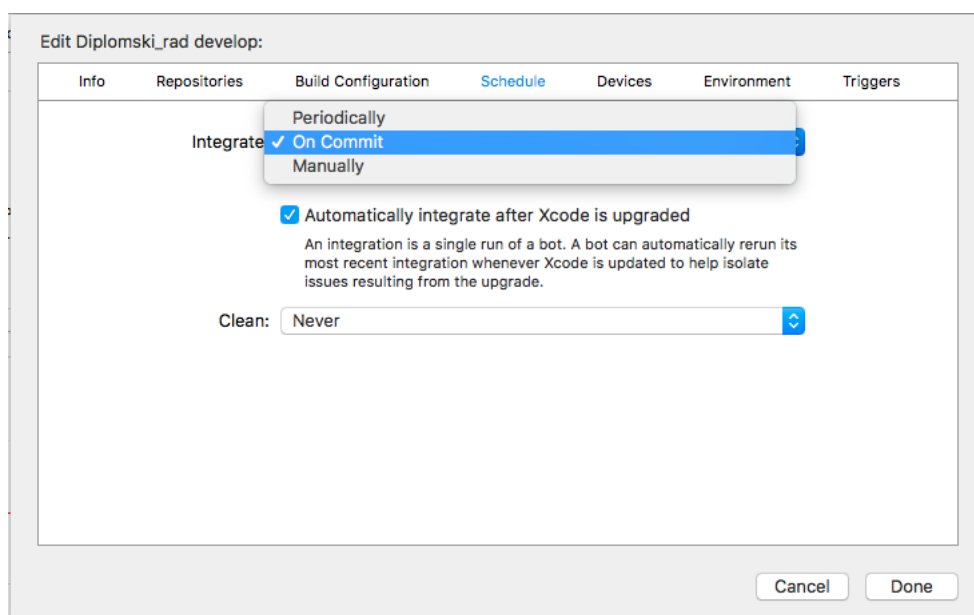


Slika 2.15: Dodavanje SSH ključa

nuiranom provjerom stanja repozitorija. Proces integracije također je moguće provoditi periodično ili ga pokretati ručno. Slika 2.16 prikazuje navedene opcije.

Korištenjem navedenih opcija kontinuirana integracija pokreće se nakon kreiranja novog stanja na odabranoj grani repozitorija.

SSH autentifikacija U sklopu rada koristim SSH autentifikaciju za pristup udaljenom git repozitoriju. SSH ključevi obično se pohranjuju u direktoriju `~/ .ssh`. Ako



Slika 2.16: Odabir načina pokretanja procesa integracije

SSH ključ već ne postoji u navedenom direktoriju, onda ga je potrebno kreirati. Skripta 2.2 prikazuje proces generiranja ključa. Naredba pod #1 generira novi SSH ključ sa željenom adresom e-pošte. Preporučeno je u tijeku kreiranja ključ zaštititi lozinkom.

Nakon generiranja, ključ je potrebno dodati SSH agentu kako se šifra ključa ne bi morala unositi pri svakom korištenju. Naredbe #2 i #3 ostvaruju navedenu funkcionalnost. Na kraju, javni dio ključa potrebno je registrirati na platformi koja *hosta* repozitorij. Naredba #4 kopira javni dio novokreiranog ključa.

Skripta 2.2: Postavljanje SSH autentifikacije

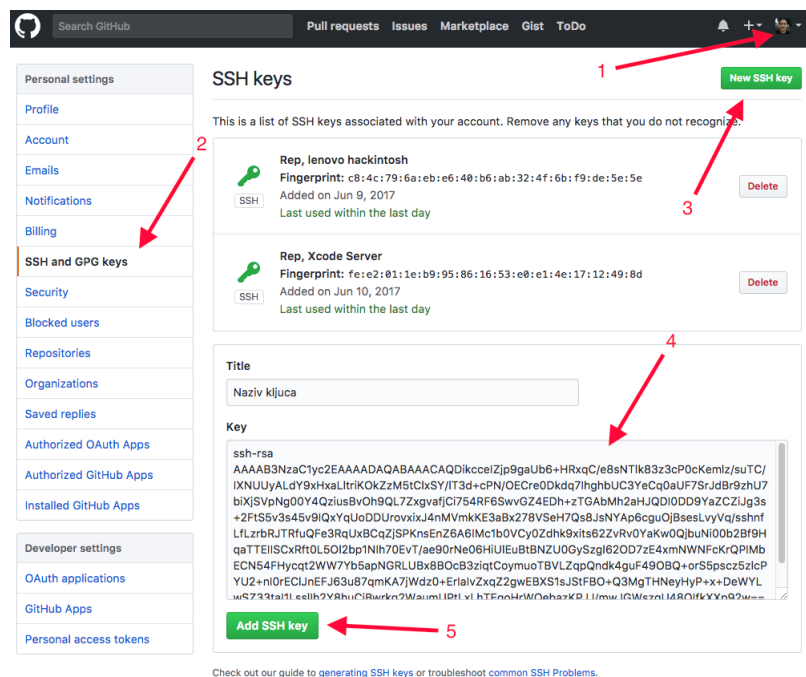
```
ssh-keygen -t rsa -b 4096 -C "{e-mail adresa}" #1
```

```
eval "$(ssh-agent -s)" #2
```

```
ssh-add -K ~/.ssh/{ime_ključa} #3
```

```
pbcopy < ~/.ssh/{ime_ključa}.pub #4
```

U sklopu rada koristim GitHub platformu. Slika 2.17 prikazuje proces registriranja SSH ključa na platformi. SSH ključ dodaje se odabirom opcije *Settings -> SSH and GPG keys -> New SSH key* te dodavanjem kopiranog javnog dijela ključa u polje za ključ. Nakon spremanja ključa isti je moguće koristiti za autorizaciju komunikacije s



Slika 2.17: Dodavanje SSH ključa na GitHub platformu

GitHub platformom.

2.2.2. Priprema sustava

Priprema sustava sastoji se od provjere postojanja, dohvata i konfiguracije potrebnih alata te od pripreme projekta za izgradnju.

Provjeru postojanja alata obavljam korištenjem skripte 2.3. Naredba #1 provjerava postojanje alata korištenjem alata `which` koji je dostupan u sklopu instalacije macOS operacijskog sustava. U slučaju nepostojanja alata, potrebno ga je dohvatiti i instalirati.

Skripta 2.3: Provjera postojanja alata

```
if !(which {ime_alata} >/dev/null); then #1
    {naredba za instalaciju alata} #2
fi
```

Na navedeni način provjeravam postojanje i instaliram tri alata, alate za dohvat ovisnosti CocoaPods i Carthage te alat za provjeru ispravnosti koda Swiftlint. Skripta 2.4 prikazuje automatiziranu instalaciju navedenih alata.

Skripta 2.4: Automatizirana instalacija alata

```
if !(which pod >/dev/null); then #1
    echo "Installing CocoaPods"

    gem install cocoapods --user-install
    pod repo update
fi

if !(which carthage >/dev/null); then #2
    echo "Instaling Carthage"

    brew install carthage
fi

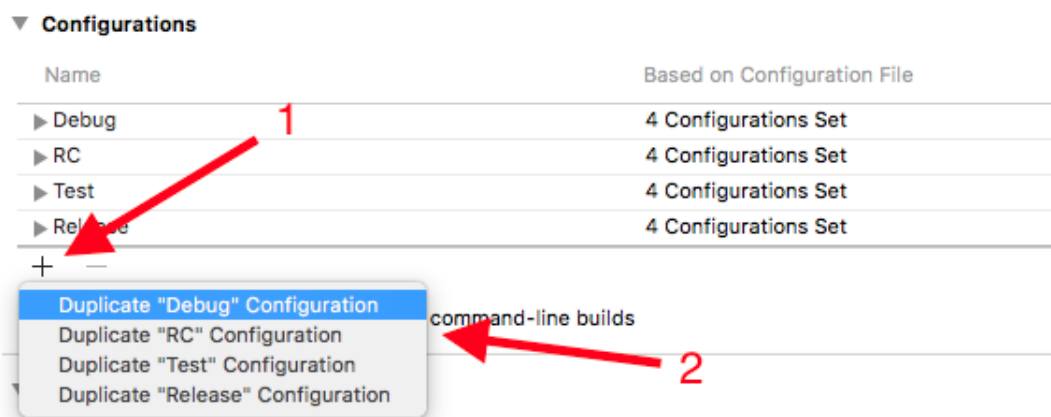
if !(which swiftlint >/dev/null); then #3
    echo "Instaling Swiftlint"

    brew install swiftlint
fi
```

Naredba pod #1 provjerava i instalira CocoaPods alat korištenjem `alta gem` dostupnog u sklopu instalacije macOS operacijskog sustava. Kako bih izbjegao unošenje administrativne lozinke, kod instalacije koristim opciju `--user-install` koja alat instalira samo za trenutnog korisnika. Naredbe #2 i #3 provjeravaju i instaliraju alate Carthage i Swiftlin korištenjem alata `brew`.

Konfiguracija Xcode projekta pohranjena je unutar `.xcodeproj` datoteke. Datoteka je namijenjena za modifikaciju i čitanje korištenjem alata Xcode. Datoteka nije pogodna za ručnu izmjenu i čitanje. Za olakšanje nadgledanja i modifikacije postavki projekta često se koriste `xcconfig` datoteke.

Navedene su datoteke tekstualnog formata s `.xcconfig` nastavkom te sadrže listu `{ključ} = {vrijednost}` linija koje specificiraju postavke projekta. Navedene je datoteke potrebno dodati projektu te iskoristiti za konfiguraciju željene sheme. Pomoću Xcode aplikacije potrebno je otvoriti projekt te odabrati željenu shemu i sekciju Info. U odjeljku Configurations odabirom opcije Plus potrebno je kreirati novu konfiguraciju. Slika 2.18 prikazuje proces dodavanja nove konfiguracije.



Slika 2.18: Kreiranje konfiguracije projekta

2.2.3. Upravljanje ovisnostima

Prije izgradnje projekta potrebno je dohvatiti ovisnosti koje projekt koristi. Za dohvaćanje ovisnosti u iOS razvoju koriste se dva alata: *CocoaPods* i *Carthage*. Oba se sustava široko koriste te izbor uvelike ovisi o osobnom ukusu. Zbog toga u radu koristim oba alata.

CocoaPods CocoaPods je stariji, široko prihvaćen, centraliziran alat za upravljanje ovisnostima iOS projekata. Alat je jednostavan i intuitivan za korištenje. Dovoljno je

specificirati ovisnosti korištenjem `Podfile` datoteke i pokrenuti proces dohvaćanja ovisnosti. Alat samostalno kreira i konfigurira radno okruženje te time olakšava proces upravljanja ovisnostima.

Međutim, najveći problem alata upravo je ova učestala modifikacija datoteka radnog okruženja. Alat pri svakom dohvat izmjenjuje postavke izgradnje što može uzrokovati neželjeno ponašanje. Dodatno, budući da je alat centraliziran, sve korištene biblioteke moraju biti registrirane u CocoaPods sustavu. Upravo to otežava korištenje privatnih biblioteka i biblioteka u razvoju.

Inicijalizacija CocoaPods alata prikazana je u skripti 2.5. Naredbu je potrebno pokrenuti u direktoriju projekta.

Skripta 2.5: Inicijalizacija CocoaPods alata

```
pod init
```

Naredba kreira `Podfile` datoteku koja služi za specifikaciju ovisnosti. Skripta 2.6 prikazuje primjer `Podfile` datoteke. Datoteka za cilj `Diplomski_rad` specificira dvije ovisnosti `UIKit` i `Fabric`.

Skripta 2.6: Primjer Podfile datoteke

```
use_frameworks!
```

```
target 'Diplomski_rad' do
  pod 'UIKit'
  pod 'Fabric'
end
```

Ovisnosti se dohvaćaju pokretanjem naredbe `pod install` u direktoriju projekta.

Skripta 2.7 automatizira dohvaćanje ovisnosti korištenjem CocoaPods alata. Naredba #1 provjerava postojanja `Podfile` datoteke te u slučaju njezina postojanja nastavlja s izvođenjem skripte. Naredba #2 provjerava postojanje CocoaPods alata. Ako alat nije instaliran, onda se instalira korištenjem `gem` alata. Na kraju, naredba #3 dohvaća ovisnosti korištenjem CocoaPods alata.

Skripta 2.7: Dohvat ovisnosti korištenjem alata CocoaPods

```
if [ -f Podfile ]; then #1
  echo "Podfile found. Starting CocoaPods"

  if ! which pod >/dev/null; then #2
```

```

        echo "Installing CocoaPods"

        gem install cocoapods --user-install
        pod repo update
    fi

    pod install #3

    echo "Finished dependency fetch using CocoaPods"
fi

```

Carthage Carthage je noviji, decentralizirani alat za upravljanje ovisnostima. Alat omogućava jednostavno dohvaćanje i izgradnju biblioteke bez potrebe njihove prijašnje registracije. Za razliku od CocoaPods alata, Carthage ne modificira radno okruženje. Ovisnosti je potrebno samostalno uključiti u projekt zbog čega je alat složeniji za korištenje od CocoaPods alata. Međutim, u istom trenutku alat otklanja neželjene posljedice koje nosi učestala izmjena datoteka razvojnog okruženja.

Potrebno je kreirati `Cartfile` datoteku. Datoteka je jednostavna lista ovisnosti zajedno s lokacijom izvornog repozitorija. Skripta 2.8 prikazuje primjer `Cartfile` datoteke.

Skripta 2.8: Primjer `Cartfile` datoteke

```

github "JohnSundell/Unbox"
git "https://gitlab.rep.com/Testni_projekt"

```

Primjer dohvaća dvije ovisnosti. Javno javnu ovisnost `Unbox` objavljenu na GitHub platformi te privatnu ovisnost objavljenu na Gitlab platformi.

Dohvaćanje se ovisnosti pokreće naredbom `carthage update`.

Skripta 2.9 implementira dohvaćanje ovisnosti korištenjem Carthage alata. Naredba #1 provjerava postojanje `Cartfile` datoteke te, u slučaju postojanja iste nastavlja obavljanje skripte. Naredba #2 provjerava postojanje Carthage alata te ga dohvaća ako ne postoji. Naredba #3 dohvaća ovisnosti korištenjem Carthage alata. Za bolje performanse operacije koristim dva argumenta. Argument `--platform ios` specificira dohvaćanje ovisnosti samo za iOS platformu. Argument `--cache-builds` dohvaća ovisnosti samo ako već nisu dostupne.

Skripta 2.9: Dohvat ovisnosti korištenjem alata Carthage

```

if [ -f Cartfile ]; then #1
    echo "Cartfile found. Starting Carthage"

    if !(which carthage >/dev/null); then #2
        echo "Instaling Carthage"

        brew install carthage
    fi

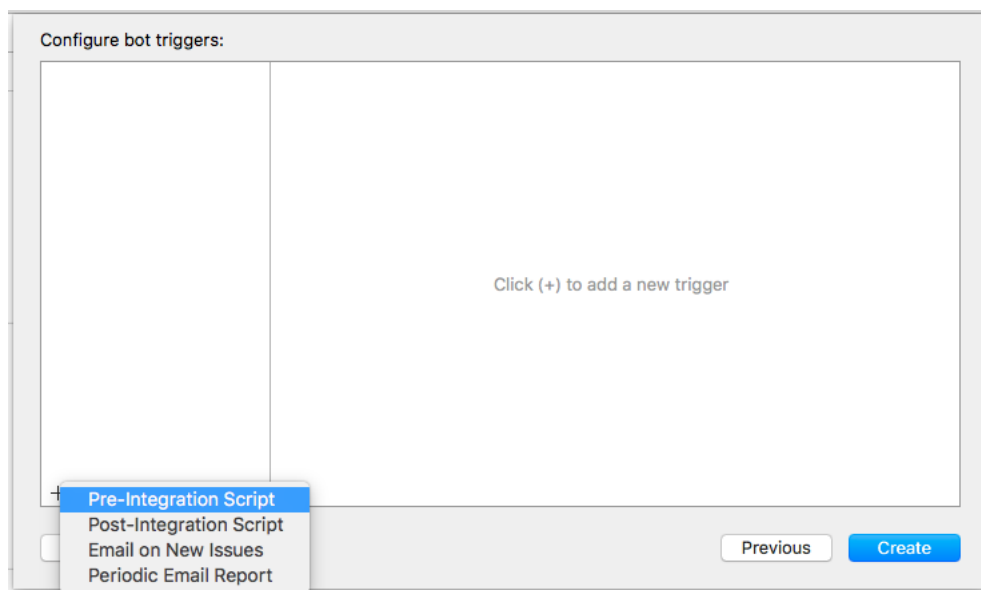
    carthage update --platform ios --cache-builds #3

    echo "Finished dependency fetch using Carthage"
fi

```

Automatizacija dohvaćanja ovisnosti Obje skripte potrebno je pokrenuti prije obavljanja integracije. Za ostvarenje navedene funkcionalnosti *botu* dodajem skriptu koja se izvršava prije pokretanja integracije (engl. *Pre-Integration Script*). Slika 2.19 prikazuje proces dodavanja nove skripte. Skripta se dodaje odabirom opcije *Edit Bot... -> Triggers -> Pre Integration Script*.

Unutar novokreirane skripte potrebno je pozvati sve naredbe koje se trebaju izvršiti prije integracije. Radi jednostavnosti i fleksibilnosti navedene je naredbe ko-



Slika 2.19: Dodavanje skripte koja se izvršava prije integracije

risno izdvojiti u zasebnu datoteku. Korištenjem ovog pristupa olakšavam implementaciju kontinuirane integracije te omogućavam laku izmjenu skripte koja se izvršava prije integracije. Skripta 2.10 se korištenjem `XCS_PRIMARY_REPO_DIR` varijable okruženja navigira u početni direktorij projekta te provjerava postojanje datoteke `scripts/preintegration`. U slučaju postojanja datoteke ista se izvršava.

Skripta 2.10: Poziv skripte koja se izvršava prije obavljanja integracije

```
#!/bin/bash

cd $XCS_PRIMARY_REPO_DIR

if [ -f scripts/preintegration ]; then
    ./scripts/preintegration
fi
```

Skripta 2.11 se izvršava prije integracija. Skripta jednostavno poziva prethodno definirane skripte. Na isti je način moguće dodati proizvoljan broj naredbi. Također, skripta u `PATH` varijablu okruženja dodaje dvije putanje koje olakšavaju korištenje postojećih alata.

Skripta 2.11: Skripta koja se izvršava prije integracije

```
#!/bin/bash

export PATH="/usr/local/bin:~/.gem/ruby/2.0.0/bin/:$PATH"

if [ -f scripts/cocoapods ]; then
    ./scripts/cocoapods
fi

if [ -f scripts/carthage ]; then
    ./scripts/carthage
fi
```

Sve se skripte trebaju nalaziti u `scripts` direktoriju repozitorija.

2.2.4. Izgradnja

Izgradnja iOS aplikacija obavlja se korištenjem alata `xcodebuild`[1]. Razvio ga je Apple za izgradnju programske potpore za macOS operacijski sustav. Alat je vrlo moćan

te pruža veliki broj funkcionalnosti i mogućih konfiguracija. S vremenom je i proširen te danas podržava izgradnju aplikacija za iOS, tvOS i watchOS operacijske sustave. Alat izgradnju obavlja korištenjem Xcode projekta. Prije definiranja procesa izgradnje potrebno je upoznati se sa strukturom Xcode projekta.

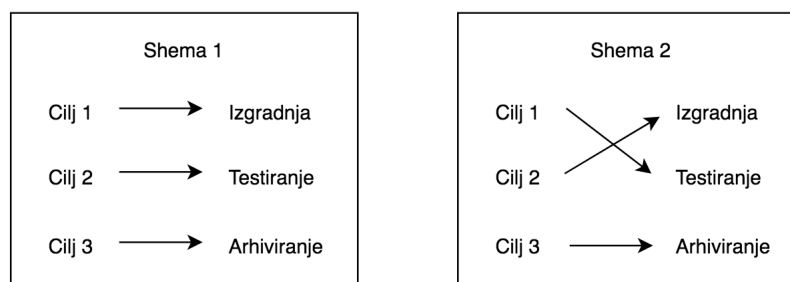
Xcode je službeni Appleov alat za razvoj programske potpore za iOS i macOS operacijske sustave. Na tržištu postoji nekoliko alternativa, ali Xcode daleko je najkorišteniji. Svi alati koriste alat `xcodebuild` za izgradnju te zbog toga imaju vrlo sličnu strukturu projekta. Ovaj tip projekta naziva se Xcode projekt.

Xcode projekt sadrži jedan ili više ciljeva (engl. *target*) i jednu ili više shema (engl. *scheme*). Cilj definira postavke koje se koriste kod izvršavanja operacije za navedeni cilj. Jedan projekt može sadržavati više ciljeva. Pomoću ciljeva moguće je isti kod distribuirati za različite verzije operacijskog sustava, različite operacijske sustave i testirati projekt. Shema definira koji se cilj koristi za koju operaciju. Projekt može koristiti više shema kako bi objedinio operacije za pojedinu distribuciju. Odnos cilja i sheme prikazan je na slici 2.20. Projekt sadrži tri cilja i dvije sheme. Sheme različito definiraju koji se cilj koristi za koju operaciju.

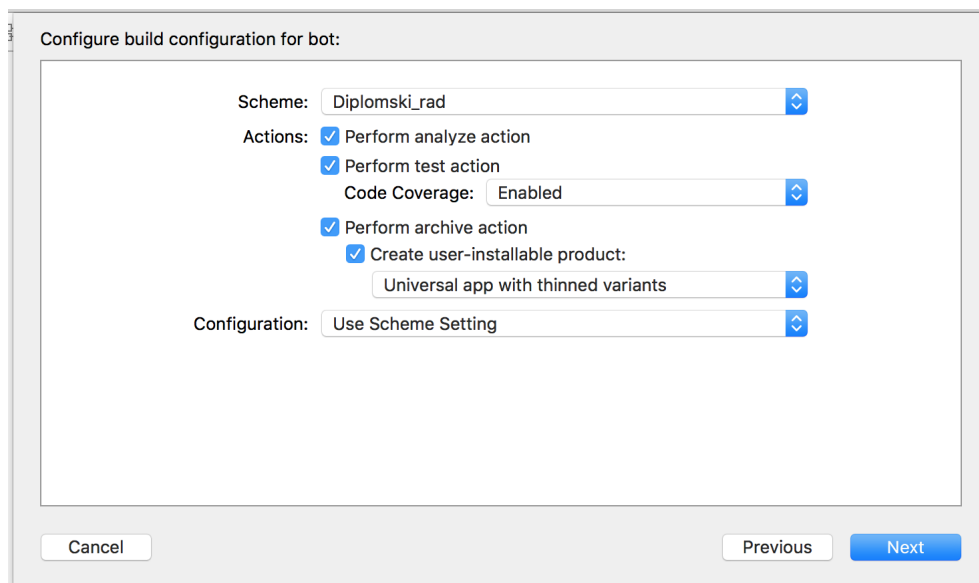
Xcode projekte moguće je grupirati u Xcode radno okruženje (engl. *workspace*). Radno okruženje olakšava segmentiranje velikog projekta i olakšava upravljanje ovisnostima.

Izgradnja iOS projekta u praksi se pokreće gotovo isključivo korištenjem alata Xcode. Međutim, navedeni pristup nije moguće automatizirati. Zbog navedenog, procesi koji automatiziraju izgradnju koriste alat `xcodebuild` direktno ili koriste alat koji interno koristi `xcodebuild`. Alat je detaljnije specificiran u dodatku B.

Xcode Server značajno olakšava korištenje alata `xcodebuild`. Nakon povezivanja bota s repozitorijem izvornog koda, moguće je konfigurirati opcije integracije. Moguće je i odabrati shemu projekta za koju se provodi integracija, operacije koje će se izvršavati u sklopu integracije te postavke koje se koriste za izgradnju projekta. Slika



Slika 2.20: Xcode projekt s tri cilja i dvije sheme

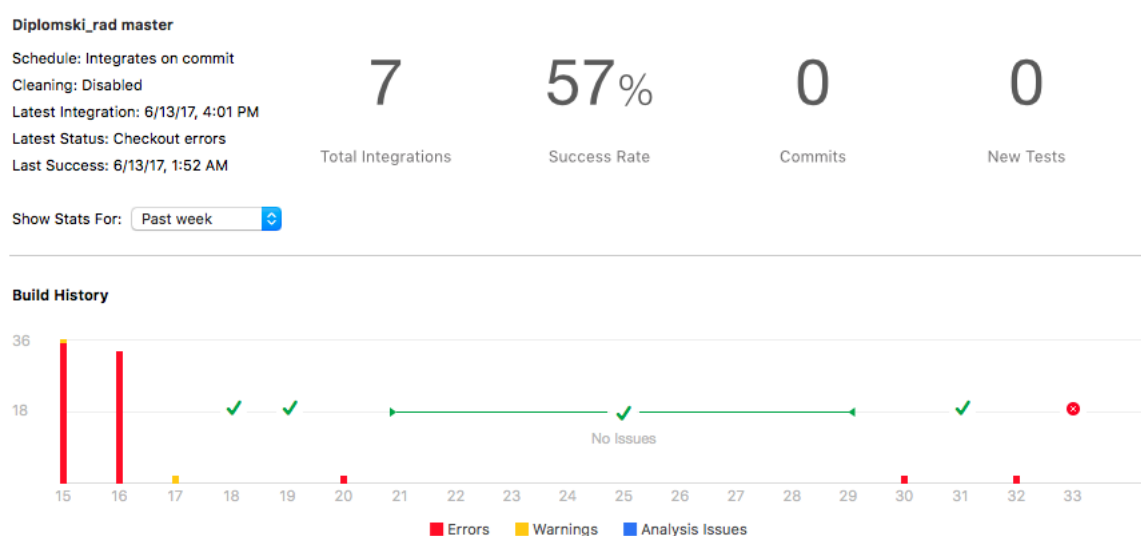


Slika 2.21: Konfiguracija osnovnih opcija integracije

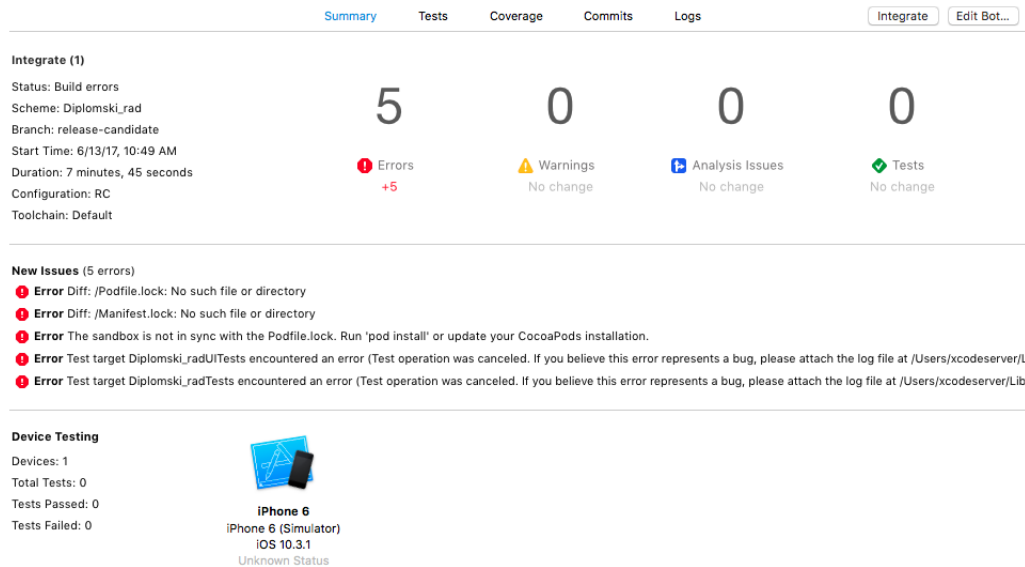
2.21 prikazuje opcije koje se mogu konfigurirati.

Kako bi se shema mogla iskoristiti za integraciju, ona mora biti javno objavljena. Shema se može konfigurirati korištenjem Xcode aplikacije.

Ovako konfiguriran *bot* provodi kontinuiranu integraciju za odabranu granu. Rezultat integracije moguće je vidjeti otvaranjem početnog ekranu *bota*. Slika 2.22 prikazuje primjer rezultata integracije. Prvi redak slike prikazuje opće informacije o kontinuiranoj integraciji kao što su broj obavljenih integracija, postotak uspješnosti te broj verzija repozitorija. Drugi redak slike prikazuje uspješnost izvođenja procesa izgradnje



Slika 2.22: Primjer prikaza rezultata kontinuirane integracije



Slika 2.23: Detaljan prikaz integracije

za svaku pojedinu integraciju.

Prve dvije prikazane integracije završile su greškom koja je u trećoj integraciji otklonjena. Četvrta integracije otklanja upozorenje dojavljeno u prve tri integracije. Integracija broj 20 dojavljuje novu grešku koju otklanja sljedeća integracija.

Odabirom pojedine integracije otvara se ekran s detaljima odabrane integracije. Detalji integracije podijeljeni su u nekoliko sekcija od kojih svaka pruža vrlo detaljne informacije o integraciji. Prva sekcija Summary prikazuje najvažnije podatke na jednom, sažetom ekranu. Slika 2.23 prikazuje primjer navedene sekcije za integraciju koja je dojavila pet novih pogrešaka u procesu izgradnje. Detalji novopronađenih pogrešaka prikazani su na ekranu što olakšava pronalaženje i otklanjanje pogrešaka.

Zadnja sekcija detaljnog prikaza integracije prikazuje ispis svih faza integracije, uključujući i ručno definirane faze. Sekcija je vrlo korisna za provjeru ponašanja pojedine faze.

2.3. Testiranje

Testiranje je sastavni dio razvoja programske potpore. Implementacijom kvalitetnih testova ne samo da se osigurava ispravan rad programske potpore, već se i sprječava nazadovanje koda (engl. *code regression*) te značajno smanjuje potreba za ručnim testiranjem ispravnosti[19].

Pogrešno je mišljenje da implementacija testova produljuje vrijeme razvoja. Svaku

implementiranu funkcionalnost i obavljenу izmjenu potrebno je testirati. Jedino je pitanje hoće li se navedeno testiranje obavljati automatski. Jednom napisan kvalitetan test može se pokrenuti proizvoljan broj puta. S druge strane, provođenje ručnog testiranja svaki put zahtijeva vrijeme članova tima. Bilo to u sklopu razvoja ili s ciljem provjere ispravnosti, ručna provjera ispravnosti zahtijeva više resursa i daje lošije rezultate.

Navedenu konstataciju ne treba zamijeniti s potpunim isključenjem ručnog testiranja aplikacije. Bez obzira na kvalitetu testova, pogreške se uvijek mogu dogoditi. Međutim, pisanjem kvalitetnih testova vjerojatnost pojave pogreške značajno se smanjuje.

Ovaj odlomak ne ulazi u proces pisanja testova, već samo automatizira njihovo pokretanje. Implementacija kvalitetnih testova vrlo je složeno područje te nadilazi okvire ovog rada.

Proces razvoja programske potpore za iOS operacijski sustav definira dva tipa testova: *unit* i *UI* testove.

Unit testovi su nesretno imenovani. Oni ne predstavljaju standardne *unit* testove, već se koriste kao ime za testove koji imaju pristup kodu koji testiraju. Testovi direktno komuniciraju s kodom koji testiraju i putem ove komunikacije provjeravaju ispravnost izvođenja. Ovaj tip testa pokreće se kao omotač oko izvorne aplikacije.

S druge strane, *UI* testovi nemaju pristup izvornom kodu aplikacije. Oni programsku potporu testiraju njezinim pokretanjem i simuliranjem korisničke interakcije. Programer specificira korisničke akcije i ponašanje koje očekuje od aplikacije nakon primanja navedne akcije. *UI* testovi pokreću dvije aplikacije: aplikaciju koju testiraju i aplikaciju koja simulira korisničku interakciju.

Oba tipa testova implementirani su kao testni ciljevi Xcode projekta. Kod testnog cilja je odvojen te se ne koristi u procesu izgradnje. Testni cilj referencira cilj koji testira. Dodatno, shema može specificirati koji se testni ciljevi pokreću prilikom pokretanja operacije testiranja. Na ovaj način jedna shema može u procesu testiranja pokrenuti više testnih ciljeva.

Proces testiranja može se pokrenuti na iOS Simulatoru, aplikaciji koja simulira iOS operacijski sustav na macOS operacijskom sustavu, ili na stvarnom uređaju. Simulator se instalira u sklopu instalacije Xcode aplikacije.

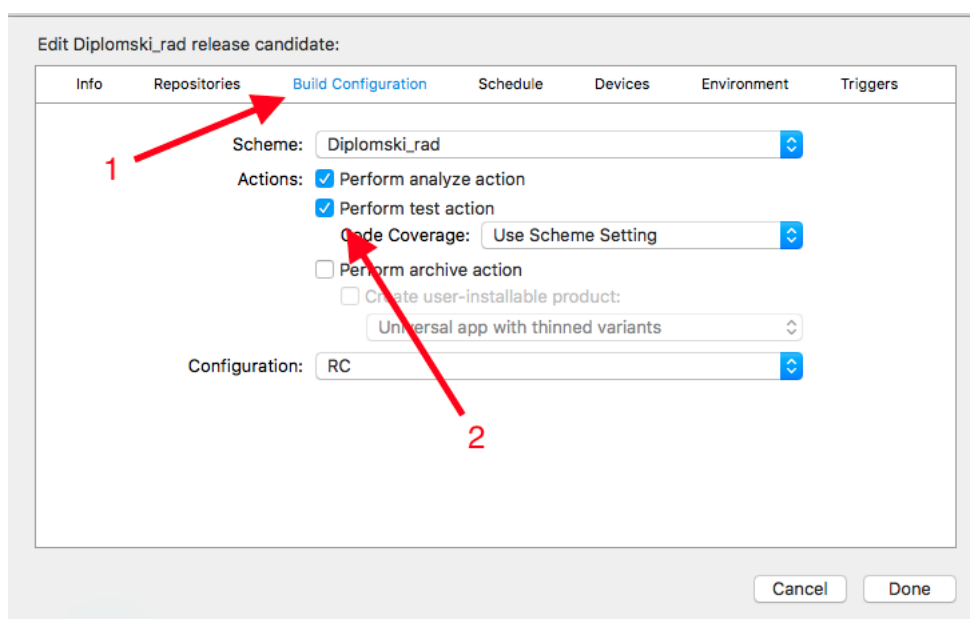
2.3.1. Automatizacija testiranja

Xcode Server detektira testne ciljeve sheme za koju obavlja proces integracije. Za pokretanje testova u sklopu integracije treba odabrati opciju `Edit bot...` te u sekciji `Build configuration` označiti opciju `Perform test action`. Slika 2.24 prikazuje navedeni proces. Sljedeća obavljena integracija pokrenut će sve testne ciljeve koje definira odabrana shema.

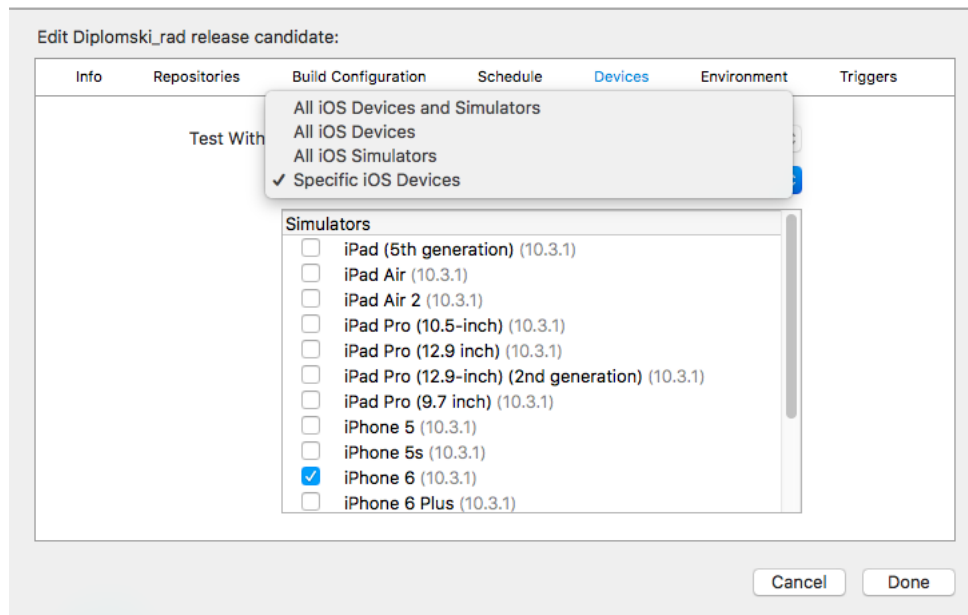
Isto tako, moguće je odabrati uređaje na kojima se pokreću testovi. Testove je moguće pokrenuti na svim dostupnim uređajima, samo na dostupnim iOS simulatorima, samo na dostupnim stvarnim uređajima ili na samo odabranim uređajima. Slika 2.25 prikazuje navedeni izbornik. Izborniku se pristupa odabirom opcije `Edit bot...` te otvaranjem sekcije `Devices`.

Novi se testni cilj shemi može dodati korištenjem Xcode aplikacije. Treba pokrenuti iOS projekt te odabrati željenu shemu iz padajućeg izbornika u gornjoj alatnoj traci aplikacije. Odabirom opcije `Edit scheme` otvara se prozor prikazan na slici. U lijevom izborniku treba odabrati opciju `Test` te pritiskom na plus ikonu dodati željeni testni cilj postojećoj shemi. Nakon spremanja promjene integracija u sljedećem se izvršavanju obavlja i novododani testni cilj.

Rezultati obavljanja testova također su prikazani na početnom ekranu *bota* u sekciji `Test History`. Osim broja uspješnih i neuspješnih testova, alat omogućuje i prikaz detaljnih rezultata testiranja. Odabirom rezultata testiranja pojedine integracije



Slika 2.24: Uključivanje pokretanja testova u sklopu integracije

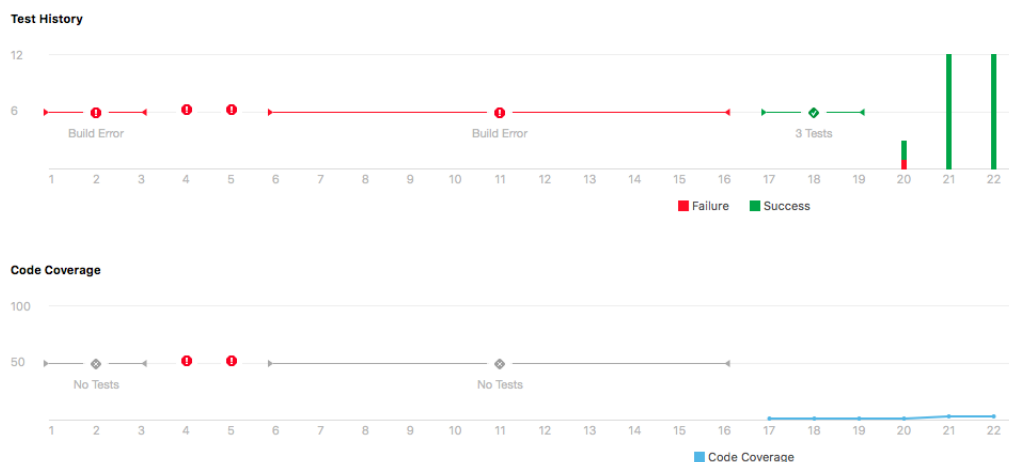


Slika 2.25: Odabir uređaja na kojima se pokreću testovi

prikazuju se detalji odabranog testiranja.

2.4. Osiguranje kvalitete

Osiguranje kvalitete (engl. *code quality*) proces je u sklopu programskog inženjerstva koje postavljanjem umjetnih restrikcija nastoji poboljšati kvalitetu koda i konačnog produkta. Restrikcije se najčešće definiraju u obliku pravila koja moraju biti zadovoljena. Pravila se mogu kretati od jednostavnih do vrlo složenih te uvelike ovise o korištenim alatima i potrebama tima.



Slika 2.26: Rezultati procesa testiranja i osiguranja kvalitete

U sklopu rada implementiram dvije provjere: provjeru pokrivenosti koda testovima i provjeru ispravnosti koda korištenjem alata Swiftlint.

Pokrivenost koda testovima (engl. *code coverage*) mjera je koja govori u kojem je postotku izvorni kod pokriven testovima. Svaki redak koda koji je barem jednom izvršen u procesu testiranja pokriven je testovima. Što je navedena mjera veća, to je više koda testirano. Zbog toga se može reći da veća pokrivenost koda testovima, u generalnom slučaju, vodi do bolje kvalitete konačnog produkta.

Međutim, navedena se mjera može vrlo lako zloupotrijebiti. Na primjer, moguće je napisati vrlo jednostavan test koji samo pokreće aplikaciju i time poziva značajan postotak koda. Zbog navedenog se u praksi često koriste modificirane verzije mjerenja pokrivenosti koda testovima koje procesu dodaju dodatna pravila te time nastoje utvrditi stvarnu kvalitetu testiranja[15].

Alat xcodebuild implementira mjerenje pokrivenosti koda testovima. Dovoljno je naredbi za pokretanje testova dodati argument `-showBuildSettings`. Ispis naredbe pohranjuje se kao skup datoteka koje služe za vizualan prikaz pokrivenosti koda testovima u Xcode aplikaciji. Uz sam postotak pokrivenosti koda testovima, Xcode prikazuje i pokrivenost pojedinog dokumenta te broj poziva svake pojedine linije koda.

Xcode Server također pruža navedenu opciju, ali u reduciranom obliku. Prikupljanje podataka o pokrivenosti koda testovima uključuje se odabirom opcije `CodeCoverage->Enabled` u postavkama bota. Slično Xcode aplikaciji, Xcode Server uz postotak pokrivenosti koda testovima prikazuje i pokrivenost pojedinog dokumenta. Slika 2.27 prikazuje pokrivenost datoteka testnog projekta testovima.

Name	Change	Coverage	iPhone 5 Simulator	iPhone 6 Simulator
▼ Diplomski_rad.app	-	<div><div></div></div>	71%	71%
▶ AppDelegate.swift	-	<div><div></div></div>	36%	36%
▶ SafeArray.swift	-	<div><div></div></div>	100%	100%
▶ LayoutKit.framework	-	<div><div></div></div>	0%	0%
▶ Nimble.framework	-	<div><div></div></div>	1%	1%

Slika 2.27: Pokrivenost datoteka testnog projekta testovima

Swiftlint je alat za statičku analizu koda napisanog u programskom jeziku Swift. Alat definira veliki broj pravila kojima nastoji osigurati praćenje stila i konvencija jezika Swift[6]. Većina pravila se odnosi na izgled i format koda, ali postoje i pravila koja nastoje izbjeći pojavu grešaka. Slični alati kreirani su za gotovo svaki programski jezik koji se koristi u praksi. Ovi se alati nazivaju alati za uređivanje koda (engl. *linting tools*). Navedene alate uglavnom kreira i održava zajednica.

Nepoštivanje pravila izaziva dojavu upozorenja (engl. *warning*) ili greške (engl. *error*). Moguće je kreirati nova pravila i modificirati ili isključiti postojeća. Veliki broj

timova definira vlastiti stil pisanja koda koji je moguće osigurati korištenjem navedenih funkcionalnosti.

Alati za uređivanje koda postaju nezaobilazni u praksi. Pridržavanje strogog formata pisanja koda olakšava timski rad, poboljšava čitljivost koda i izbjegava pojavu lako izbjegnutih grešaka.

Alat se pokreće pozivom naredbe `swiftlint` u početnom direktoriju projekta. Naredbu je moguće uključiti u proces izgradnje korištenjem `Run Script` faze. Pomoću Xcode aplikacije otvoriti željeni projekt te odabrati cilj koji izvršava izgradnju. U sekciji `Build Phases` odabirom opcije `Plus -> New Run Script Faze` kreirati novu fazu. Fazi dodati sadržaj skripte 2.12.

Skripta 2.12: Provjera postojanja i pokretanje Swiftlint alata

```
if which swiftlint >/dev/null; then
    swiftlint
else
    echo "warning: Swiftlint nije instaliran"
fi
```

Ako je poziv `swiftlint` naredbe dodan kao faza izgradnje, onda se naredba obavlja prilikom svake izgradnje. Ispis naredbe parsira se zajedno s ispisom `xcodebuild` alata te se rezultati dojavljuju zajedno. Ako naredba nije dodana projektu, onda je poziv potrebno ručno implementirati nakon obavljanja integracije.

3. Kontinuirana dostava

Kontinuirana dostava praksa je u sklopu programskog inženjerstva koja automatiziranjem procesa isporuke programske podrške nastoji olakšati i time povećati učestalost isporuke produkta u produkciju. Proces isporuke može biti kompleksan i vremenski zahtjevan. Prije isporuke produkt je potrebno izgraditi, testirati, kreirati artefakt koji se isporučuje te ga objaviti na željenoj platformi. Za izgradnju je potrebno dohvatiti željenu verziju repozitorija, pripremiti sustav te dohvatiti sve potrebne ovisnosti.

Navedeni proces vremenski je zahtjevan zbog čega se ne provodi često. Rijetko provođenje procesa isporuke uzrokuje nekoliko problema. Prvo, rijetkom isporukom veći se broj funkcionalnosti grupira i isporučuje zajedno. Što je veći broj funkcionalnosti isporučen zajedno i što duže te funkcionalnosti nisu isporučene, to je teže osigurati ispravnost isporučenih funkcionalnosti. Također, što duže funkcionalnost nije objavljena to je veći propušteni dobitak koji je funkcionalnost mogla donijeti. Svaka se funkcionalnost razvija s određenim ciljem. Nakon što je funkcionalnost razvijena, a prije nego što je objavljena, navedeni cilj ostaje bespotrebno neispunjen.

Automatizacija procesa isporuke značajno olakšava navedeni proces i time potiče češću isporuku. Dodatno, implementacija jednostavnog i automatiziranog procesa isporuke smanjuje mogućnost pojave ljudske pogreške. Kontinuirana dostava na navedeni način nastoji smanjiti trošak, rizik i vrijeme razvoja programske potpore te u isto vrijeme poboljšati kvalitetu i osigurati ispravnost programskog produkta[16].

Kao što je navedeno u uvodu poglavlja, programsku je potporu prije isporuke potrebno izgraditi, testirati te kreirati artefakt za isporuku. Iako različiti razvojni procesi različito nazivaju artefakt za distribuciju, zbog konzistentnosti ga nazivam arhiva. Arhiviranje je proces kreiranja arhive programske potpore. Arhiva omogućava instalaciju programske potpore na operacijskom sustavu za koji je programska potpora izgrađena.

Jednako tako prije automatizacije isporuke produkta potrebno je automatizirati izgradnju, testiranje, provjeru ispravnosti i arhiviranje programske potpore. Proces automatizacije izgradnje, testiranja i provjere ispravnosti naziva se kontinuirana integracija te je definirana u poglavlju 2. Kontinuirana dostava obuhvaća proces kontinuirane in-

tegracije te proces automatizacije isporuke.

Međutim, u praksi se pojam kontinuirana dostava češće koristi za proces automatizacije isporuke koji zahtijeva odvojenu implementaciju procesa kontinuirane integracije. Zbog jednostavnosti navedeni pristup koristim u ovom radu. Proces izgradnje, testiranja i provjere ispravnosti implementiram u sklopu kontinuirane integracije na koju dodajem automatizaciju isporuke i time ostvarujem kontinuiranu dostavu.

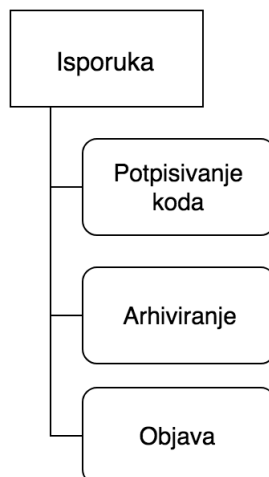
Važno je napomenuti da kontinuirana dostava ne isporučuje produkt automatski, već samo automatizira proces isporuke. Tim još uvijek mora pokrenuti proces isporuke za željenu verziju produkta. Automatska objava programske potpore obrađena je u sljedećem, 4 poglavlju.

Programsku potporu moguće je isporučiti na više načina - direktnom instalacijom na mobilni uređaj, ručnom distribucijom arhive, objavom arhiva korištenjem distribucijske platforme ili objavom arhive korištenjem App Store platforme. Navedeni tipovi isporuke dijele sličan proces izgradnje i arhiviranja, ali se značajno razlikuju u procesu objave arhive. Samim time i automatizacije navedenih načina isporuke značajno se razlikuju.

iOS operacijski sustav podržava četiri tipa isporuke: direktnu isporuku (engl. *direct distribution*), *ad hoc* isporuku (engl. *ad hoc distribution*), unutarnju isporuku (engl. *in-house distribution*) i isporuku korištenjem App Store platforme. Odabir načina isporuke ovisi o korisnicima za koje se obavlja isporuka.

Direktna isporuka programsku potporu instalira direktno na povezani mobilni uređaj. Mobilni uređaj mora biti povezan s računalom koje obavlja isporuku te mora biti registriran kao testni uređaj na Apple Developer platformi. Ovaj tip isporuke programsku potporu instalira na jedan uređaj koji mora biti direktno povezan s računalom, zbog čega se koristi gotovo isključivo u sklopu razvoja i za isporuku programske potpore unutar razvojnog tima. Dodatno, ovaj se tip isporuke ne može automatizirati niti bi to bilo pogodno. Zbog sigurnosnih razloga Apple nije objavio proces direktne isporuke, već ju je moguće obaviti isključivo korištenjem Xcode aplikacije.

Ad hoc i unutarnja isporuka vrlo su slične. Obje proizvode arhivu koju je potrebno samostalno dostaviti željenim korisnicima. Arhiva se može dostaviti ručno, npr. korištenjem poruke e-pošte ili se može objaviti na platformi za distribuciju programske potpore. Moguće je kreirati vlastitu platformu ili iskoristiti neku od velikog broja već postojećih. Isporuke se razlikuju u tome što uređaji, na koje se instalira programska potpora isporučena ad hoc načinom isporuke, moraju biti registrirani kao testni uređaji na Apple Developer platformi, dok to nije potrebno za unutarnju isporuku. Zbog navedenog je korištenjem unutarne isporuke programsku potporu moguće isporučiti



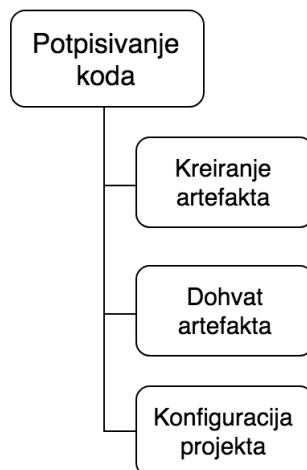
Slika 3.1: Proces isporuke

neograničenom broju korisnika. Za korištenje unutarnje isporuke potrebno je posjedovati *Enterprise* Apple Developer račun, skuplju verziju običnog Apple Developer računa.

Ad hoc isporuka najčešće se koristi za isporuku programske potpore unutar tima, npr. za isporuku testne verzije aplikacije timu za osiguranje kvalitete. Unutarnja isporuka koristi se gotovo isključivo za isporuku aplikacija razvijenih za unutarnje potrebe kompanije. Ovaj se tip isporuke ne smije iskoristiti za objavu aplikacija namijenjenih za javno tržište.

App Store isporuka koristi se za objavu programske potpore namijenjene za javno tržište. Za razliku od ostalih tipova isporuke, Apple strogo nadzire aplikacije objavljene ovim tipom isporuke. Svaka aplikacija koja zatraži objavu prolazi kroz strogi proces provjere kvalitete i podudaranja s velikim brojem Appleovih smjernica. Ovaj proces može trajati i nekoliko tjedana, zbog čega je nužno osigurati ispravnost aplikacije prije pokretanja procesa. Svaka izmjena aplikacije prolazi kroz nešto kraći proces provjere koji može trajati od nekoliko sati do nekoliko dana.

Iako se navedeni tipovi isporuke razlikuju u implementaciji, moguće ih je generalno podijeliti u tri ista koraka: potpisivanja koda, arhiviranja programske potpore i objave arhive. Svaki od koraka definiran je i automatiziran odvojeno u nastavku poglavlja. Slika 3.1 prikazuje navedenu podjelu.



Slika 3.2: Potpisivanje koda

3.1. Potpisivanje koda

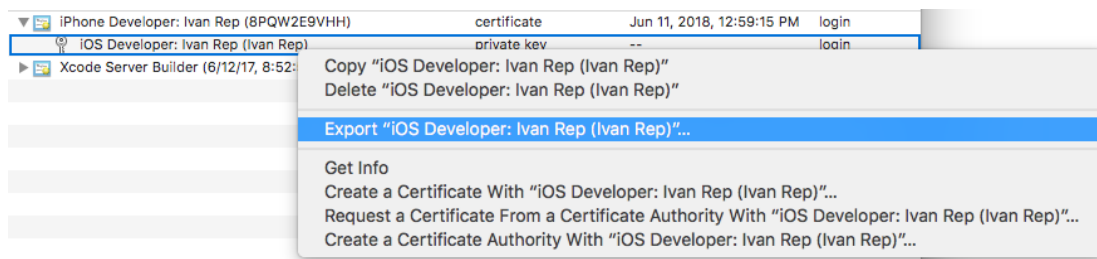
Kako bi osigurao autentičnost i neizmijenjenost programske potpore, iOS operacijski sustav implementira proces potpisivanja koda (engl. *code signing*). Korištenjem certifikata, identifikatora aplikacije (engl. *application identifier*) i pripremnih profila (engl. *provisioning profiles*) proces potpisuje i enkriptira kreiranu arhivu. Prije kreiranja arhive potrebno je kreirati i dohvatiti navedene artefakte te ispravno konfigurirati proces arhiviranja.

Proces postpisivanja koda može se podijeliti na tri faze: kreiranje artefakata, dohvat artefakata i konfiguracija procesa arhiviranja. Slika 3.2 prikazuje navedenu podjelu.

Ako potrebni artefakti ne postoje, potrebno ih je kreirati te nakon toga preuzeti i instalirati na računalu. Proces arhiviranja potrebno je zatim konfigurirati kako bi koristio ispravne artefakte.

Svi se artefakti kreiraju i dohvaćaju korištenjem `https://developer.apple.com` platforme. Potrebno je prijaviti se na platformu korištenjem Apple Developer računa, odabrati opciju `Certificates, Identifiers & Profiles` i željeni tip artefakta iz lijevog bočnog izbornika.

Certifikati se koriste za osiguranje nekoliko različitih procesa zbog čega postoji nekoliko tipova certifikata. U sklopu isporuke zanimaju nas dva tipa certifikata: razvojni (engl. *development*) i produkcijski (engl. *production*). Razvojni se certifikati koriste u razvoju i za direktnu isporuku programske potpore. Uobičajno je kreirati zaseban certifikat za svakog člana tima. Produkcijski certifikati koriste se za *ad hoc*, unutarnju i App Store isporuku. Standardno je kreirati jedan certifikat po timu koji obavlja isporuku. Certifikat se kreira pomoću CSR datoteke koju je moguće kreirati korištenjem



Slika 3.3: Izdvajanje privatnog ključa certifikata

Keychain Access aplikacije.

Potrebno je pokrenuti Keychain Access aplikaciju i u izborniku na vrhu ekrana odabrati opciju Certificate Assistant -> Request a Certificate. U novootvorenom prozoru treba unijeti adresu e-pošte Apple Developer računa, odabrati Save to disk opciju i lokaciju spremanja novokreirane datoteke. Kreiranu datoteku treba dostaviti u formu za kreiranje certifikata na Apple Developer platformi te je nakon toga obrisati.

Za korištenje certifikata potrebno je posjedovati certifikat i privatni ključ kojim je on kreiran. Certifikat je moguće preuzeti s Apple Developer platforme, dok je privatni ključ potrebno dohvatiti s računala koje ga posjeduje. Nakon kreiranja certifikata to je isključivo uređaj koji je kreirao CSR datoteku. Za dohvat privatnog ključa treba pokrenuti Keychain Access aplikaciju te pronaći željeni certifikat. Ako na uređaju postoji privatni ključ navedenog certifikata, moguće ga je izdvojiti korištenjem opcije *Export*. Slika 3.3 prikazuje izdvajanje privatnog ključa certifikata.

Ovako izdvojeni privatni ključ certifikata može se distribuirati. Nužno je osigurati tajnost privatnog ključa, jer se u suprotnom narušava sigurnost svih aplikacija koje koriste navedeni certifikat.

Identifikator aplikacije identificira i opisuje servise koje koristi aplikacija koja se isporučuje. Identifikator aplikacije definiran je pomoću identifikatora paketa (engl. *bundle identifier*), niza znakova jedinstvenog za svaku aplikaciju. Identifikator paketa najčešće je sljedećeg oblika: `com.ime_kompanije.ime_aplikacije`. Postoje dva tipa identifikatora aplikacije: *wildcard* i eksplicitni identifikatori aplikacije. Wildcard identifikatori aplikacije specificiraju samo dio identifikatora paketa te ih karakterizira asterisk (*) u sklopu imena, na primjer `com.rep.*`. Sve aplikacije čiji identifikator paketa započinje navedenim nizom znakova mogu biti isporučene korištenjem navedenog identifikatora. Wildcard identifikatori se mogu koristiti u razvoju programske potpore. Eksplicitni identifikatori jedinstveno identificiraju aplikaciju, na primjer `com.rep.testna_aplikacija` te ih se koriste za isporuku specifične



Slika 3.4: Postavljanje artefakta za potpisivanje koda

aplikacije.

Tijekom kreiranja identifikatora aplikacija potrebno je specificirati ime identifikatora, dodati identifikator paketa i odabrati servise koje aplikacija koristi. Nakon kreiranja identifikator je potrebno preuzeti i instalirati korištenjem Keychain Access aplikacije.

Pripremni profil povezuje identifikator aplikacije i certifikat. Svaki od četiriju tipova isporuke definira vlastiti tip pripremnog profila. Kreiranje pripremnog profila započinje odabirom tipa profila, odgovarajućeg identifikatora aplikacije te certifikata. Pripremni profil za direktnu isporuku zahtijeva razvojni certifikat, dok pripremni profili ostalih tipova isporuka zahtijevaju produkcijski certifikat.

Potrebno je preuzeti novokreirane artefakte i pripremiti ih za korištenje. Profile je potrebno spremiti na računalu koje obavlja isporuku u direktoriju `~/Library/MobileDevice/ProvisioningProfiles/`, a certifikate je potrebno pokrenuti korištenjem Keychain Access aplikacije.

Prije pokretanja arhiviranja potrebno je konfigurirati projekt kako bi se u sklopu arhiviranja koristili ispravni artefakti. Pomoću Xcode aplikacije otvoriti projekt, odabrati željeni cilj u sekciji *General*, postaviti *Debug* i *Release* certifikate za potpisivanje koda.

Direktna i *ad hoc* isporuka zahtijevaju registraciju mobilnog uređaja kao testnog uređaja korištenjem Apple Developer platforme. Uređaj se registrira korištenjem njegovog UUID identifikatora. UUID se može dohvatiti korištenjem iTunes aplikacije dostupne u sklopu instalacije macOS operacijskom sustavu. Potrebno je povezati mobilni uređaj s računalom korištenjem USB kabela te u iTunes aplikaciji odabrati povezani mobilni uređaj i kopirati njegov UUID. Slika 3.5 prikazuje lokaciju UUID broja u iTunes aplikacije.

3.1.1. Automatizacija potpisivanja koda

Ručno se kreiranje i održavanje certifikata i profila može vrlo lako zakomplicirati. Ne samo da je proces kreiranja sam po sebi složen, već je potrebno i ispravno distribuirati artefakte bez ugrožavanja njihove sigurnosti. Artefakti se mogu slobodno distribuirati, ali je njihove privatne ključeve nužno održati tajnim. Do prije nekoliko godina cijeli se proces odrađivao ručno. Nakon što bi se artefakti kreirali, zajedno bi se s privatnim ključem pohranili na sigurnu lokaciju te distribuirali po potrebi.

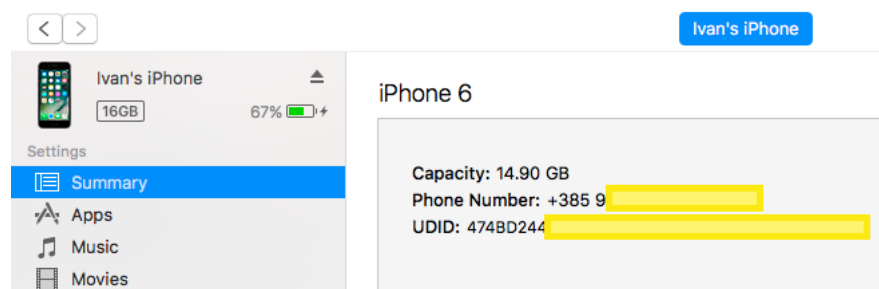
Danas na tržištu postoji nekoliko pristupa i alata koji olakšavaju i automatiziraju proces potpisivanja koda. Automatizacija potpisivanja koda oslanja se na javni API Apple Developer platforme koji omogućuje kreiranje, modificiranje i dohvat svih artefakata navedenih u prošlom odlomku.

Također, većina se zajednice zalaže za pohranu artefakata zajedno s pripadajućim privatnim ključevima u jednom tajnom git repozitoriju. Na ovaj se način njihova pohrana i dohvat značajno pojednostavljuju te je cijeli proces moguće jednostavno automatizirati. Međutim, pohrana sigurnosnih podataka u git repozitoriju donosi i određeni sigurnosni rizik. Pristup repozitoriju nužno je detaljno nadzirati. Preporuča se SSH protokol definiran u 2.2.1 odlomku. Dodatno, korisno je repozitorij pokrenuti unutar lokalne mreže kako ne bi bio vidljiv izvan mreže[7].

Automatizaciju potpisivanja koda u ovom radu ostvarujem korištenjem alata *match* koji je dio *fastlane* familije[11]. Alat samostalno kreira, dohvaća i priprema potrebne artefakte i time značajno olakšava proces potpisivanja koda.

Alat se *match* inicijalizira pozivanjem naredbe `fastlane match init` u početnom direktoriju projekta. Proces inicijalizacije zahtijeva unos lokacije git repozitorija te autorizaciju pristupa repozitoriju. Lokacija repozitorija zajedno se s ostalim parametrima alata sprema u `Matchfile` datoteku gdje ih je moguće modificirati.

Match radi povećanja sigurnosti zahtijeva da SSH ključ, korišten u procesu SSH autentifikacije, bude vidljiv samo trenutnom korisniku operacijskog sustava. Skripta



Slika 3.5: Dohvat UUIDa uređaja korištenjem iTunes aplikacije

3.1 modificira prava pristupa za odabrani ključ kako bi njegovo korištenje bilo dostupno samo trenutnom korisničkom računu operacijskog sustava.

Skripta 3.1: Ograničavanje prava pristupa SSH ključu na samo trenutnog korisnika

```
chmod 600 ~/.ssh/{imeključa}.pub
```

Ujedno je potrebno konfigurirati SSH protokol kako bi koristio željeni ključ. Treba kreirati novu datoteku `.ssh/config` sa sadržajem skripte 3.2.

Skripta 3.2: Postavke SSH protokola za alat *match*

```
Host *
UseKeychain yes
AddKeysToAgent yes
IdentityFile ~/.ssh/{imeključa}
```

Alat *match* sve potrebne certifikate i profile kreira te dohvaća automatski. Dovoljno je pokrenuti alat i specificirati tip isporuke koji se koristi. Alat definira četiri tipa koji odgovaraju prethodno definiranim tipovima isporuke: *development*, *adhoc*, *enterprise* i *appstore*. Naredba u nastavku dohvaća i po potrebi kreira artefakte za ad hoc isporuku.

Skripta 3.3: Dohvaćanje artefakta pomoću dodatka *match* za ad hoc isporuku

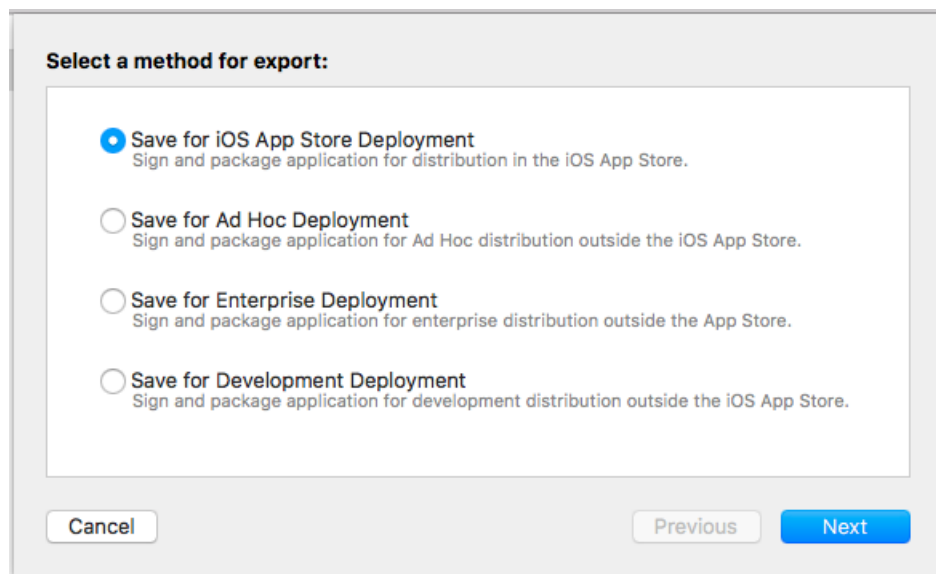
```
match (type: 'adhoc')
```

Alat ne obavlja konfiguraciju procesa arhiviranja, no s obzirom na to da ju je potrebno napraviti samo jednom, ista se najčešće obavlja ručno. Pomoću *match* alata treba kreirati sve potrebne artefakte, odnosno pozvati *match* naredbe za svaki tip isporuke koja se koristi. Navedene će naredbe kreirati i dohvatiti potrebne artefakte. Zatim treba otvoriti Xcode aplikaciju i odabrati željenu shemu. Na početnoj, *General* sekciji sheme u odlomku *Signing* za svaku konfiguraciju treba postaviti odgovarajući pripremni profil.

Nakon spremanja promjena cijeli se proces potpisivanja koda svodi na poziv alata *match* sa željenim tipom isporuke.

3.2. Arhiviranje

Arhiviranje je proces kreiranja artefakta pomoću kojeg se aplikacija instalira na korisničkom uređaju. Artefakt se naziva arhiva aplikacije te je označena `.ipa` nastavkom.



Slika 3.6: Odabir tipa isporuke u procesu kreiranja arhive

Za instalaciju aplikacije, uz arhivu, potrebna je i datoteka koja opisuje, verificira i locira arhivu. Navedena se datoteka naziva manifest te je označena `.plist` nastavkom.

Proces arhiviranja za *ad hoc*, unutarnju isporuku i App Store isporuku razlikuje se jedino u odabiru tipa isporuke. Arhiviranje obavlja alat `xcodebuild`. Međutim, alat je vrlo složen i pruža veliki broj opcija zbog čega na tržištu postoji nekoliko alata koji olakšavanju njegovo korištenje. U sklopu rada za ručno arhiviranje koristim aplikaciju Xcode, a za automatizirano arhiviranje alat `fastlane`, to jest njegov dodatak `gym`. Proces arhiviranja korištenjem alata `xcodebuild` prikazan je u dodatku B.

Korištenjem aplikacije Xcode potrebno je odabrati opciju `Product -> Archive` iz izbornika na vrhu ekrana. Nakon kraćeg perioda otvara se novi prozor koji prikazuje sve dosad kreirane arhive. Jedino što je potrebno jest izdvojiti arhivu za objavu. U prozoru je potrebno odabrati opciju `Export`. Tada se otvara novi prozor u kojem je potrebno odabrati željeni tip isporuke te odabrati između nekoliko opcija kreiranja arhive. Slika 3.6 prikazuje početni ekran prozora, odnosno odabir tipa isporuke arhive. Nakon odabira željene opcije potrebno je slijediti upute za dovršenje izdvajanja arhive.

Proces generira dvije prethodno definirane datoteke. Arhivu aplikacije s `.ipa` nastavkom i manifest aplikacije s `.plist` nastavkom.

Direktna isporuka aplikaciju instalira na mobilni uređaj koji je direktno povezan s računalom koje obavlja isporuku. Zbog navedenog, proces arhiviranja u sklopu direktne isporuke ne rezultira arhivom, već je direktno instalira na uređaj. Dodatno, ovaj je tip arhiviranja moguće ostvariti isključivo korištenjem Xcode aplikacije. Apple zbog sigurnosnih razloga ne želi objaviti proces direktne instalacije aplikacije na uređaj.

Arhiviranje i instalacija obavljaju se odabirom željenog povezanog mobilnog uređaja i pokretanjem `run` operacije na željenoj shemi.

3.2.1. Automatizacija arhiviranja

Xcode Server implementira funkcionalnost arhiviranja aplikacije u sklopu obavljanja integracije. Međutim, funkcionalnost je vrlo ograničena. Nije moguće dinamički konfigurirati artefakte koji se koriste za potpisivanje koda niti podesiti argumente arhiviranja.

Zbog toga automatizaciju arhiviranja ostvarujem korištenjem fastlane dodatak *gym*. Alat za izgradnju interno koristi alat `xcodebuild` i na prvi se pogled od njega ne razlikuje značajno. Međutim, ako se za cijeli proces isporuke koristi fastlane familija, onda korištenje alata *gym* značajno olakšava implementaciju.

Alat izgrađuje i arhivira aplikaciju te omogućava vrlo jednostavnu konfiguraciju oba procesa. Također, alat automatski detektira i koristi artefakte za potpisivanje koda dohvaćene korištenjem dodatka *match*. Skripta 3.4 prikazuje arhiviranje projekta za *ad hoc* tip isporuke.

Skripta 3.4: Arhiviranje aplikacije za ad hoc isporuku pomoću dodatka *gym*

```
gym(scheme: "Diplomski_rad", export_method: "ad-hoc")
```

Kreiranje arhive pomoću `xcodebuild` alata ostvaruje se pozivanjem naredbe uz korištenje opcije `archive`. Shema i cilj na temelju kojih se izgrađuje arhiva definira se jedna kao i kod drugih opcija naredbe `xcodebuild`.

3.3. Objava

Za dovršetak isporuke željenim korisnicima potrebno je omogućiti instalaciju aplikacije korištenjem kreirane arhive. Navedeni se proces naziva objava programske potpore. Isporuka programske potpore korištenjem App Store platforme arhivu, naravno, objavljuje na App Store platformi. S druge strane, *ad hoc* i unutarnja isporuka arhivu mogu objaviti na nekoliko načina.

Kako bi pokrenuo instalaciju ili osvježanje verzije aplikacije, korisnik mora preuzeti ispravno konfigurirani manifest. Manifest uz podatke o aplikaciji sadrži i lokaciju arhive. Nakon preuzimanja manifesta operacijski sustav samostalno pokreće dohvat arhive i instalaciju aplikacije.

Najjednostavniji, ali i najograničeniji, način objave jest manifest i arhivu objaviti na generalnoj platformi za dijeljenje podataka kao što su Dropbox i Google Drive. Navedene platforme omogućavaju kreiranje URI-a za pojedinu datoteku. Potrebno je kreirati URI za objavljeni arhiv i dodati je manifestu. Iako je ovaj način isporuke na prvi pogled jednostavan, vremenski je vrlo zahtjevan. Svaku novu verziju potrebno je ručno dodati na platformu, konfigurirati te zatim obavijestiti korisnike o novoj verziji.

Za objavu je moguće kreirati vlastitu platformu. Na primjer, jednostavnu HTML stranicu koja omogućava preuzimanje manifesta koji referencira arhiv objavljen u vlastitom poslužitelju. Međutim, na tržištu postoji nekoliko platformi koje već implementiraju navedenu funkcionalnost. U sklopu rada promatram dvije platforme ovog tipa: *Mobile Device Management* i *Crashlytics* platforme.

Za objavu programskog produkta za iOS operacijski sustav moguće je koristiti službeni Appleov alat *Mobile Device Management*, *MDM*. Alat je namijenjen za jednostavnu isporuku i distribuciju aplikacija koje nisu namijenjene za javnu objavu. Po funkcionalnosti koje pruža, platforma je vrlo slična App Store platformi. MDM omogućava jednostavnu objavu aplikacije i kontrolu pristupa te omogućava automatsku izgradnju i osvježanje verzije aplikacije. Međutim, alat je skup te nije popularan u zajednici.

3.3.1. Crashlytics

Crashlytics trenutno je najpopularnija od svih platformi za objavu programske potpore za iOS operacijski sustav. Platforma je jednostavna, besplatna te omogućava automatizaciju procesa isporuke. Navedenu platformu koristim u ovom radu. Razvoj platforme započeo je 2011. godine, a osnovana je s ciljem jednostavnog praćenja i dojava pogreška pri izvršavanju mobilnih aplikacija. Kompaniju je 2013. godine kupio Twitter, a početkom 2017. godine preuzeo Google. Danas alat uz praćenje pogrešaka olakšava distribuciju aplikacija i praćenje velikog broja metrika.

Prije implementacije isporuke potrebno je kreirati profil aplikacije na platformi. Kreiranje profila odrađuje Crashlytics biblioteka koju je potrebno dodati i pokrenuti zajedno s aplikacijom.

Prvo treba dodati ovisnosti definirane u skripti 3.5 u `Podfile` datoteku.

Skripta 3.5: Ovisnosti potrebne za objavu korištenjem Crashlytics platforme

```
pod 'Fabric'  
pod 'Crashlytics'
```

Nakon toga treba kreirati novu Run scrip fazu za željeni cilj te joj dodati skriptu 3.6.

Skripta 3.6: Fabric Run Script faza

```
"${PODS_ROOT}/Fabric/run" {api_kljuc} {tajni_kljuc}
```

Potrebno je dohvatiti API i tajni ključ korištenjem Fabric platforme. Zatim je potrebno prijaviti se na Fabric platformu korištenjem poveznice <https://www.fabric.io/settings/organizations>. Nakon prijave treba odabrati željenu organizaciju te preuzeti ključeve.

Na kraju se mora dodati naredba `Fabric.with([Crashlytics.self])` na početak `application(application: didFinishLaunchingWithOptions:)` metode klase `AppDelegate`. Registracija aplikacije dogodit će se pri prvom pokretanju aplikacije. Profil aplikacije nakon njezina pokretanja nalazi se na stranici <https://fabric.io/home>.

Nakon kreiranja profila, arhivu i manifest moguće je ručno dodati na Crashlytics platformu.

Automatizacije objave Za automatizacije objave korištenjem Crashlytics platforme koristim fastlane dodatak *crashlytics*. Dodatak omogućava jednostavnu objavu aplikacije uz minimalnu konfiguraciju projekta te se nadovezuje direktno na dodatak gym. Za pokretanje objave potrebno je pokrenuti dodatak nakon kreiranja arhive aplikacije. Skripta 3.7 prikazuje fastlane stazu koja izgrađuje, arhivira i objavljuje aplikaciju na Crashlytics platformi.

Skripta 3.7: Fastlane staza za isporuku korištenjem Crashlytics platforme

```
lane :develop do
  increment_build_number

  match(app_identifier: "com.rep.Diplomski-rad.
    development", type: "development")
  gym(scheme: "Diplomski_rad", export_method: "
    development")

  crashlytics(
    api_token: {api_kljuc},
    build_secret: {tajni_kljuc},
```

```

        groups: 'Rep'
    )

end

```

Dodatno, kako prilikom pokretanja naredbe ne bih morao unositi crashlytics API i tajni ključ, iste je moguće specificirati kao argumente crashlytics naredbe. Grupa testera kojoj će aplikacija biti vidljiva može se dodati kao argument Crashlytics naredbi - groups: {ime_grupe}.

3.3.2. App Store

Objava javnih iOS aplikacija ostvaruje se korištenjem App Store platforme. App Store je službena platforma za distribuciju programske potpore za macOS, iOS, tvOS i watchOS operacijske sustave. Uz dostupnost aplikacije svim korisnicima navedenih operacijskih sustava, platforma pruža i brojne druge funkcionalnosti kao što su automatsko instaliranje novih verzija aplikacija i praćenje ponašanja korisnika.

Objava programske potpore na App Store platformu obavlja se korištenjem iTunes Connect web-stranice. iTunes Connect omogućava kreiranje i modificiranje profila aplikacije te dodavanje arhive koja se objavljuje. Potrebno je prije objave kreirati profil aplikacije korištenjem iTunes Connect alata.

Kreiranje profila obavlja se na <https://itunesconnect.apple.com/> stranici. Nakon prijave potrebno je odabrati opciju My Apps te kreirati novi profil. Slika 3.7 prikazuje formu za kreiranje profila aplikacije.

Nakon kreiranja profila aplikacije potrebno je ispuniti dodatne podatke o aplikaciji kao što su opis i kategorija aplikacije. Slično kao i kod Crashlytics platforme, sada je moguće jednostavno dostaviti arhivu i manifest aplikacije.

Iako isporuka aplikacija korištenjem App Store platforme donosi brojne prednosti, ujedno donosi i brojne nedostatke. Ovaj je tip isporuke poprilično složen te traje puno duže u usporedbi s ostalim tipovima isporuke. Apple vrlo strogo regulira aplikacije koje se nalaze na App Store platformi.

Proces provjere aplikacije zna trajati i do nekoliko tjedana te često rezultira odbijanjem aplikacije. Aplikacija može biti odbijena jer je previše slična već postojećoj aplikaciji, jer se ne slaže s nekom od Appleovih politika, zbog loše implementacije i brojnih drugih razloga. Nakon otklanjanja razloga odbijanja potrebno je ponovno proći cijeli proces.

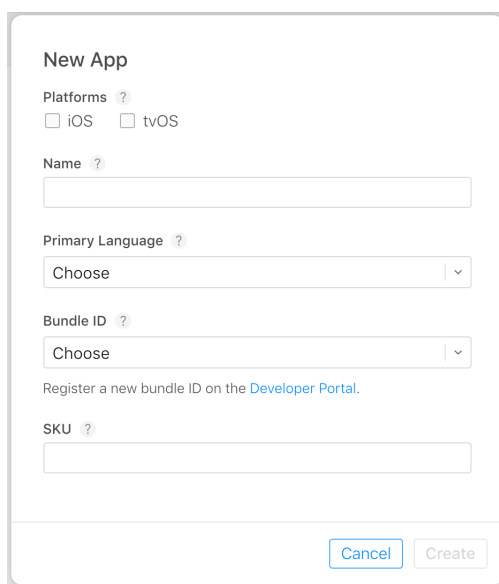
Za objavu nove verzije aplikacije potrebno je proći nešto blažu provjeru. Ona može

trajati od nekoliko sati do nekoliko dana te najčešće završava odobrenjem promjena. Odbijanje promjena vrlo je rijetko te služi prvenstveno za prevenciju očitog narušavanja Appleovih politika.

Cijeli je proces isporuke složen i dugotrajan. Iako se u zadnjih nekoliko godina proces značajno poboljšao, velik broj članova industrije još je uvijek vrlo nezadovoljan. Međutim, Apple zabranjuje isporuku aplikacija za javnu uporabu korištenjem bilo koje druge metode. Ako se navedeni proces uspoređi s procesom isporuke web aplikacija, onda su njegovi nedostaci jasno vidljivi. Nova verzija web aplikacije isporučuje se jednostavno promjenom verzije koja se nalazi na poslužitelju. Prvi sljedeći korisnik koji pristupi poslužitelju automatski koristi sljedeću verziju. Ne samo da je proces isporuke iOS aplikacija daleko teži i sporiji, već korisnik mora samostalno preuzeti novu verziju aplikacije.

Apple nastoji što jednostavnije i neprimjetnije osvježiti verzije aplikacije. Aplikacije se automatski osvježuju kad je uređaj spojen na WiFi mrežu te je dovoljno napunjen. Međutim, određeni broj korisnika isključuje ovu funkcionalnost ili ručno odobrava svaku novu verziju aplikacije. Zbog navedenog uvijek postoji određeni postotak korisnika koji vrlo dugo koriste starije verzije aplikacije. Ovaj problem iziskuje dugoročnu podršku starijih verzija aplikacije što značajno otežava razvoj.

Zbog navedenih razloga prominentni pojedinci u iOS zajednici predviđaju skoru zamjenu App Store platforme nekim boljim načinom isporuke. Međutim, Apple nije objavio nikakvu naznaku ovog te smo zasad ograničeni sustavom koji imamo.

The image shows a 'New App' form from the iTunes Connect web interface. The form is titled 'New App' and contains several fields for creating a new application. At the top, there is a 'Platforms' section with two checkboxes: 'iOS' and 'tvOS'. Below this is a 'Name' field with a text input box. The 'Primary Language' field is a dropdown menu currently showing 'Choose'. The 'Bundle ID' field is also a dropdown menu showing 'Choose'. Below the Bundle ID field, there is a link that says 'Register a new bundle ID on the Developer Portal.' The 'SKU' field is a text input box. At the bottom right of the form, there are two buttons: 'Cancel' and 'Create'.

Slika 3.7: Forma za kreiranje profila aplikacije na iTunes Connect web-stranici

App Information

This information is used for all platforms of this app. Any changes will be released with your next app version.

Diplomski rad

37

Subtitle ?

Optional

30

General Information

Bundle ID ? [Register a new bundle ID.](#)

Diplomski rad - com.rep.Diplomski-rad

Your Bundle ID com.rep.Diplomski-rad

SKU ?

Diplomski_rad

Apple ID ?

Slika 3.8: iTunes Connect profil aplikacije s označenim identifikatorom aplikacije

Automatizacija objave Isporuku programske potpore za App Store platformu implementiram pomoću fastlane dodatka *deliver*[9]. Dodatak korištenjem javnog API-ja iTunes Connect platforme automatizira dostavu arhive i manifesta. Alat se direktno nadovezuje na alat *gym*, samostalno detektira potrebne artefakte te omogućava jednostavnu konfiguraciju procesa.

Inicijalizacija *deliver* dodataka prikazana je u skripti 3.8. Naredba projekt povezuje s iTunes Connect platformom kako bi ubuduće dodatak mogao samostalno obaviti objavu. Za autorizaciju pristupa potrebno je unijeti korisničko ime i lozinku Apple Developer računa. Inicijalizaciju treba dovršiti unosom iTunes identifikatora profila aplikacije. Identifikator je moguće dohvatiti korištenjem iTunes Connect platforme. Nakon prijave na web-stranicu `https://itunesconnect.apple.com` je potrebno odabrati opciju *My Apps* te odabrati željenu aplikaciju. Slika 3.8 prikazuje lokaciju identifikatora aplikacije na profilu aplikacije.

Skripta 3.8: Inicijalizacija *deliver* dodatka

```
fastlane deliver init
```

Naredba kreira nekoliko datoteka u *fastlane* direktoriju. Tekstualna datoteka *Deliverfile* pohranjuje podatke vezane uz objavu aplikacije na App Store platformi kao što su korisničko ime Apple Developer računa te identifikator aplikacije. Direktorij *metadata* sadrži nekoliko dokumenata koji omogućavaju jednostavan unos

podataka koji će se koristiti prilikom isporuke aplikacije na App Store platformu. Na primjer, pomoću dokumenta `description.txt` moguće je unijeti opis aplikacije. Dodatno, moguće je kreirati zaseban opis za svaki jezik koji Apple podržava.

Nakon inicijalizacije dodatka je isporuku moguće obaviti jednostavno pozivom `deliver` alata. Naravno, kreirana arhiva mora biti potpisana certifikatima i profilima za isporuku na App Store platformi.

Staza 3.9 implementira cijeli proces isporuke aplikacije na App Store platformu.

Skripta 3.9: Isporuka na App Store platformu korištenjem dodatka `deliver`

```
lane :release do
  match(type: "appstore")
  gym(scheme: "Production", export_method: "app-store")
  deliver
end
```

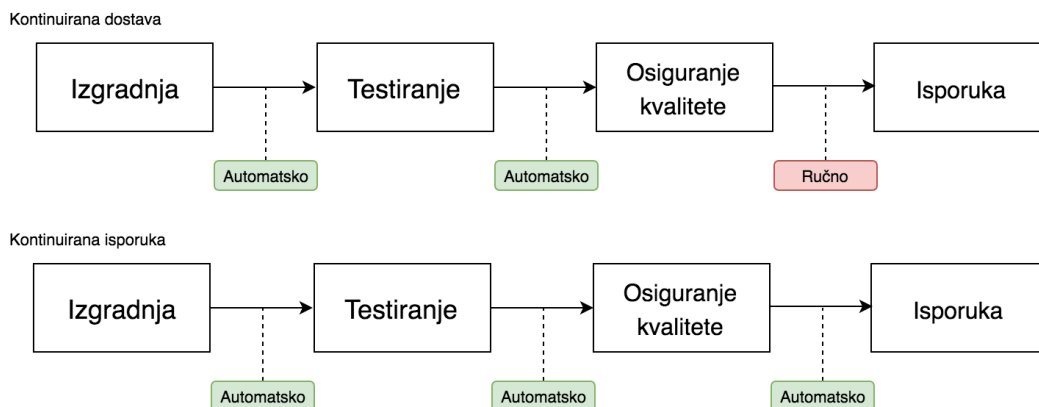
Staza korištenjem dodatka `match` dohvaća potrebne certifikate i profile, pomoću dodatka `gym` izgrađuje arhivu aplikacije te na kraju arhivu objavljuje koristeći `deliver` dodatak.

4. Kontinuirana isporuka

Kontinuirana isporuka (engl. *continuous deployment*) praksa je u sklopu programskog inženjerstva koja automatskom isporukom programske potpore nastoji olakšati proces i povećati učestalost isporuke razvijenih funkcionalnosti. Slično kao što kontinuirana integracija automatski obavlja proces integracije za svaku novu verziju repozitorija, tako i kontinuirana isporuka automatski obavlja proces isporuke za svaku novu verziju repozitorija. Međutim, dok kontinuirana integracija nad svakom verzijom obavlja isti proces integracije, kontinuirana isporuka provodi nekoliko različitih procesa isporuke.

Automatizacija procesa isporuke zahtijeva prethodnu implementaciju procesa isporuke. U sklopu rada koristim proces isporuke definiran u sklopu kontinuirane dostave, odnosno u poglavlju 3. Jednako tako proces isporuke zahtijeva prethodnu implementaciju procesa integracije. Zbog navedenog se kontinuirana isporuka može smatrati nastavkom kontinuirane dostave i samim time nastavkom na kontinuiranu integraciju. Razlika između kontinuirane dostave i kontinuirane isporuke prikazana je na slici 4.1[3].

Kontinuirana isporuka vrlo je mlada praksa. Termin je prvi puta iskoristio poznati autor Kent Beck u raspravi o agilnom razvoju programske potpore. Kent Beck je neisporučenu funkcionalnost usporedio s inventarom koji nije prikazan korisniku.



Slika 4.1: Usporedba kontinuirane dostave i kontinuirane isporuke



Slika 4.2: Broj dnevnih isporuka promjena u produkciju kompanije *Intercom*

Termin kontinuirana isporuka prvi je puta definiran u članku *The Deployment Production Line* autora Jeza Humblea, Chrisa Reada i Dana Northa[5]. Danas praksu razvija i unaprjeđuje uglavnom zajednica web programera koji je, zbog jednostavnosti isporuke programske podrške za web-poslužitelje, mogu jednostavno implementirati.

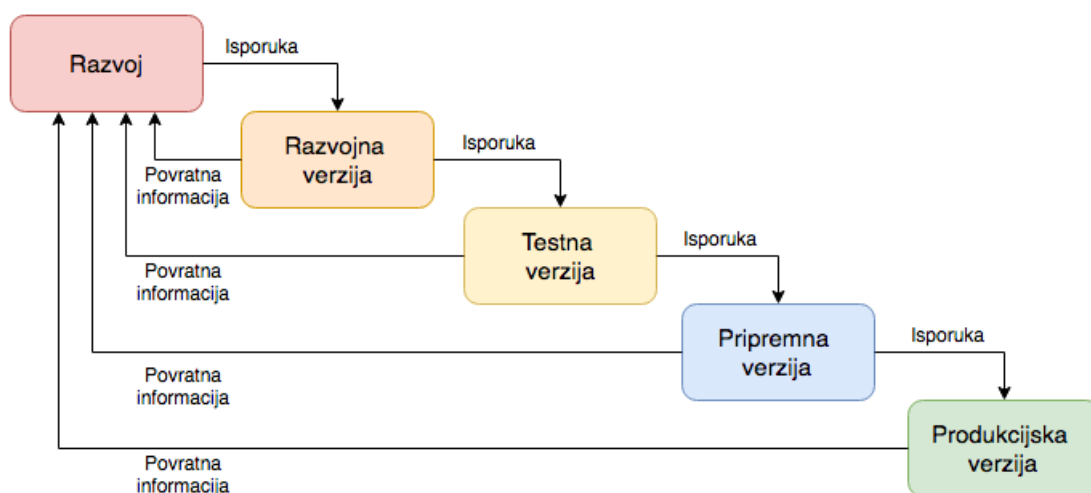
Isporuka programske podrške za web-poslužitelje ima značajnu prednost u usporedbi s ostalom programskom potporom. Najvažnije, web-aplikacije nije potrebno instalirati na uređaj. Korisnik promjene vidi čim su one dodane na poslužitelju. Dodatno, razvojni tim može direktno upravljati koju će verziju programske potpore korisnik vidjeti. U usporedbi s tim, isporuka iOS aplikacija prvo mora proći provjeru Appleovog tima te korisnik zatim mora novu verziju preuzeti na uređaj.

Intercom, novonastala kompanija bazirana u Irskoj proces kontinuirane isporuke implementirala je krajem 2012. godine. Uz automatizirano obavljanje isporuke, kompanija u sklopu procesa implementira i brojne složenije funkcionalnosti. Na primjer, kompanija isporuku obavlja u fazama. Umjesto isporuke nove promjene svim korisnicima, promjene vidi samo manji postotak iskusnih korisnika. U blogu objavljenom 2016. godine bilježi se stabilan porast broja dnevnih isporuka za promjenu od produkcija. Od ispod 10 isporuka u 2012. godini do preko 80 u 2015. godini. Slika 4.2 prikazuje broj dnevnih isporuka promjena u produkciju kompanije Intercom[4].

Nažalost, praksa je puno slabije zastupljena u razvoju programske potpore za iOS operacijski sustav. Postoji nekoliko neslužbenih objava u kojima članovi zajednici raspravljaju o procesu automatizacije, međutim navedeni su procesi značajno reducirani u usporedbi s procesima koji se koriste u razvoju programske potpore za web-platformu.

4.1. Tipovi isporuke

Ovisno o potrebama tima, kontinuirana isporuka može automatsku isporuku obavljati i za desetak različitih verzija programske potpore. Navedeni broj verzija koristi se zbog osiguranja kvalitete programske potpore. Nema smisla tek razvijenu funkcionalnost objaviti direktno krajnjim korisnicima. Funkcionalnost je prvo potrebno testirati te osigurati njezinu spremnost za objavu u produkciji. Umjesto direktne isporuke nove verzije u produkciju, ista prolazi nekoliko stupnjeva provjere ispravnosti. Drugim riječima, verzija se isporučuje korištenjem nekoliko različitih procesa isporuke. U sklopu rada isporučujem četiri verzije programske potpore: produkcijsku, pripremnu, testnu i razvojnu verziju. Tipovi i redoslijed isporuke verzija prikazani su na slici 4.3.



Slika 4.3: Postupna isporuka programske potpore skupovima korisnika

Razvojna verzija programske potpore prikazuje najnovije stanje aplikacije. Razvojna verzija sadrži najnovije implementirane izmjene koje još nisu nužno prošle proces osiguranja kvalitete. Ovu verziju aplikacije koriste članovi tima za provjeru razvijenih funkcionalnosti.

Testnu aplikaciju koristi tim za osiguranje kvalitete kako bi osigurao ispravnost novorazvijenih funkcionalnosti. Kako bi osigurao ispravnost, tim provjerava sve izmjene ostvarene od zadnje verzije aplikacije. Zbog navedenog poželjno je da je broj izmjena između verzija manji, ali dobro dokumentiran. Verzija ovog tipa aplikacije osvježava se otprilike jednom tjedno.

Pripremna verzija aplikacije služi za završnu provjeru aplikacije koja se izdaje u produkciju. Za razliku od testne verzije aplikacije koja sadrži funkcionalnosti korištenje samo u testiranju, pripremna aplikacija mora biti identična onoj koja se želi

isporučiti. Jedino se na ovaj način može provjeriti ispravnost aplikacije u produkcijskom okruženju. Verzija aplikacije kreira se jednom kada tim odluči objaviti novu produkcijsku verziju.

Produkcijska aplikacija je aplikacija trenutno dostupna korisniku. Učestalost isporuke ovisi o timu, ali zbog Appleovog procesa provjere gotovo se nikad ne obavlja češće od jednom tjedno.

Iz navedenog je jasno da trenutni proces ne slijedi principe kontinuirane isporuke. Spor proces izdavanja promjena u produkciju rezultira većim brojem ljudskih pogrešaka i lošijim iskustvom korisnika. Zbog navedenog, automatizacijom procesa ne samo da pokušavam smanjiti vrijeme koje tim ulaže u isporuku, već i smanjiti period između dvije isporuke.

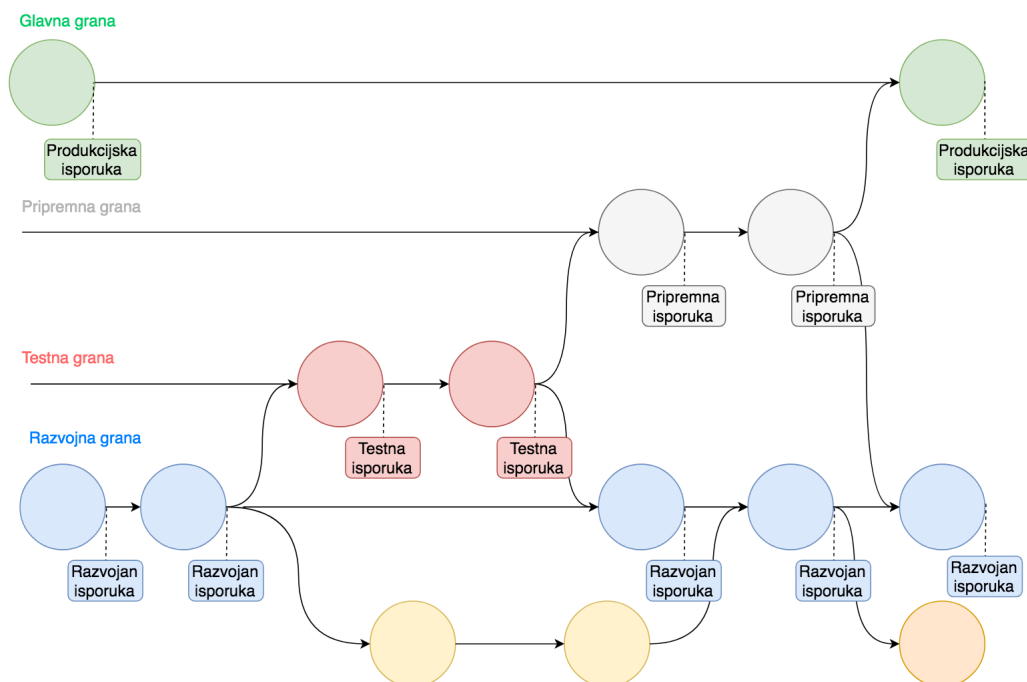
4.2. Odabir tipa isporuke

Međutim, pitanje je kako odrediti koju verziju repozitorija objaviti kojim tipom isporuke? Kako bih istovremeno implementirao isporuku različitih verzija aplikacije te zadržao fleksibilnost razvoja, koristim modificirani Gitflow tok rada.

Standardni Gitflow proces sadrži tri grane: glavnu, razvojnu i pripremnu granu. Glavna grana (engl. *master branch*) predstavlja trenutno stanje programske potpore u produkciji. Svaka verzija ove grane obilježena je oznakom koja sadrži ljudski čitljivu verziju aplikacije i popis promjena obavljenih od prošle verzije. Razvojna grana (engl. *develop branch*) predstavlja trenutno stanje aplikacije u razvoju. Objava promjena u produkciju obavlja se korištenjem pomoćne pripremne grane koja se kreira iz razvojne grane. Nakon provjere ispravnosti promjena, one se dodaju glavnoj grani i time objavljuju u produkciju. Nove funkcionalnosti razvijaju se u zasebnoj grani koja se za potrebe razvoja funkcionalnosti kreira iz razvojne grane te se po završetku razvoja u nju spaja. Pregled cijelog procesa dostupan je u odlomku 2.2.1.

Uz navedene grane koristim i testnu granu koja služi za održavanje i isporuku testne aplikacije. Kako bi se olakšali i poboljšali procesi testiranja, korisno je testne verzije objavlјivati sistematizirano. Nova testna verzija kreira se spajanjem svih verzija dodanih razvojnoj grani od zadnjeg kreiranja testne verzije. Sve promjene dodane u sklopu ove testne verzije potrebno je jasno specificirati. Greške koje se otkriju u sklopu testiranja otklanjaju se na testnoj grani te se pri završetku testiranja spajaju s razvojnom granom. Slika 4.4 prikazuje proces automatizirane isporuke korištenjem *gitflow* procesa.

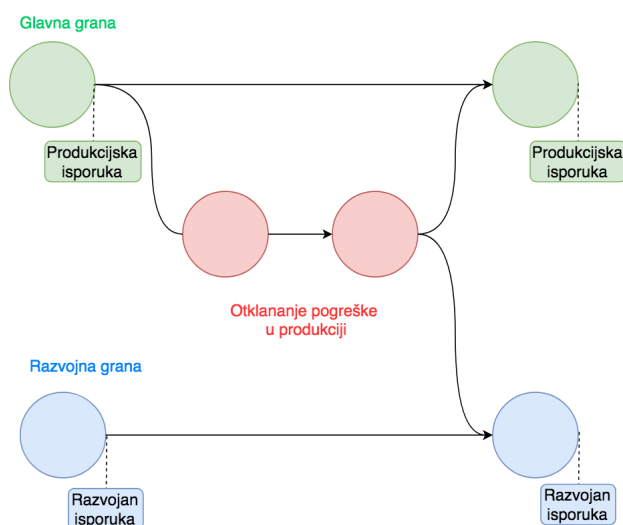
Sada je vrlo jednostavno odrediti koji tip isporuke iskoristiti za koju verziju repozi-



Slika 4.4: Automatska isporuka primjenom *gitflow* procesa

torija. Ako se izmjena nalazi na razvojnoj grani, potrebno je osvježiti razvojnu verziju aplikacije, a ako se izmjena nalazi na testnoj grani, potrebno je osvježiti testnu verziju.

U praksi se često javljaju novi, neočekivani zahtjevi. Jedan od primjera ovog tipa zahtjeva jest otkrivanje pogreške u produkcijskoj verziji koju je odmah potrebno ukloniti. Budući da razvojna grana sadrži promjene koje još nisu objavljene na produkciji, grešku nije moguće otkloniti slijedeći standardnu praksu. Međutim, moguće je granu



Slika 4.5: Automatska isporuka otklananja pogreške u produkciji

za otklanjanje pogreške kreirati direktno iz glavne grane. Nakon otklanjanja pogreške i testiranja, promjena se spaja s glavnom granom čime se ista objavljuje u produkciju i s radnom granom. Primjer otklanjanja greške u produkciji nalazi se na slici 4.5. Važno je prilikom pojave zahtjeva ovog tipa iskoristiti postojeće procese isporuke.

Skripta 4.1 prikazuje proces kreiranja potrebnih grana.

Skripta 4.1: Kreiranje potrebnih grana Gitflow radnog toka

```
git checkout -b develop
git push --set-upstream origin develop

git checkout -b release-candidate
git push --set-upstream origin release-candidate

git checkout -b test
git push --set-upstream origin test
```

4.3. Priprema projekta za automatsku isporuku

Uz različite procese isporuke, različite verzije aplikacije mogu zahtijevati i različitu konfiguraciju projekta. Na primjer, standardno je testnoj verziji aplikacije dodati funkcionalnosti koje olakšavaju testiranje, a nisu dostupne stvarnom korisniku. Dodatno, kako bi se na istom uređaju moglo imati više različitih verzija aplikacije, navedene se verzije moraju na neki način razlikovati. iOS operacijski sustav aplikacije to razlikuje na temelju identifikatora aplikacije (engl. *Bundle Identifier*).

Navedeni se tipovi funkcionalnosti implementiraju konfiguracijom postavka projekta. Svaki cilj iOS projekta sadrži najmanje jednu konfiguraciju projekta (engl. *configuration*). Konfiguracija omogućava modifikaciju velikog broja opcija definiranih u procesu kreiranja projekta. Konfiguracija je pohranjena u `.xcodproj` datoteci zajedno s ostatkom informacija o projektu koja nije pogodna za ručnu izmjenu. Datoteku je najlakše modificirati korištenjem Xcode aplikacije koja konfiguraciju prikazuje u jednostavnom i intuitivnom grafičkom sučelju. Slika 4.6 prikazuje dio opcija konfiguracije prikazanih pomoću Xcode alata.

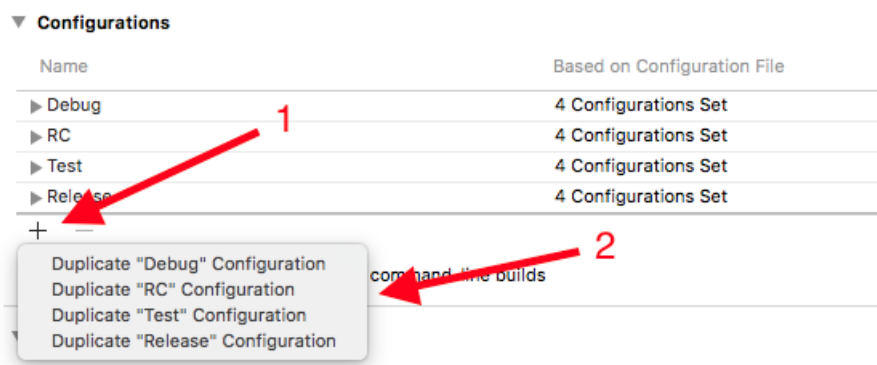
Također, projekt može sadržavati više od jedne konfiguracije. Standardno se koristi jedna glavna i više specifičnih konfiguracija. Glavna konfiguracija definira standardna pravila za cijeli projekt. Specifične konfiguracije nasljeđuju glavnu konfiguraciju, izmijenjuju željene postavke te se koriste za konfiguraciju operacija projekta.

▼ Apple LLVM 8.1 - Language - C++				
Setting	Resolved	Diplomski_rad	Config.File (Development.xcc...	Diplomski_rad
C++ Language Dialect	GNU++11 [-std=gnu++11] ↕		GNU++11 [-std=gnu++11] ↕	
C++ Standard Library	libc++ (LLVM C++ standard libr... ↕		libc++ (LLVM C++ standard li...	
Enable C++ Exceptions	Yes ↕			
Enable C++ Runtime Types	Yes ↕			
▼ Apple LLVM 8.1 - Language - Modules				
Setting	Resolved	Diplomski_rad	Config.File (Development.xcc...	Diplomski_rad
Allow Non-modular Includes In Framework Modules	No ↕			
Enable Clang Module Debugging	Yes ↕			
Enable Modules (C and Objective-C)	Yes ↕		Yes ↕	
Link Frameworks Automatically	Yes ↕			
▼ Apple LLVM 8.1 - Language - Objective C				
Setting	Resolved	Diplomski_rad	Config.File (Development.xcc...	Diplomski_rad
Enable Objective-C Exceptions	Yes ↕			
Implicitly Link Objective-C Runtime Support	Yes ↕			
Objective-C Automatic Reference Counting	Yes ↕		Yes ↕	
Weak References in Manual Retain Release	No ↕			

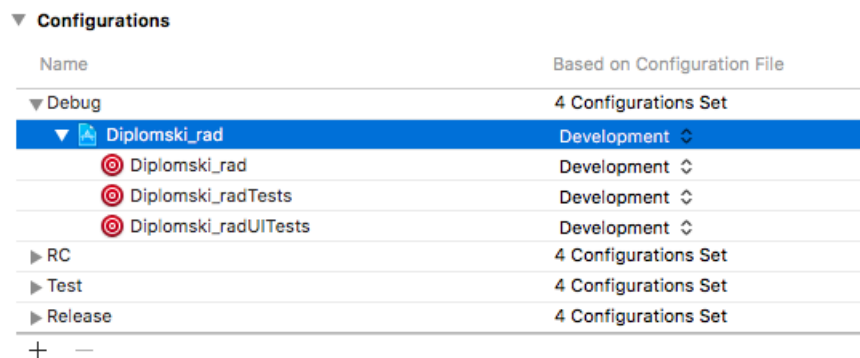
Slika 4.6: Dio opcija konfiguracije iOS projekta

Slika 4.7 prikazuje proces kreiranja specifične konfiguracije. Kreiranje konfiguracije ostvaruje se dupliciranjem postojeće konfiguracije. Potrebno je okrenuti projekt korištenjem Xcode aplikacije te odabrati željenu shemu. U sekciji Info te u odjeljku Configurations odabirom opcije Plus i željene postojeće konfiguracije treba kreirati konfiguraciju.

Modifikacija postavki pomoću specifične konfiguracije jednostavna je, ali je teško pronaći sve specificirane postavke. Isto tako, vrlo je teško pratiti promjenu postavki budući da se one dodaju `.xcodeproj` datoteci. Zbog navedenog se u praksi često koriste zasebne tekstualne `.xcconfig` datoteke koje specificiraju samo željene opcije. U sklopu rada kreiram četiri `.xcconfig` datoteke. Sadržaj datoteka prikazan je u skripti 4.2. Naredba #1 dodaje konfiguraciju CocoaPods projekta novokreiranoj konfiguraciji, dok naredba #2 postavlja vrijednost identifikatora aplikacije. Testna konfiguracija sadrži dodatnu `IS_TEST` opciju koja se koristi za dodavanje testnih funkcionalnosti.



Slika 4.7: Kreiranje nove konfiguracije Xcode projekta



Slika 4.8: Dodavanje .xcconfig datoteke postojećoj konfiguraciji projekta

Skripta 4.2: Sadržaj .xcconfig datoteke

```
#include "Pods/Target Support Files/Pods-Diplomski_rad/
    Pods-Diplomski_rad.debug.xcconfig" #1

PRODUCT_BUNDLE_IDENTIFIER = com.rep.Diplomski-rad.{sufix}
#2

IS_TEST = 1 #3
```

Kreiranje .xcconfig datoteke potrebno je dodati konfiguraciji. U sekciji Info te u odjeljku Configurations treba odabrati željenu konfiguraciju te za sve ciljeve iskoristiti istu .xcconfig datoteku. Slika 4.8 prikazuje dodavanje datoteka razvojnoj konfiguraciji.

4.4. Implementacija kontinuirane isporuke

Za dovršenje implementacije potrebno je isporuku pokrenuti korištenjem bota. Prvo je za svaki tip isporuke potrebno kreirati zaseban bot. Najjednostavnije je duplicirati već postojeći bot te promijeniti na kojoj isporuci bot pokreće integraciju. Slika 4.9 prikazuje botove korištene u radu.



Slika 4.9: Novokreirani botovi

Nakon kreiranja botova potrebno je promijeniti konfiguraciju koju botovi koriste te pozvati ispravan proces isporuke.

Konfiguraciju koju projekt koristi moguće je prilagoditi u postavkama bota. Treba odabrati opciju *Edit Bot...* -> *Build Configurations* te za svaki bot odabrati odgovarajuću konfiguraciju.

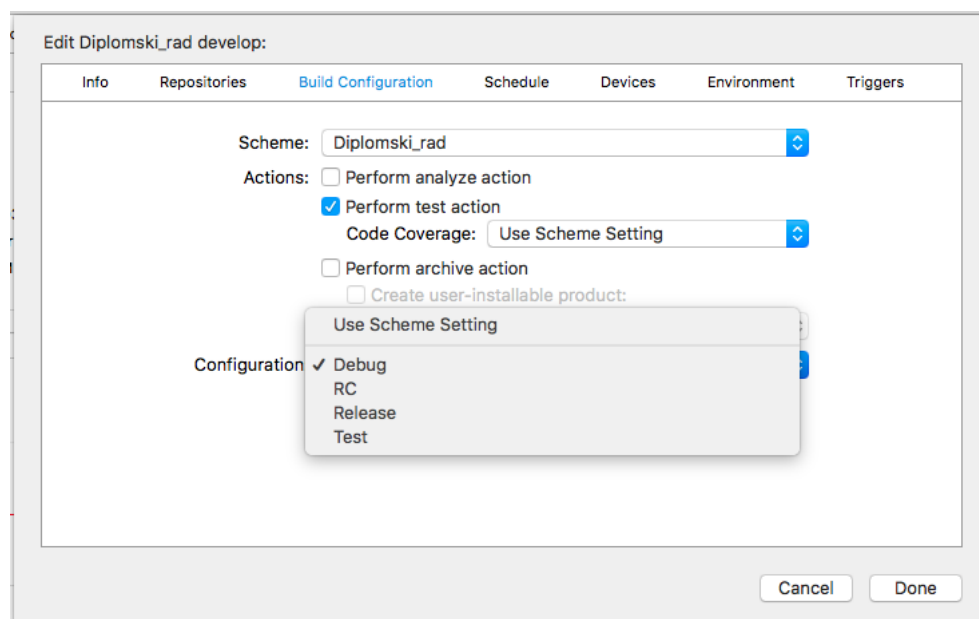
Isporuku ostvarujem korištenjem fastlane staza definiranim u prošlom poglavlju. Stazu je potrebno pokrenuti pri završetku obavljanja integracije za što koristim fazu koja se obavlja nakon integracije (engl. *Post-integration script*). Potrebno je odabrati opciju *Edit Bot...* -> *Triggers* te dodati novu fazu. Nakon toga, odabrati opciju pokretanja jedino u slučaju uspješnog obavljanja integracije. Svaki bot treba pozivati stazu koja implementira željeni tip isporuke. Skripta 4.3 prikazuje sadržaj faze.

Skripta 4.3: Sadržaj faze nakon obavljanja isporuke

```
#!/bin/bash

cd $XCS_PRIMARY_REPO_DIR #1

export PATH="$~/.fastlane/bin:$~/com.rep.Diplomski-rad.
development:/usr/local/bin:$~/gem/ruby/2.0.0/bin/:
$PATH" #2
```



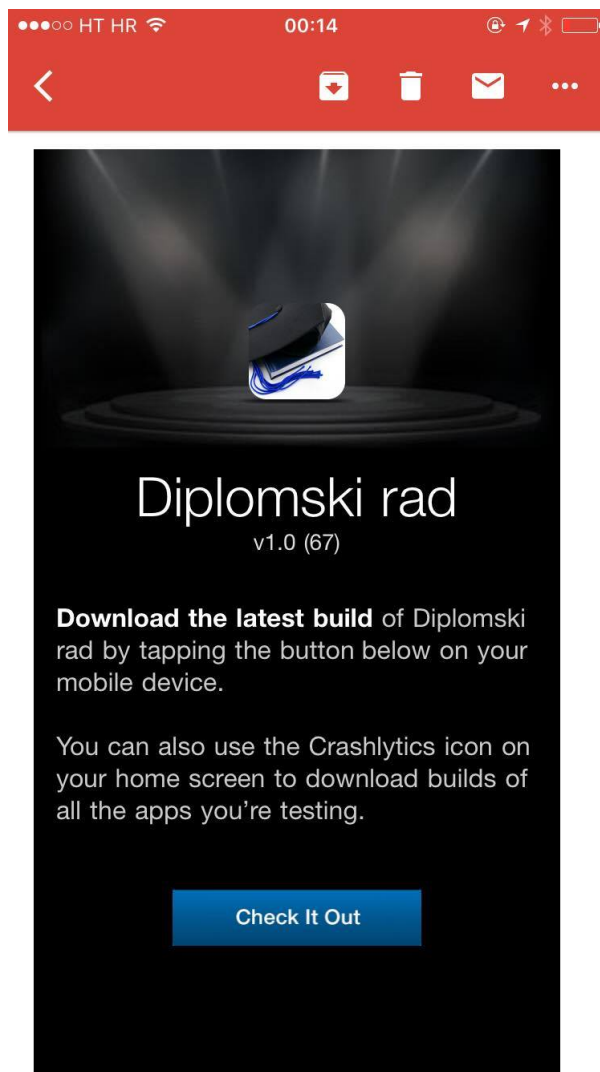
Slika 4.10: Odabir konfiguracije koju koristi bot

```
fastlane {ime_staze} #3
```

Naredba #1 navigira proces u ispravni direktorij, naredba #2 dodaje potrebne putanje u `PATH` varijablu okruženja, dok naredba #3 pokreće proces isporuke implementiran pomoću fastlane staze.

4.5. Pregled rezultata implementacije kontinuirane isporuke

Kreiranjem novog stanja repozitorija na razvojnoj, testnoj, pripremnoj i glavnoj grani pokreće se proces integracije i isporuke korištenjem navedenog stanja. Integracija se sastoji od izgradnje, testiranja i osiguranja kvalitete nove verzije aplikacije. Ako se



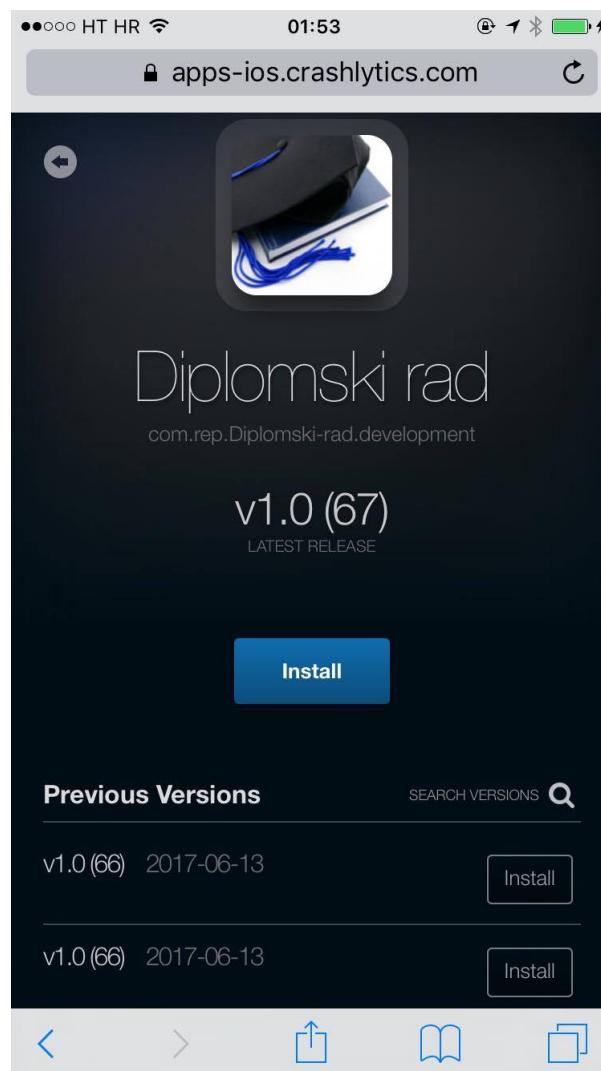
Slika 4.11: Crashlytics poruka e-pošte izdana nakon objave nove verzije aplikacije

svi procesi ispravno izvrše, započinje proces isporuke koji odgovara grani na kojoj je stanje kreirano. Glavna verzija aplikacije objavljuje se na App Store platformi, dok se ostale verzije aplikacije objavljuju na Crashlytics platformi.

Navedeni se proces odvija u potpunosti automatizirano.

Novu verziju aplikacije moguće je preuzeti s Crashlytics platforme desetak minuta nakon kreiranja novog stanja repozitorija. Svaka osoba dodana kao tester aplikacije dobiva poruku e-pošte pomoću koje može preuzeti aplikaciju. Na željenom mobilnom uređaju treba otvoriti dobivenu poruku te odabirom opcije `Let Me In` započeti proces preuzimanja nove verzije aplikacije. Slika 4.11 prikazuje dobivenu poruku e-pošte.

Instalacija aplikacija objavljenih na Crashlytics platformi obavlja se pomoću aplikacije *Beta*. Ako aplikacija već nije instalirana na mobilnom uređaju, opcija pokreće proces instalacije i konfiguracije navedene aplikacije. Slika 4.12 prikazuje izgled



Slika 4.12: Preuzimanje aplikacije s Crashlytics platforme korištenjem aplikacije Beta

ekrana aplikacije Beta s kojeg je moguće preuzeti objavljene aplikacije. Odabirom opcije `Install` pokreće se instalacija najnovije verzije aplikacije. Moguće je preuzeti i neku od starijih verzija.

S druge strane, verzija objavljena na App Store platformi prvo mora proći Apple-ovu provjeru. Ako provjera uspješno završi, aplikaciju je potrebno objaviti korištenjem iTunes Connect platforme. Navedeni proces trenutno nije moguće automatizirati. Nakon objave nova je verzija vidljiva na App Store platformi te se već instalirane aplikacije samostalno osvježavaju.

5. Zaključak

Kontinuirana je isporuka tek u svojim začecima u iOS zajednici. Iako praksa dobiva sve više na pozornosti, alati koji olakšavaju njeno ostvarenje novi su i malobrojni. Složeno je implementirati čak i najosnovnije funkcionalnosti. Potrebno je koristiti kombinaciju nekoliko alata te često koristiti proces pokušaja i pogreške. Što je gore, u stvarnom svijetu situacija nikad nije idealna. Gotovo svaki tim definira vlastiti način rada, a izuzeci se koriste čak i u najbolje organiziranim timovima.

Međutim, benefiti koje pruža kontinuirana isporuka nadilaze ove prepreke. Automatska isporuka promjena eliminira potrebu obavljanja ručne isporuke. Ovisno o timu i zahtjevima razvoja, ručna isporuka može zauzeti i do desetak posto ukupnog vremena razvoja. Utrošak je posebno visok u timovima koji koriste *agilan* pristup razvoju. Ovaj pristup učestalom isporukom i komunikacijom s klijentom nastoji osigurati izgradnju ispravnog produkta, ali upravo navedena učestala isporuka zahtijeva utrošak značajnog dijela vremena.

Nastavljanjem kontinuirane isporuke direktno na proces kontinuirane integracije ostavljam netaknutim korišteni proces razvoja programske potpore. Kontinuirana isporuka jednostavno već integriranu verziju objavljuje odgovarajućim tipom isporuke. Korištenjem Gitflow tijekom rada implementiram odabir tipa objave, ali je isto moguće ostvariti na više načina, npr. korištenjem dodatnih argumenata prilikom kreiranja verzije repozitorija.

Kontinuirana isporuka automatski izvodi isporuku implementiranu u sklopu kontinuirane dostave. U sklopu rada implementirao sam četiri tipa isporuke korištenjem fastlane alata: razvojnu, *ad hoc*, unutarnju i produkcijsku isporuku. Kontinuiranu integraciju ostvario sam korištenjem Xcode Server alata, spoja Xcode i macOS Server aplikacije.

Glavni nedostatak implementiranog sustava jest složenost njegovog ostvarenja. Čak i uz praćenje dobro definiranih uputa implementacija traje desetak sati. Glavni razlog složenosti implementacije proizlazi iz činjenice da je Xcode Server namijenjen za jednostavnu implementaciju kontinuirane za iOS operacijski sustav. Navedena jednos-

tavnost značajno otežava proširenje osnovnog seta funkcionalnosti. Osnovni problem leži u sučelju alata koje je za razliku većine sličnih alata grafičko. Zbog navedenog nije moguće automatizirati njegovu instalaciju i korištenje. Svaku izmjenu procesa potrebno je odraditi ručnom izmjenom *bota*.

Kako bi kontinuirana isporuka postala standardan dio razvoja programske potpore za iOS operacijski sustav, potrebno je olakšati proces njene implementacije.

LITERATURA

- [1] Apple. xcodebuild, 2013. URL <https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/xcodebuild.1.html>. [Online; accessed 16-March-2017].
- [2] Atlassian. Comparing workflows, 2016. URL <https://www.atlassian.com/git/tutorials/comparing-workflows>. [Online; accessed 15-March-2017].
- [3] Carl Caum. What's the diff, 2013. URL <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>. [Online; accessed 20-May-2017].
- [4] Intercom. Why continuous deployment just keeps on giving, 2015. URL <https://blog.intercom.com/why-continuous-deployment-just-keeps-on-giving/>. [Online; accessed 20-May-2017].
- [5] Dan North Jez Humble, Chris Read. The deployment production line, 2006. URL https://continuousdelivery.com/wp-content/uploads/2011/04/deployment_production_line.pdf. [Online; accessed 20-Jun-2017].
- [6] realm. Swiftlint, 2017. URL <https://github.com/realm/SwiftLint>. [Online; accessed 24-Apr-2017].
- [7] Open source. Fastlane code signing, 2017. URL <https://codesigning.guide/>. [Online; accessed 1-May-2017].
- [8] Open source. fastlane, 2017. URL <https://github.com/fastlane/fastlane/tree/master/fastlane>. [Online; accessed 29-Apr-2017].

- [9] Open source. deliver, 2017. URL <https://github.com/fastlane/fastlane/tree/master/deliver>. [Online; accessed 3-May-2017].
- [10] Open source. gym, 2017. URL <https://github.com/fastlane/fastlane/tree/master/gym>. [Online; accessed 1-May-2017].
- [11] Open source. match, 2017. URL <https://github.com/fastlane/fastlane/tree/master/match>. [Online; accessed 1-May-2017].
- [12] Open source. scan, 2017. URL <https://github.com/fastlane/fastlane/tree/master/scan>. [Online; accessed 1-May-2017].
- [13] Open source. Homebrew, 2017. URL <https://brew.sh/>. [Online; accessed 11-Jun-2017].
- [14] Wikipedia. Booch method — Wikipedia, the free encyclopedia, 2015. URL https://en.wikipedia.org/wiki/Booch_method. [Online; accessed 10-Feb-2017].
- [15] Wikipedia. Code coverage — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Code_coverage. [Online; accessed 23-Apr-2017].
- [16] Wikipedia. Continuous delivery — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Continuous_delivery. [Online; accessed 28-Apr-2017].
- [17] Wikipedia. Hudson, 2017. URL [https://en.wikipedia.org/wiki/Hudson_\(software\)](https://en.wikipedia.org/wiki/Hudson_(software)). [Online; accessed 22-Jun-2017].
- [18] Wikipedia. Jenkins, 2017. URL [https://en.wikipedia.org/wiki/Jenkins_\(software\)](https://en.wikipedia.org/wiki/Jenkins_(software)). [Online; accessed 22-Jun-2017].
- [19] Wikipedia. Software testing — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Software_testing. [Online; accessed 23-Apr-2017].
- [20] Wikipedia. Software versioning — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Software_versioning. [Online; accessed 18-Feb-2017].

- [21] Wikipedia. Version control — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Version_control. [Online; accessed 18-Feb-2017].

Implementacija kontinuirane isporuke programske podrške za operacijski sustav iOS

Sažetak

Kontinuirana isporuka je praksa u programskom inženjerstvu koja automatskim obavljanjem isporuke programske potpore nastoji povećati njenu učestalost i time povećati kvalitetu razvijenog produkta. Kontinuirana isporuka direktno se nastavlja na kontinuiranu dostavu, praksu koja automatiziranjem procesa isporuke nastoji olakšati njegovo obavljanje. Praksa nastoji osigurati mogućnost isporuke programske potpore u bilo kojem trenutku uz minimalnu količinu rada. Obje su prakse razvijene na temelju te se direktno nastavljaju na kontinuiranu integraciju, praksu automatskog obavljanja integracije nad svakom promjenom izvornog koda. Ove tri prakse nastoje olakšati timski razvoj, poboljšati kvalitetu koda i produkta te ubrzanjem i automatizacijom isporuke ubrzati objavu novih funkcionalnosti.

Rad definira kontinuiranu integraciju, dostavu i isporuku iz perspektive razvoja za iOS operacijski sustav, definira zahtjeve koji se javljaju u praksi te implementira sustav koji ih zadovoljava. Kako bi sustav opravdao korištenje, mora biti prenosiv, jednostavan za implementaciju i svakodnevno korištenje. Za implementaciju sustava koristim aplikaciju Xcode Server i alat fastlane.

Sustav obavlja integraciju i isporuku svake nove verzije repozitorija čime eliminiрам potrebu obavljanja ručne isporuke. Dodatno, automatizacijom procesa potičem češće i redovito obavljanje isporuke.

Ključne riječi: Ključne riječi, odvojene zarezima.

Kontinuirana isporuka, kontinuirana dostava, kontinuirana izgradnja, Xcode Server, fastlane

Implementation of Continuous Deployment for iOS Operating System

Abstract

Continuous Deployment is a software engineering approach which by automatic software deployment seeks to increase the frequency and quality of the deployed product. Continuous Delivery is a direct extension of Continuous Delivery, an approach that tries to automate the whole process of product delivery in order to reduce its complexity. Additionally, Continuous Delivery ensures that the software can be reliably released at any time. Both approaches were developed on top of Continuous Integration which automatically performs integration process for every new repository state. These three practices aim to simplify teamwork, improve the quality of code and product, and shorten the process of delivery.

This thesis defines Continuous Integration, Continuous Delivery and Continuous Deployment from the perspective of iOS development, defines requirements which occur in the real world and to implement a system that satisfies them. In order for a system to be usable, it needs to be easily transferable and simple to implement and use. The system is implemented using Xcode Server application and fastlane tool.

The system performs automated integration and delivery of every new repository version thus eliminating the need to perform manual delivery. Additionally, process automation encourages frequent and regular software delivery.

Keywords: Keywords.

Continuous Deployment, Continuous Delivery, Continuous Integration, Xcode Server, fastlane

Dodaci

A. Usporedba alata za implementaciju kontinuirane integracije

Kontinuirana integracija ustaljena je i dobro prihvaćena praksa u sklopu razvoja programske potpore. Jedan od glavnih razloga njezine prihvaćenosti je postojanje velikog broja alata koji značajno olakšavaju njenu implementaciju.

Međutim, navedeni alati međusobno se značajno razlikuju. Dok neki alati nastoje podržati što veći broj funkcionalnosti za što više tehnologija, drugi se fokusiraju na pojedinu instancu razvojnog procesa. Neki se sustavi ističu po svojoj jednostavnosti, dok drugi pružaju veću fleksibilnost.

Cilj ovog dodatka jest korištenjem i ocjenom četiri trenutno najpopularnija alata ustanoviti koji najbolje odgovara zahtjevima kontinuirane integracije za iOS operacijski sustav. U sklopu dodatka rankiram četiri alata: Jenkins, Travis CI, CircleCI i Xcode Server.

Uz ovoliku količinu gotovih rješenja teško je naći argument za razvoj vlastitog rješenja. Gotovo sve implementacije kontinuirane integracije koriste neki od postojećih alata. Čak i kada tim počne razvijati vlastito rješenje, najčešće modificira ili proširuje već postojeće alate otvorenog koda.

Međutim, teško je iz navedene količine dostupnih sustava izabrati onaj koji najviše odgovara zahtjevima tima, a da u isto vrijeme zahtijeva najmanje truda za implementaciju i korištenje. Javno dostupni dokumenti ne pomažu značajno po tom pitanju. S ciljem boljeg razumijevanja danog problema i jednostavnije procjene karakteristika sustava kontinuirane integracije provodim ispitivanje sustava korištenjem dva jednostavna upitnika. Rezultati upitnika isključivo su moja procjena te je njihova glavna svrha bolja organizacija i lakša procjena karakteristika sustava.

Oba se upitnika zajedno s ocjenama sustava nalaze na kraju dodatka.

A.1. Prvi upitnik

Prvi upitnik procjenjuje koliko dobro sustav zadovoljava funkcionalnosti opisane u prethodnom poglavlju. Kako bi se lakše provelo ocjenjivanje, a rezultati bili bolje razumljiviji, pitanja su podijeljena u pet kategorija.

Prva kategorija, *opći zahtjevi*, specificira koliko je lako sustav uklopiti u postojeći proces razvoja programske potpore. U ovo se poglavlje ubraja mogućnost integracije sustava sa sustavima za kontrolu verzija i mogućnost integracije u postojeći proces razvoja. Poželjno je sustav kontinuirane integracije ostvariti ne mijenjajući postojeći proces razvoja.

Druga kategorija, *provjera ispravnosti*, ispituje podržava li sustav sve zahtjeve automatizacije provjere ispravnosti integracije. Ovi se zahtjevi sastoje od automatiziranja izgradnje, testiranja i osiguranja kvalitete. Također, kategorija ocjenjuje konfigurabilnost testne okoline i mogućnost testiranja na stvarnim uređajima.

Treća kategorije, *izvršavanje procesa*, provjerava koliku slobodu sustav pruža pri definiranju i obavljanju procesa. Kategorija promatra kada je moguće pokrenuti procese te pruža li sustav napredne mogućnosti ulančavanja procesa i stvaranje nestandardnih procesa.

Četvrta kategorija, *rezultati procesa*, prikazuje u kojem stupnju sustav podržava automatsko izvještavanje te koliko je dostupan i detaljan pregled rezultata procesa.

Peta kategorija, *kontinuirana dostava i isporuka*, daje uvid u stupanj u kojem sustav podržava željene funkcionalnosti kontinuirane dostave i isporuke.

Svaka se točka upitnika ocjenjuje brojčanom ocjenom od 0 do 5. Ocjena 0 znači da sustav ne podržava danu funkcionalnost, dok ocjena 5 znači da sustav podržava danu funkcionalnost bez potrebne dodatne modifikacije (engl. *off the shelf*). Ocjene od 1 do 4 označavaju da je funkcionalnost podržavana, ali je potrebna različita količina truda za njezinu implementaciju. Ocjena 1 označava da je funkcionalnost jako teško implementirati, dok ocjena 4 označava da je funkcionalnost jednostavno implementirati.

Upitnik ne daje definitivnu ocjenu sustava, već služi za generalnu navigaciju po širokom tržištu alata kontinuirane integracije. Uz ukupan rezultat pojedinog sustava, korisno je promatrati i rezultate pojedine kategorije.

A.2. Drugi upitnik

Drugi se upitnik fokusira na jednostavnost korištenja, implementacije i održavanja sustava iz perspektive razvoja programske podrške za mobilne operacijske sustave. Cilj

ovog ispitivanja jest procijeniti koliko je složeno implementirati odabrani sustav kontinuirane integracije za postojeće procese razvoja programske potpore za mobilne operacijske sustave.

Na tržištu trenutno dominiraju tri mobilna operacijska sustava: Android, iOS i Windows Phone. U sklopu ovog rada preskačem Windows Phone zbog neizvjesnosti nastavka njegove primjene.

Kod razvoja programske potpore za iOS i Android prepoznavamo nekoliko različitih procesa razvoja i korištenih tehnologija. Najkorišteniji su procesi vezani uz odvojeni razvoj korištenjem službenih alata i tehnologija. Za iOS to je Xcode IDE s Objective-C ili Swift programskim jezicima, dok je za Android to Android Studio IDE i programski jezik Java. Alternativna opcija razvoja aplikacija za mobilne platforme jest višepplatformski razvoj (engl. *cross-platform*) koji omogućuje potpunu ili djelomičnu ponovnu iskoristivost koda na više platformi. Prvi je pristup puno popularniji te pruža značajno šire mogućnosti razvoja. Zbog toga se u sklopu ovog rada fokusiram prvenstveno na odvojeni pristup razvoju. Međutim, zaključci do kojih se dolazi u ovom poglavlju s manjim se ograničenjem mogu preslikati na višepplatformski razvoj.

Poželjno je isti sustav kontinuirane integracije iskoristiti za oba razvojna procesa. Međutim, kako su određeni sustavi specijalizirani za određeni proces, oni istovremeno mogu biti vrlo dobri za jedan, a vrlo loši za drugi razvojni proces.

Upitnik se sastoji od sljedećih pitanja:

1. jednostavnost korištenja
2. jednostavnost ugradnje
3. jednostavnost konfiguracije
4. jednostavnost održavanja

pri čemu se svaka stavka ocjenjuje ocjenom od 1 do 5, gdje 1 znači vrlo složeno a 5 vrlo jednostavno. Poredak sugerira važnost pojedine stavke na temelju prethodnog iskustva.

A.3. Ispitani sustavi

Sustave za kontinuiranu integraciju generalno možemo podijeliti u dvije grupe. *SaaS*, *Software as a Service* alati hostani su od strane davatelja usluge. Navedeni alati zahtijevaju minimalnu konfiguraciju, dostupni su prema potrebi korisnika te su jednostavni za korištenje. Međutim, oni uglavnom pružaju predefimirani i nefleksibilan set funkcionalnosti. Dostupni su uglavnom kao aplikacije za web-poslužitelje koje dodatne

mogućnosti omogućuju pružanjem fleksibilnog programskog sučelja (engl. *Application Interface, API*). Drugi tip alata čine samostalno hostani alati koje korisnik podiže na vlastitom računalu. Ovi su tipovi sustava fleksibilniji te korisniku pružaju napredne funkcionalnosti, ali zahtijevaju veći napor za postavljanje, održavanje i korištenje.

Kroz ovu cjelinu detaljno promatram po dva sustava iz pojedine kategorije. Oda-brani sustavi ubrajaju se među najpoznatije i najkorištenije sustave kontinuirane integracije. Smatram da ovaj mali skup pokriva većinu zahtjeva koje sam definirao u drugom poglavlju ovog rada, odnosno, većinu stvarnih potreba razvoja programske potpore za mobilne sustave.

A.3.1. Jenkins

Jenkins je najpopularniji samostalno hostani CI sustav. Golemu popularnost zaslužio je zbog velike fleksibilnosti i kompatibilnosti s gotovo svim alatima korištenim u razvoju programske potpore. Jenkins je moguće koristiti kao generički sustav kontinuirane integracije ili ga pretvoriti u složeni sustav koji podržava specifične zahtjeve pojedinog tima. Navedena je fleksibilnost ostvarena *plugin arhitekturom*. Pluginovi, koji se jednostavno razvijaju i spajaju s osnovnim sustavom, omogućuju integraciju Jenkinsa s bilo kojom tehnologijom i razvojnim procesom. Danas postoji stotine pluginova koji omogućavaju jednostavnu integraciju Jenkinsa s gotovo svim IDEovima, sustavima za kontrolu verzija i bazama podataka[18].

Jenkins je sustav otvorenog koda, originalno razvijen u kompaniji *Sun Microsystems* pod imenom *Hudson*[17] 2004. godine. Ubrzo je stekao popularnost kao bolji od tada dostupnih poslužitelja za izgradnju programske podrške (engl. *build server*). Između Hudsonove zajednice i vlasnika kompanije *Oracle* 2010. se godine pojavio mali problem oko korištene infrastrukture. Problem je prerastao u široku raspravu oko vlasništva i kontrole nad sustavom. Iako su se zajednica i Oracle slagali u velikom broju pitanja, vlasništvo nad imenom “Hudson” postao je nepremostivi problem. Zajednica je u siječnju 2011. objavila poziv na glasanje za izmjenu imena projekta iz “Hudson” u “Jenkins”. Izmjena je usvojena velikom većinom glasova te je osnovan projekt Jenkins. Oracle nastavlja razvoj Hudsona odvojeno od Jenkinsa, međutim, bez podrške zajednice Hudson značajno zaostaje u razvoju i popularnosti.

Jenkins je odličan ako je timu potrebna velika razina kontrole nad procesom automatizacije. Ova razina kontrole može donijeti značajnu korist timu i u cijelosti olakšati te poboljšati proces razvoja programske potpore. Naravno, visoka razina kontrole otežava postavljanje, održavanje i korištenje sustava.

Jenkins je na prvom ocjenjivanju ostvario 44/60 bodova. Najlošije ostvaruje 5. kategoriju zbog nepostojanja jednostavnog načina podjele korisnika i automatske isporuke. Ovo je očekivano s obzirom na to da je Jenkins jedan od najfleksibilnijih alata trenutno dostupnih na tržištu, međutim zahtijeva puno programiranja te je ga je složeno ovladati.

Na drugom ocjenjivanju sustav je ocijenjen puno lošije. Jenkins je skupio 11/20 bodova iz perspektive mobilnih sustava. Sustav je jednostavan za korištenje, međutim vrlo je složen za ugradnju i konfiguriranje. Potrebno je uložiti barem 20 sati kako bi programer postao upoznat s osnovama sustava i njegovog korištenja. Svejedno, postoje vrlo dobri pluginovi koji podržavaju razvoj aplikacija za iOS u Xcodeu i aplikacija za Android u Android Studiju koji ovaj problem djelomično otklanjaju.

Jenkins je jedno od najboljih rješenja za kontinuiranu integraciju ako je tim spreman uložiti nekoliko mjeseci rada za ostvarivanje sustava kontinuirane integracije. Mogućnosti su gotovo beskonačne, a velika zajednica koja ga okružuje čini implementaciju puno lakšom.

A.3.2. Xcode Server

Xcode Server omogućava vrlo jednostavnu kontinuiranu integraciju za korisnike programskog alata Xcode, odnosno za razvoj aplikacija za iOS i OS X operacijske sustave. Xcode Server kombinacija je dvaju alata: Xcodea, alata za razvoj iOS i OS X aplikacija, te OS X Servera, alata za automatizaciju razvojnog procesa. Xcode Server omogućava automatiziranu integraciju, izgradnju, testiranje, analizu i arhiviranje aplikacija. Također omogućava automatsku dojavu rezultata i generiranje statistika za velik broj parametara.

Xcode Server, Appleov alat za kontinuiranu integraciju, po mnogima je najbolji sistem za kontinuiranu integraciju iOS i OS X aplikacija. Xcode Server se razvija kontinuirano uz Xcode. Za razliku od ostalih alata, sve su nove funkcionalnosti potrebne za integraciju Xcode projekata dostupne na dan njihovog izdavanja. Ova je funkcionalnost vrlo korisna pri ranom testiranju novih verzija sustava te se pokazala vrlo korisnom u mnogim projektima.

On je, kao i Jenkins, samostalno hostan i fleksibilan sustav. Funkcionalnosti se implementiraju izgradnjom *botova* koji zatim pokreću zadani proces. Procesi su implementirani korištenjem jezika korištenog u razvoju iOS i OS X aplikacija. Zbog korištenja istog alata i istog jezika kao i za razvoj iOS aplikacija, implementiranje kontinuirane integracije korištenjem Xcode Servera vrlo je jednostavno programerima

upoznatima s razvojem iOS aplikacija. Navedeno značajno olakšava implementaciju koja tada zahtijeva značajno manje od implementacije Jenkins sustava.

Dodatna prednost Xcode Servera jest mogućnost automatske instalacije aplikacije na testne uređaje te lako praćenje i nadzor instaliranih testnih aplikacija. To omogućava automatizaciju testne distribucije i značajno olakšava proces testiranja. Međutim, navedene funkcionalnosti ne podržavaju sve funkcionalnosti kontinuirane dostave zbog čega je ovaj dio potrebno implementirati odvojeno.

Na kraju, laka integracija s postojećim Appleovim sustavima, kao što su sustavi za certificiranje i arhiviranje čine Xcode Server najboljim alatom za ostvarivanje kontinuirane integracije za razvoj programske potpore za iOS operacijski sustav.

Xcode Server na testiranju funkcionalnosti ostvario je 52/60 bodova za razvoj iz perspektive iOS operacijskog sustava. Xcode Server omogućava automatizaciju svih procesa korištenih prilikom razvoja u Xcodeu. Xcode Server, kao i Jenkins, nema jaku podršku za kontinuiranu isporuku. Moguće je implementirati obilježavanje funkcionalnosti, ali funkcionalnosti je teško razdvojiti u produkciji.

Xcode Server dobre rezultate ostvaruje i u ocjeni jednostavnosti gdje ostvaruje 17/20 bodova, naravno, iz perspektive razvoja za iOS operacijski sustav. Korištenje, ugradnja i konfiguracija jednostavni su, dok održavanje ne zahtijeva puno truda. Jedini potencijalni minus jest ograničenost alata na OS X operacijski sustav, ali ovo je standardno kod razvoja iOS aplikacija zbog čega navedeno ne uzimam kao manu.

Iako je korištenjem univerzalnog alata, na primjer Jenkinsa, moguće iskoristiti jedan alat za implementaciju kontinuirane integracije za više razvojnih procesa, još je uvijek svaki proces potrebno odvojeno implementirati. Zbog navedenog nije moguće značajno uštedjeti na vremenu potrebnim za implementaciju kontinuirane integracije. Glavnina uštede dolazi iz potrebe svladavanja samo jednog sustava.

A.3.3. CircleCI

CircleCI jedan je od najpopularnijih SaaS alata koji olakšavaju ostvarenje kontinuirane integracije. CircleCI se ističe svojom jednostavnošću, intuitivnim dizajnom i podrškom velikog broja popularnih razvojnih procesa.

CircleCI jednostavno se dodaje na postojeći repozitorij izvornog koda te samostalno konfigurira početno okruženje. Iz postojećeg koda otkriva koji se jezik i alati koriste iz čega zatim provodi konfiguraciju okruženja. Veliki broj podržanih alata koje je vrlo jednostavno uključiti u svoje okruženje dodatno olakšava ugradnju CircleCI alata u postojeći proces razvoja.

Alat se ističe kao jedan od najjednostavnijih alata za ostvarenje kontinuirane integracije na tržištu koji istovremeno podržava razvoj za sve mobilne platforme. Uz jednostavnost, alat pruža veliki broj funkcionalnosti koje je jednostavno ostvariti i koristiti. Njegova je glavna mana, s druge strane, nefleksibilnost, odnosno teško proširenje osnovnog seta funkcionalnosti. Ovo sprječava potpunu implementaciju kontinuirane isporuke i podjelu korisnika.

CircleCI je ostvario 40/60 bodova u ocjenjivanju funkcionalnosti sustava. Velika jednostavnost sustava donekle ograničava njegove mogućnosti. Nastavljaju se problemi sustava s kontinuiranom dostavom i isporukom, posebno s podjelom korisnika i obilježavanju funkcionalnosti. CircleCI ne omogućava brojne dodatne funkcionalnosti provjere ispravnosti i složene funkcionalnosti procesa gdje gubi dio bodova.

S druge strane, sustav je vrlo jednostavno za koristiti. On ostvaruje 18/20 bodova u ocjenjivanju jednostavnosti sustava.

CircleCI svoju popularnost može zahvaliti jednostavnosti korištenja i instalacije. Navedena jednostavnost djelomično opravdava nedostatak brojnih funkcionalnosti.

A.3.4. Travis CI

Travis CI je jedan od najpopularnijih CI sustava otvorenog koda. Dostupan je kao samostalno hostani i SaaS proizvod, ali je SaaS verzija u zadnje vrijeme puno popularnija. Sustav se jednostavno spaja na postojeći repozitorij koda te omogućava većinu funkcionalnosti navedenih u drugom poglavlju. Alat je besplatan za repozitorije otvorenog koda te pruža prihvatljive cijene za privatne repozitorije.

Pokretanje i konfiguracija sustava vrlo je jednostavna. Moguće je provoditi automatsko testiranje ispravnosti korištenjem standardnih metoda testiranja. Ujedno, testiranje je moguće provoditi nad spajanjem te nad pokušajem spajanja (engl. *pull request*).

Automatska isporuka djelomično je moguća. Sustav omogućava prebacivanje programske podrške u produkciju, ali ne podržava označavanje funkcionalnosti.

Alat ostvaruje očekivane rezultate na prvom upitniku. Od 60 bodova Travis CI ostvaruje 41 bod. Najlošije ostvaruje kontinuiranu dostavu i isporuku. S druge strane ostvaruje 20/20 bodova na ocjeni jednostavnosti.

Ocjena funkcionalnosti sustava kontinuirane integracije		Travis CI	CircleCI	Xcode Server	Jenkins
		Ocjena	Ocjena	Ocjena	Ocjena
1	Opći zahtjevi	9 / 10	8 / 10	10 / 10	09 / 10
1.1.	integracija sa sustavima za kontrolu verzija, minimalno Git i Subversion	5	4	5	5
1.2.	integracija u postojeći proces razvoja	4	4	5	4
2	Provjera ispravnosti	10 / 15	9 / 15	15 / 15	9 / 15
2.1.	automatizacija izgradnje	4	4	5	4
2.2.	automatizacija testiranja i napredno testiranje	4	3	5	3
2.3.	konfigurabilnost testne okoline i testiranje na stvarnim uređajima	2	2	5	2
3	Izvršavanje procesa	6 / 10	7 / 10	9 / 10	9 / 10
3.1.	pokretanje procesa u proizvoljnom trenutku	4	4	4	5
3.2.	napredne mogućnosti procesa i ulančavanje procesa	2	3	5	4
4	Rezultati procesa	9 / 10	9 / 10	8 / 10	9 / 10
4.1.	automatsko izvještavanje	4	4	4	5
4.2.	pregled rezultata	5	5	4	4
5	Kontinuirana dostava i isporuka	7 / 15	7 / 15	10 / 15	8 / 15
5.1.	automatizacija isporuke	4	3	4	4
5.2.	podjela korisnika	1	2	4	2
5.3.	obilježavanje funkcionalnosti	2	2	2	2
Ukupno		41 / 60	40 / 60	52 / 60	44 / 60

Ocjena jednostavnosti sustava kontinuirane integracije

		Travis CI	CircleCI	Xcode Server	Jenkins
1	jednostavnost korištenja	5	4	4	4
2	jednostavnost ugradnje	5	4	4	2
3	jednostavnost konfiguracije	5	5	4	2
4	jednostavnost održavanja	5	5	5	3
Ukupno		20 / 20	18 / 20	17 / 20	11 / 20

Opis ocjena

Ocjene funkcionalnosti:

- 0 - ne podržava
- 1 - vrlo složena implementacija
- 2 - složena implementacija
- 3 - srednje složena implementacija
- 4 - jednostavna implementacija
- 5 - nativno podržano

Ocjene jednostavnosti:

- 1 - vrlo složeno
- 2 - složeno
- 3 - srednje složeno
- 4 - jednostavno
- 5 - nativno podržano

B. Alat xcodebuild

Alat xcodebuild implementira izgradnju, testiranje i arhiviranje programske potpore za iOS operacijski sustav. U pozadini ga koristi veliki broj aplikacija i alata: od Xcodea i Xcode Servera, do Jenkinsa i Travis CI platforme. Iako u procesu automatizacije ne koristim alat direktno, korisno je znati što aplikacije izvršavaju u pozadini. Dodatno, alat pruža naredbeno sučelje te ga je moguće direktno koristiti.

Xcodebuild razvila je kompanija Apple krajem za potrebe izgradnje programske potpore za macOS operacijski sustav. U međuvremenu alat je proširen te danas podržava razvoj programske potpore za iOS, tvOS i watchOS operacijske sustave.

Alat je jednostavan za uporabu, ali pruža vrlo velik broj opcija koje je moguće konfigurirati. Naredba kao argument prima operaciju koja se pokreće. Ako operacija nije specificirana, xcodebuild naredba predodređeno pokreće izgradnju (engl. *build*). Ostale podržane operacije su:

`analyze` - Izgrađuje i analizira cilj ili shemu

`archive` - Arhivira i priprema projekt za objavu

`test` - Izgrađuje i testira shemu

`installsrc` - Kopira izvorni kod u SRCROOT

`install` - Izgrađuje i instalira projekt u ciljni direktorij projekta DSTROOT

`clean` - Briše metapodatke i rezultate izgradnje

Projekt ili okruženje odabire se korištenjem `-project` odnosno `-workspace` argumenata. Ako se ne specificira jedan od argumenata, naredba pokreće projekt samo ako je on jedini projekt u repozitoriju. Za odabir projekta između više opcija ili za odabir okruženja potrebno je koristiti argumente.

Skripta B.1: Odabir projekta i okruženja za obavljanje operacije

```
xcodebuild -project {ime_projekta}
```

```
xcodebuild -workspace {ime_okruzenja}
```

Dodatno, moguće je odabrati shemu projekta korištenjem `-scheme` argumenta.

Operacija koristi cilj koji specificira shema. Cilj je moguće izmijeniti korištenjem argumenta `-target`.

Skripta B.2: Odabir sheme projekta

```
xcodebuild [-project {ime_projekta}] -scheme {ime_sheme}
```

Ispis `xcodebuild` alata vrlo je detaljan. Operacija ispisuje sve postupke koje obavlja te daje detaljno izvješće u slučaju pogreške. Međutim, ovaj je tip ispisa teško čitljiv. Zbog navedenog često se koriste alati koji parsiraju i prikazuju ispis u lakše čitljivom formatu.

B.1. Testiranje

Xcode projekt implementira dvije vrste testova: *Unit* i *UI* testove. Oba su tipa testa implementirani kao ciljevi unutar projekta koji referenciraju cilj koji testiraju. Unit testovi služe za testiranje unutarnje implementacije projekta. Ovaj tip testa pokreće se kao omotač oko izvorne aplikacije te pristupa njenim resursima. UI test omogućava testiranje ponašanja aplikacije u stvarnom svijetu. Navedeni tip testa simulira korisničku interakciju te provjerava ponašanje aplikacije.

Oba tipa testa pokreću se na iOS simulatoru. Zbog toga je potrebno imati barem jedan simulator prihvatljive verzije operacijskog sustava. Nove simulatore moguće je instalirati pomoću Xcode alata. Za prikaz dostupnih simulatora pokrenuti naredbu u skripti B.3.

Skripta B.3: Ispis dostupnih simulatora

```
instruments -s devices
```

Naredba koja pokreće proces testiranja prikazana je u skripti B.4.

Skripta B.4: Pokretanje testne operacije korištenjem `xcodebuild` alata

```
xcodebuild test -workspace Diplomski_rad.xcworkspace -  
scheme Diplomski_rad -destination 'platform=iOS  
Simulator, OS=10.3, name=iPhone 7'
```

Naredba će pokrenuti testni cilj odabrane sheme na odabranom radnom okruženju. Odabir drugog projekta, cilja i sheme radi se jednako kao i kod izgradnje. Za pokretanje drugog testnog cilja potrebno je kreirati novu shemu te joj kao cilj testne operacije postaviti željeni testni cilj.

B.2. Osiguranje kvalitete

U sklopu osiguranja kvalitete provodim dva procesa: provjeru pokrivenosti koda testovima i statičku provjeru koda alatom *Swiftlint*.

Provjeru pokrivenosti koda dobivamo koristeći parametar `-showBuildSettings` pri izgradnji i testiranju projekta. Naredba je prikazana u skripti B.5

Skripta B.5: Prikupljanje podataka o pokrivenosti koda tekstovima

```
xcodebuild -workspace Diplomski_rad.xcworkspace -scheme  
Diplomski_rad -showBuildSettings
```

Naredba podatke o pokrivenosti koda sprema u `~/Library/Developer/Xcode/DerivedData/{ime_projekta}/Build/Intermediates/CodeCoverage` direktoriju. Generirani dokumenti teško su čitljivi. Postoji nekoliko alata koji ih obrađuju i generiraju čitljive rezultati. Budući da u razvoju koristim Xcode, neću u njih dublje ulaziti.

Swiftlint je alat za statičku analizu koda napisanog u programskom jeziku Swift. Alat definira veliki broj pravila kojim nastoji osigurati praćenje stila i konvencija jezika Swift. Većina pravila se odnosi na izgled i format koda, ali postoje i pravila koja nastoje izbjeći pojavu grešaka.

Alat se pokreće pozivanjem naredbe `swiftlint` u početnom direktoriju projekta. Ispis alata sličan je onome `xcodebuild` alata. Za lakše praćenje pogrešaka i pokretanje naredbe kod svakog procesa izgradnje moguće je Xcode projektu dodati novu `Run Script` fazu sa sadržajem skripte B.6.

Skripta B.6: Pokretanje provjere koda korištenjem alata Swiftlint

```
if which swiftlint >/dev/null; then  
    swiftlint  
else  
    echo "warning: Swiftlint nije instaliran"  
fi
```

B.3. Arhiviranje

Arhiviranje je proces kreiranja arhive i manifesta pomoću kojih je moguće aplikaciju instalirati na mobilni uređaj. Arhiviranje se provodi specificiranjem operacije `archive`.

Skripta B.7: Pokretanje operacije arhiviranja korištenjem xcodebuild alata

```
xcodebuild -scheme {ime_sheme} archive
```

Naredba generira dvije datoteke. Arhivu aplikacije s `.ipa` nastavkom i manifest aplikacije s `.plist` nastavkom.

C. Fastlane

Fastlane je alat za automatizaciju isporuke programske potpore za iOS i Android operacijske sustave. Alat je danas dio Fabric platforme koju je u siječnju 2017. godine preuzela kompanija Google. Fastlane je kolekcija manjih alata od kojih je svaki zadužen za automatizaciju pojedine operacije u sklopu isporuke. Ovi se alati nazivaju dodaci[8].

Alat je otvorenog koda te je veliki broj dodataka razvila zajednica. Što je više, kontinuirano se razvijaju novi dodaci. Dodatke je jednostavno povezati i time implementirati željeni proces. Osim procesa isporuke, fastlane danas podržava i automatizaciju ostalih operacija kao što su izgradnja i testiranje. Fastlane u sklopu rada koristim za automatizaciju isporuke što je prikazano u 3.1.1 odjeljku. Zbog toga navedene operacije ne ponavljam u ovom dodatku.

Alat implementira gotovo sve funkcionalnosti potrebne za automatizaciju izgradnje, testiranja i isporuku te je vrlo jednostavan za korištenje. Međutim, alat ne omogućava njihovo automatizirano pokretanje. Zbog navedenog alat se pokreće ručno ili u sklopu drugog automatiziranog procesa.

Instalacija fastlanea obavlja se korištenjem Homebrew alata. Naredba je prikazana skripti C.1.

Skripta C.1: Dohvat i instalacija fastlane alata

```
brew cask install fastlane
```

Inicijalizacija fastlane alata obavlja se naredbom `fastlane init` u početnom direktoriju projekta. Naredba kreira novi direktorij imena *fastlane* te unutar njega stvara dvije tekstualne datoteke: *Fastfile* i *Appfile*.

Fastfile datoteka olakšava uporabu fastlane alata. Unutar datoteke specificiraju se staze (engl. *lane*). Svaka je staza sastavljena od proizvoljnog broja naredbi koje implementiraju željeni proces. Također, moguće je definirati naredbe koje se izvršavaju prije ili poslije izvršenja staza, pa čak i definirati uvjete u kojima se izvršavaju. Prilikom inicijalizacije alat fastlane detektira postavke projekta te na temelju njih stvara nekoliko

predodređenih staza. Na primjer, ako projekt sadrži testni cilj, onda fastlane u *Fastfile* dokumentu kreira testnu stazu. Ako projekt sadrži *Carthage* ili *Podfile* datoteku, onda fastlane dodaje pozive za dohvat ovisnosti korištenjem dodataka carthage, odnosno cocoapods. Primjer staze prikazan je u skripti C.2.

Skripta C.2: Primjer fastlane staze

```
lane :{imestaze} do
  {naredbe}
end
```

Pokretanje staze obavlja se pozivanjem naredbe `fastlane {imestaze}` u početnom direktoriju projekta.

Appfile sadrži postavke projekta i podatke koji olakšavaju korištenje alata kao što su korisničko ime i identifikator Apple Developer računa.

Uz navedene datoteke, fastlane direktorij može sadržavati i brojne druge datoteke koje sadrže postavke pojedinog dodatka.

C.1. Dohvat ovisnosti

Za dohvat ovisnosti koristim alate Carthage i CocoaPods. Fastlane podržava oba alata pomoću `carthage` i `cocoapods` dodataka. Dodaci interno koriste alate Carthage i CocoaPods te omogućavaju njihovo jednostavno korištenje i konfiguraciju u skladu s fastlane pristupom.

Staza koja pokreće dohvat ovisnosti alatom Carthage prikazana je u nastavku. Naredbi sam dodao dva argumenta: `platform: 'iOS'` budući da želim dohvatiti ovisnosti samo za iOS operacijski sustav i `cache_builds: true` kako ne bi dohvaćao ovisnosti koje su već dostupne na računalu.

Skripta C.3: Dohvaćanje ovisnosti korištenjem carthage dodatka

```
lane :carthage do
  carthage(platform: 'iOS', cache_builds: true)
end
```

Za dohvat ovisnosti pomoću CocoaPods alata dovoljno je u stazi pozvati `cocoapods` naredbu.

Skripta C.4: Dohvaćanje ovisnosti korištenjem cocoapods dodatka

```
lane :cocoapods do
```

```
cocoapods
end
```

Budući da dohvat ovisnosti želim ostvariti prije svake staze, navedene se naredbe mogu specificirati u `before_all` stazi.

Skripta C.5: Dohvat ovisnosti prije obavljanja svake staze

```
before_all do |lane|
  cocoapods
  carthage(platform: 'iOS', cache_builds: true)
end
```

C.2. Izgradnja

Fastlane za izgradnju projekta koristi dodatak `gym`[10]. Dodatak za izgradnju projekta koristi alat `xcodebuild`. Međutim, sučelje dodatka `gym` puno je jednostavnije i kompatibilnije sa stilom fastlane alata. Isto tako, alat automatski detektira projekte, sheme i ciljeve na temelju kojih obavlja izgradnju, formatira ispis kako bi bio jednostavno čitljiv te kreira datoteke potrebne za isporuku projekta.

Rezultati izvršavanja naredbe se zapisuju u `fastlane\report.xml` datoteku. Uz rezultat izvršavanja svih naredba, datoteka sadrži i njihov redoslijed te trajanje.

Izgradnja projekta obavlja se pozivom naredbe `fastlane gym`. Ako projekt sadrži više shema, naredba traži korisnika odabir željene sheme. Primjer staze koja pokreće izgradnju projekta prikazana je u skripti C.6. Naredba specificira shemu i konfiguraciju koja se koristi za izgradnju.

Skripta C.6: Izgradnja projekta korištenjem `gym` dodatka

```
lane :develop do
  increment_build_number
  gym(scheme: "Diplomski_rad", configuration: "Debug")
end
```

C.3. Testiranje

Fastlane testiranje projekta obavlja korištenjem dodatka `scan`[12]. Operaciju testiranja obavlja korištenjem `xcodebuild` alata. Dodatak samostalno detektira projekte i

sheme za koje pokreće testove. U slučaju postojanja više shema, dodatak korisnika traži odabir željene sheme te zatim pokreće sve testne ciljeve koje ona definira.

Dodatak rezultate pohranjuje u `fastlane\test_output\report.junit` datoteci. Format `junit` je široko prihvaćen način zapisa rezultata testova te postoji veliki broj alata za njegov vizualan prikaz. Skripta C.7 prikazuje stazu koja obavlja testiranje projekta.

Skripta C.7: Testiranje projekta korištenjem scan dodatka

```
lane :test do
  scan(
    workspace: "Example.xcworkspace",
    devices: ["iPhone 6s", "iPad Air"]
  )
end
```