

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.

*Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu \izvornik.*

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	27
2.3. Testiranje	30
2.3.1. Automatizacija testiranja	31
2.4. Osiguranje kvalitete	33
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	48
4. Kontinuirana isporuka	52
4.1. Kontinuirana isporuka za iOS operacijski sustav	54
4.2. Pokretanje isporuke	55
4.3. Implementacija kontinuirane isporuke	58

5. Zaključak	60
Literatura	61
Appendices	64
A. Tehnički detalji implementacije kontinuirane integracije, dostave i isporuke	65
A.0.1. Ručna isporuka	65
A.0.2. Gitflow radni tok	65
A.0.3. Konfiguracija projekta	67
A.0.4. Automatsko pokretanje isporuke	68
B. Alat Xcodebuild	70
B.0.1. Testiranje	71
B.0.2. Osiguranje kvalitete	71
B.0.3. Isporuka	72
C. Fastlane	73
C.1. Dohvat ovisnosti	74
C.2. Izgradnja	75
C.3. Testiranje	75
D. Usporedba alata za implementaciju kontinuirane integracije	76

1. Uvod

Profesionalna izrada iOS programske potpore za iOS operacijski sustav zahtjeva čestu isporuku različitih verzija aplikacije. Osim isporuke produkcijske verzije programske potpore čija učestalost može varirati od nekoliko puta tjedno do jednom u nekoliko mjeseci, u sklopu razvoja se isporučuju i druge verzije kao što su verzija za testiranje, verzija koja prikazuje trenutno stanje razvoja i pripremna produkcijska verzija. 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 isporuka programske potpore je vremenski zahtjevnija te je podložna ljudskoj pogrešci. Zbog navedenog se izbjegava često obavljanje isporuke što dovodi do zajedničke isporuke većeg broja promjena što zatim dovodi do većeg broja pogrešaka, lošije kvalitete programske potpore te sporije isporuke novih funkcionalnosti.

Ovaj problem je moguće riješiti automatizacijom cijelog ili većeg dijela isporuke programske potpore za iOS operacijski sustav. Međutim, kako bi isporučili programsku potporu, prvo ju moramo arhivirati te zatim dobivenu arhivu objaviti na željenoj platformi. Arhiviranje je proces pripreme iOS projekta za isporuku. Dodano, i proces arhiviranja i proces objave značajno ovise o platformi na kojoj objavljujemo programsku potporu.

Prije arhiviranja programske potpore je istu potrebno izgraditi te utvrditi da zadovoljava sve postavljene uvjete. Uvjeti 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 funkcionalnosti možemo 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 prvenstveno kako bi poboljšala kvalitetu programske potpore. Ovako automati-

zirani proces se provodi nad svakom novom verzijom programske potpore čime se osigurava njena ispravnost. Dodatno, kontinuirana integracija potiče učestalu integraciju radnih kopija s glavnom kopijom kako bi se izbjegla pojava konflikt 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 time povećati učestalost objave novih funkcionalnosti. Svaka funkcionalnost koja je razvijena a nije objavljena predstavlja gubitak timu. Dokle god razvijena funkcionalnost sjedi neobjavljena, ona ne donosi korist zbog koje je razvijena. Dodatno, pokazano je da isporuka manjeg broja promjena pozitivno utječe na kvalitetu isporučene programske potpore.

Kontinuirana isporuka nastoji što više smanjiti vrijeme koje protekne od završetka razvoja do objave funkcionalnosti, pa ga čak i u potpunosti eliminirati. Kontinuirana isporuka ovo ostvaruje automatskom isporukom promjena. Međutim, 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 nekad čak i ograničenom broju stvarnih korisnika. Tek nakon svih ovih faza se 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 navedenih procesa, već ih iskoristiti na stvarnim projektima. Zbog navedenog, implementacija mora biti jednostavno prenosiva na drugi projekt. Zbog ovog ograničenja ne samo da nastojim automatizirati isporuku, već nastojim automatizirati i proces dodavanja automatske isporuke iOS projektu.

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

2. Kontinuirana integracija

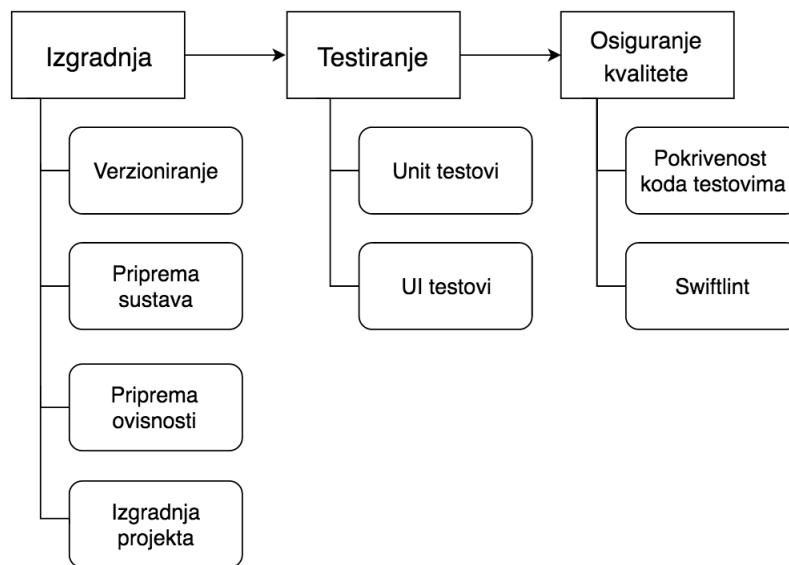
Kontinuirana integracija je praksa spajanja razvojnih kopija koda s glavnom kopijom nekoliko puta dnevno. Termin je prvi predložio i iskoristio Grady Booch 1991. godine tijekom opisa metode danas poznate kao Boochova metoda (engl. *Booch method*)[16].

Glavni cilj metode je 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. Lokalnu kopiju izvornog koda nazivamo *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 od kako 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. Dodatno, spajanje često nije moguće obaviti automatski. Ova se pojava naziva konflikt te se javlja 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 pojmovi su 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 se daje malo pozornosti.

Kontinuiranu integraciju je moguće podijeliti na tri generalne faze: izgradnju, testiranje i osiguranje kvalitete. Nadalje fazu izgradnje je moguće 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 je prikazana na slici 2.1.

Svaka od faza je obrađena zasebnim odlomkom u nastavku poglavlja. Proces verzioniranja je u praksi dio 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, Xcode i macOS Server te implementira veliki broj funkcionalnosti korištenih u sklopu kontinuirane integracije. Alat se pokazao najboljim među nekoliko sličnih

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

2.1. Priprema

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

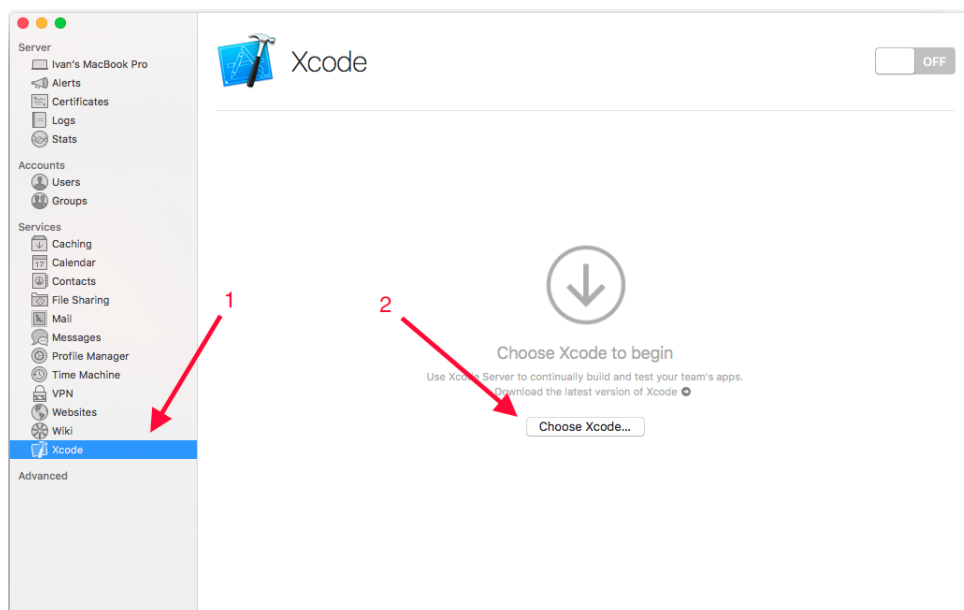
Međutim, macOS je vrlo siguran te zbog toga zatvoren operacijski sustav. Veliki broj alata zahtijeva korisničku interakciju zbog čega je njihovu instalaciju teško automatizirati. Ako se instalacija obavlja za više računala ili alat pristupa osjetljivim datotekama onda je instalaciju potrebno autorizirati lozinkom računala 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 priprema alate 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 korisničkom sučelje. Navedenim se alatima pristupa korištenjem naredbenog korisničkog sučelja - *ljuske*. U sklopu rada koristim *bash* ljusku. Pristup naredbenom korisničkom sučelju se ostvaruje 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 je spoj dva alata - Xcodea i macOS Servera. Xcode je integrirani sustav razvoj programske potpore za iOS, macOS, tvOS i watchOS operacijske sustave. macOS Server je alat za automatizaciju procesa na macOS operacijskom sustavu. Prije



Slika 2.2: Povezivanje macOS Server i Xcode aplikacija

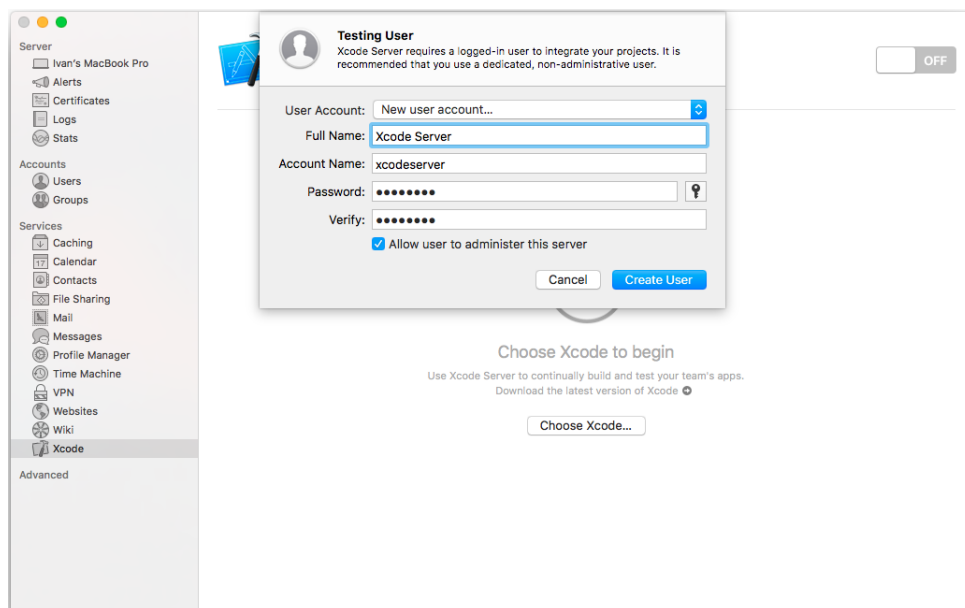
implementacije kontinuirane integracije je potrebno preuzeti oba alata, instalirati i konfigurirati oba alata.

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

Nakon instalacije alata je potrebno kreirati Xcode Server alat povezivanjem Xcode i macOS Server aplikacija. Pokrenuti macOS Server, u lijevom izborniku odabrati *Xcode* te odabrati opciju *Choose Xcode...* U novo otvorenom izborniku 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 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*. Slijediti upute nakon kreiranja računa za dovršetak spajanja.

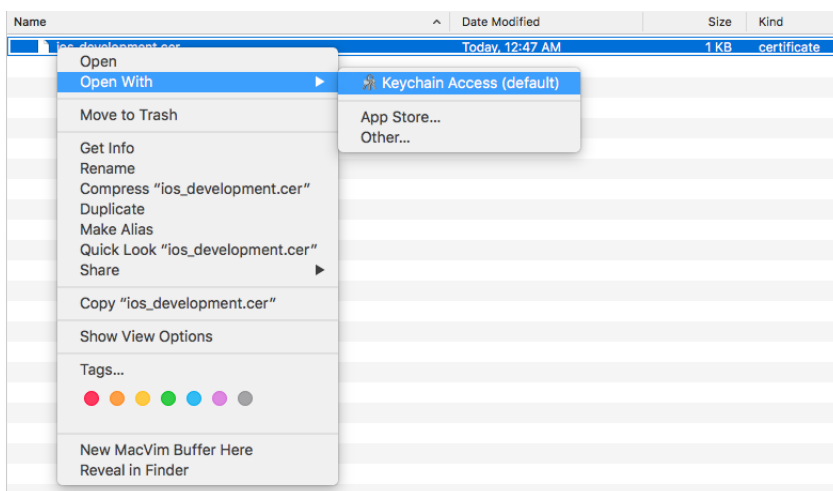
Proces potpisivanja koda osigurava autentičnost i neizmjenjenost kreirane program-



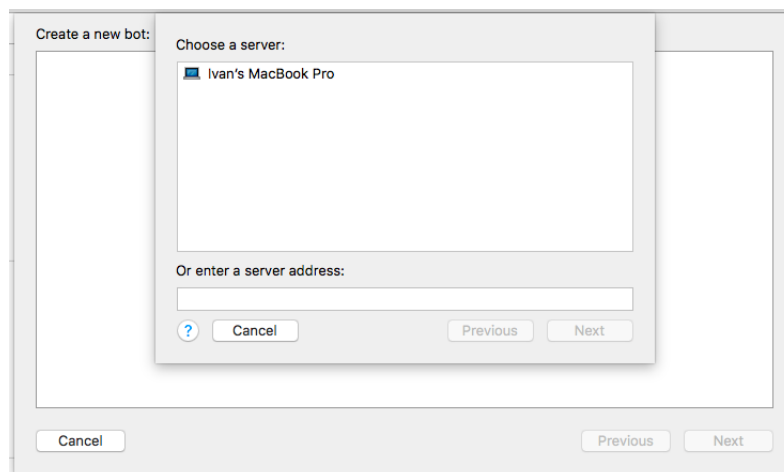
Slika 2.3: Kreiranje Xcode Server korisničkog računa

ske potpore. Proces je detaljnije objašnjen u 3.1 odlomku. Za sada je dovoljno znati da proces zahtjeva postojanje tri artefakta: certifikat člana tima, identifikator aplikacije i pripremni profil aplikacije za tip isporuke koji se koristi. Navedene certifikate je moguće kreirati i preuzeti s web stranice <https://developer.apple.com/account>.

Certifikat i identifikator se instaliraju korištenjem Keychain Access aplikacije. Dovoljno ih je jednostavno pokrenuti korištenjem navedene aplikacije. Pripremne profile je potrebno spremiti na lokaciji `~/Library/MobileDevice/ProvisioningProfiles`.



Slika 2.4: Dodavanje certifikata korištenjem Keychain Access aplikacije



Slika 2.5: Odabir macOS Servera aplikacije za obavljanje kontinuirane integracije

Xcode Server automatizaciju izgradnje, testiranja i isporuke ostvaruje korištenjem alata nazvanog *bot*. Bot se kreira i konfigurira korištenjem Xcode aplikacije a pokreće na macOS Server aplikaciji. Navedene aplikacije se ne moraju nalaziti na istom računalu ali moraju biti povezane.

Pokrenuti željeni projekt u aplikaciji Xcode. Kreiranje kontinuirane integracije se pokreće odabirom opcije *Product -> Create bot...*

Nakon odabira navedene opcije otvara se novi prozor u kojem je potrebno imenovati bot te odabrati macOS Server aplikaciju na kojoj će se bot izvršavati. Ukoliko macOS Server nije vidljiv, 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]. Dodatno, 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 alata *ruby*. Nakon pokretanja naredbe slijediti upute instalacije.

Skripta 2.1: Instalacija Homebrew alata

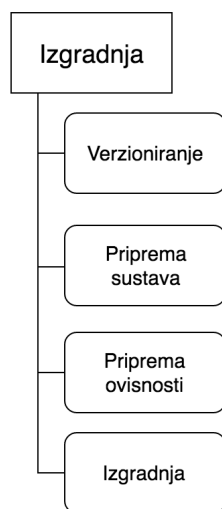
```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install)"
```

2.2. Izgradnja

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

Interpretirani jezici (engl. *interpreted languages*) se ne prevode 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 s pojmom izgradnje vezemo sve procese koji su dio pretvaranja izvornog koda u željeni artefakt. Ovisno o jeziku i alatima koje koristimo, proces izgradnje može značajno oscilirati u svojoj veličini. Generalno proces izgradnje možemo podijeliti na verzioniranje, pripremu sustava za izgradnju, dohvat i pripremu ovisnosti (engl. *dependancies*) 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 osigurava postojanje ovisnosti koje zahtijeva izvorni kod. Ovisnosti dijelimo na dva tipa, ovisnosti koje su dio razvojne okoline i vanjske (engl. *third party*) ovisnosti,



Slika 2.6: Podfaze procesa izgradnje

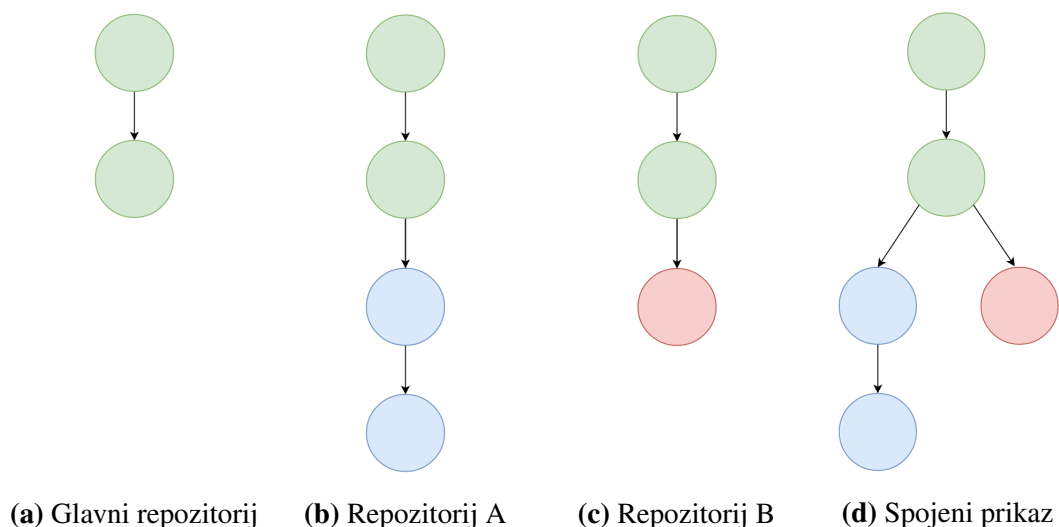
najčešće razvijene od strane zajednice. Kompajliranje prevodi izvorni kod u izvršivi artefakt. Kod interpretiranih jezika ovaj je proces često zamijenjen statičkom i dinamičkom provjerom izvedivosti programa. Zbog jednostavnijeg sporazumijevanja oba procesa nazivam izgradnja projekta. Slika 2.6 prikazuje podjelu procesa izgradnje.

Osim kreiranja artefakta, izgradnja provjerava i je li zadana verzija izvornog koda izgrađiva. Kod je izgrađiv ako se u procesu izgradnje ne izaziva pogrešku, odnosno ako se isti ispravno izvrši. Pogrešku može izazvati neispravnost u izvornom kodu, neispravna konfiguracija sustava, nepostojanje potrebnog alata ili neka druga neispravnost. Izgrađivost sustava je preduvjet za testiranja i isporuku. Samim time je automatizacija izgradnje preduvjet za automatizaciju testiranja i automatizaciju isporuke.

2.2.1. Verzioniranje

Verzioniranje je proces dodjele jedinstvene oznake (engl. *id*) stanju repozitorija[20]. Repozitorij je verzioniran direktorij te 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 se naziva verzija (engl. *commit*).

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 verzije poredamo kronološki po datumu obavljanja izmjena, onda svaka pojedina verzija ne



Slika 2.7: Stablo promjena

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 se kreira zajedničkim prikazom stabla promjena svih članova.

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

Unutarnje verzioniranje koda se naziva kontrola verzija[21]. Sustavi koji implementiraju proces kontrole verzija se nazivaju sustavi za kontrolu verzija. Kroz povijest je razvijen veliki broj sustava za kontrolu verzija te je danas 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, je kreiran 2000. godine u sklopu projekta vođenog od Apache Software Foundation zajednice. Alat je centraliziran, siguran i jednostavan za korištenje te je danas objavljen kao alat otvorenog koda. Generalno postoji jedan glavni repozitorij kojeg članovi tima kloniraju, uređuju te zatim lokalne promjene sinkroniziraju s glavnim repozitorijem.

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 svna, 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 od drugom. Zbog navedenog je pomoću gita moguće implementirati proizvoljan pristup verzioniranju. Bio to centralizirani repozitorij nalik na svnov pristup, pristup s osobama zaduženim za odobravanje promjena, distribuirani model i drugo. Najvažnije,

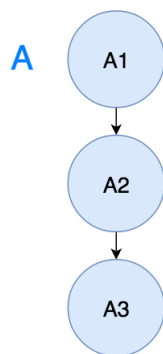
git je jednostavan ali vrlo moćan alat. Implementacija osnovnih funkcionalnosti je intuitivna dok istovremeno postoji podrška za vrlo kompleksne pothvate.

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 ga čine lakšim za upoznavanje i korištenje. Zbog navedenog u ovom radu koristim git. Sve se funkcionalnosti mogu, uz manju modifikaciju, implementirati i korištenjem svna.

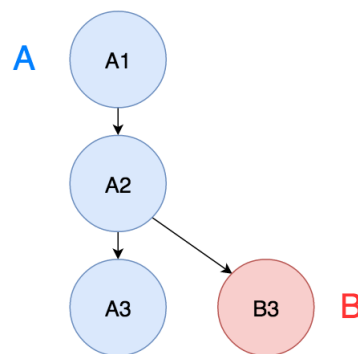
Uvod u git Temeljne funkcionalnosti git alata su repozitoriji i grane. Repozitorij je direktorij koji je verzioniran korištenjem git sustava za kontrolu verzija. Ovaj repozitorij sadrži direktorij `.git` koji specificira na koji se način verzionira direktorij te sadrži informacije o repozitoriju.

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 se mogu, uz postojanje odgovarajuće autorizacije, prenijeti na izvorni repozitorij. Prijenos promjena se ne mora obavljati isključivo između izvornog i kloniranog repozitorija, već se može obaviti između bilo koja dva povezana repozitorija.

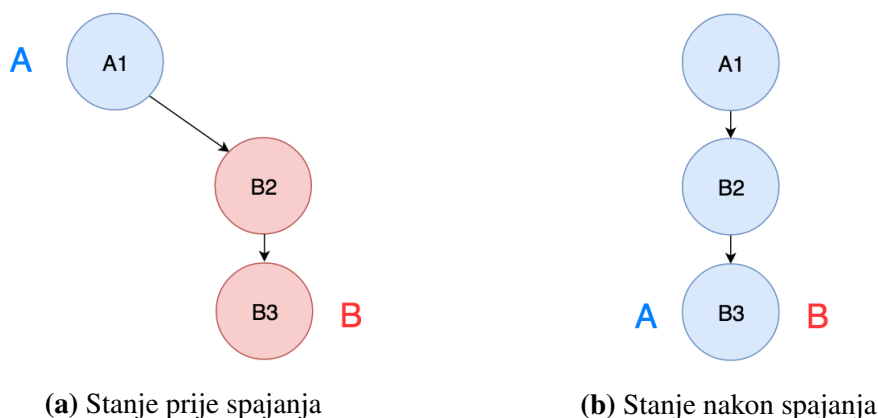
Prilikom kreiranja git repozitorija stvara se i glavna grana (eng. master branch) repozitorija. Grana je definirana slijedom verzija koje su obavljene na njoj. Stvaranje nove verzije na trenutnoj grani se ostvaruje potvrđivanjem promjena (engl. *commit*) koje su dodane repozitoriju. Potvrđivanje promjena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

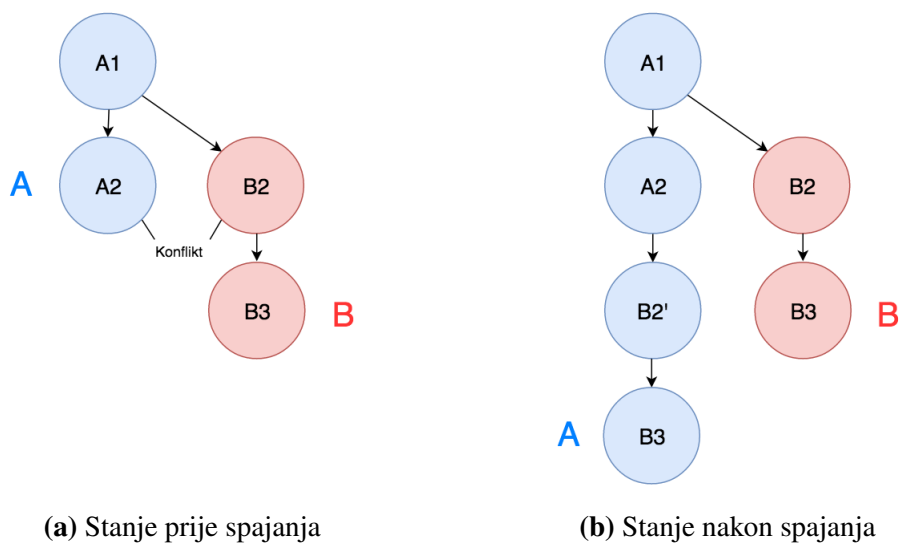
A2 i A3.

Nova grana se može kreirati iz bilo kojeg stanja postojeće granje. Ovaj se postupak naziva grananje (engl. *branching*). Izvorna i kreirana grana dijele zajedničku povijest do trenutka grananja. Daljnje promjene se primjenjuju 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 dodajemo 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 se dodaju dva nova stanja, B2 i B3. U međuvremenu je grana A ostala nepromijenjena. Zbog navedenog je spajanje grana moguće 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. Dodatno, samo spajanje je 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.

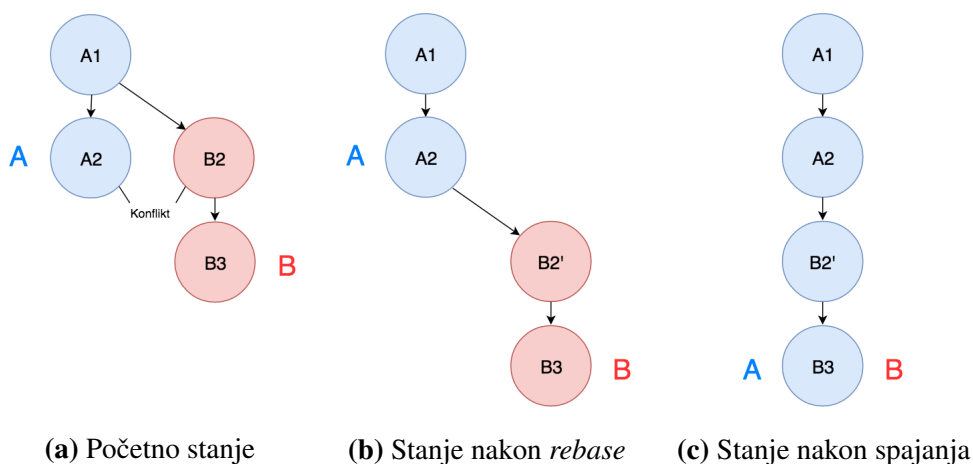
Međutim, ako promjene izazivaju konflikte, onda je te konflikte potrebno ručno razriješiti. Otklanjanje konflikata uzrokuje izmjenu verzija jedne ili obje grane. Proces otklanjanja konflikata se najčešće odrađuje dodavanjem jedne po jedne verzije izvorne



Slika 2.11: Spajanja otklananjem konflikta

grane na odredišnu granu. Ako dodana verzija ne izaziva konflikt, ona se jednostavno dodaje na vrh odredišne grane. Međutim, ako verzija izaziva konflikt, tada se isti otklanja modificirajući dodanu verziju. 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 nastala 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 verziju A grane. Slika 2.12b prikazuje stanje nakon obavljanja *re-base* 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 se još uvijek mijenja, međutim sada je povijest repozitorija linearna.

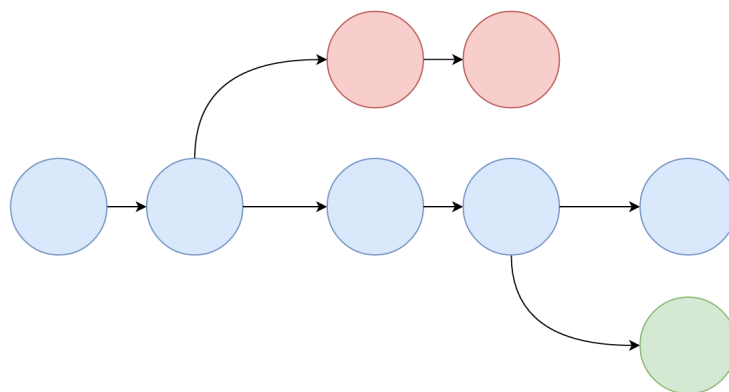
Tijek verzioniranja Ostaje otvoreno pitanje kako koristiti alat za kontrolu verzija. Koliko često kreirati novu verziju koda i koliko često promjene spajati sa zajedničkim repozitorijem. Kod korištenja gita se javljaju i pitanja kako organizirati repozitorije i sustav grananja.

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 pristup je 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 (engl. *forking workflow*). Svaki od navedenih pristupa ima svoje prednosti i mane te se koristi u različitim tipovima projekta[3].

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 iste spaja 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 je vrlo jednostavan te je sličan načinu rada svna.

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 funkcionalnost implementira u grani kreiranoj specifično za nju. Programer za novu funkcionalnost kreira novu granu u lokalnom repozitoriju te u nju dodaje promjene. Po



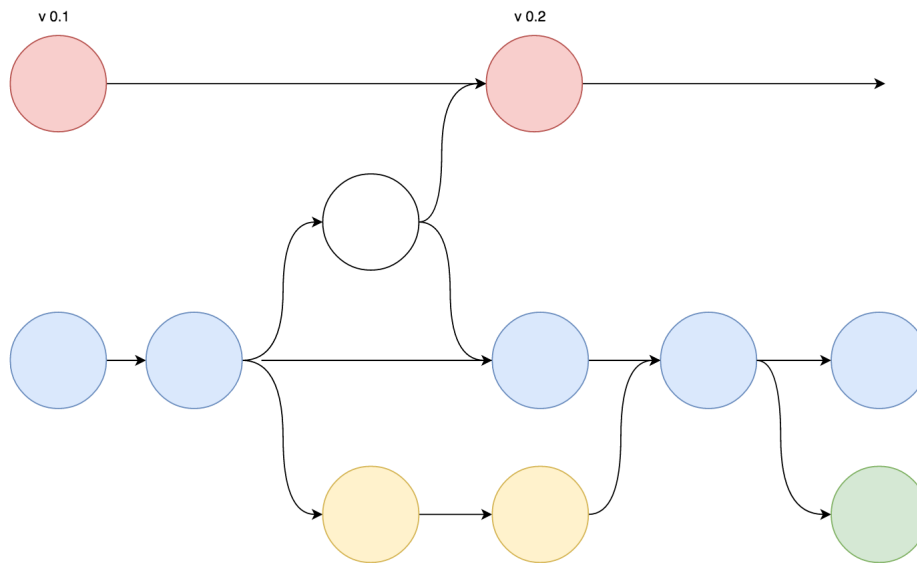
Slika 2.13: Primjer tijek rada grananja funkcionalnosti

završetku implementacije funkcionalnosti programer granu spaja s glavnom granom centralnog repozitorija. Ovaj proces daje jasniji uvid u napredak projekta i implementirane funkcionalnosti. Dodatno, 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*) je vrlo slična glavnoj grani u procesu grananja funkcionalnosti. Grana za novu funkcionalnost se kreira iz razvojne grane te se po završetku implementacije u nju spaja. S druge strane, glavna grana sadrži samo produkcijske verzije izvornog koda, odnosno one verzije projekta koje su obavljene 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 verzija glavne grane se zatim objavljuje. Verzije na glavnoj grani se označavaju sa objavljenom verzijom projekta.

Primjer korištenja gitflow procesa je prikazan na slici 2.14. Crvenom bojom je prikazana glavna grana a plavom razvoja grana. Grane funkcionalnosti, prikazane zelenom i žutom bojom se granaju iz razvojne grane te u nju spajaju. Bijelom bojom je označena 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 kod pripreme za objavu bile dodane projektu.

Navedeni pristup olakšava upravljanje objavom projekta. Buduće da je krucijalno



Slika 2.14: Primjer gitflow tijek rada

objaviti ispravan produkt sam proces objave treba biti kontroliran i 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. Dodatno, programer može predložiti dodavanje vlastitih promjena drugom repozitoriju. Navedeni se proces naziva zahtjev za povlačenjem promjena (engl. *pull request*).

Forking tijek rada se najčešće 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 vodili samo ovim principom centralizirani repozitorij bi najbolje zadovoljavao postavljene zahtjeve. Međutim, centralizirani repozitorij se u praksi ne koristi za ništa osim najjednostavnijih projekata.

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

spajanja radnih kopija. Na primjer, gitflow tijekom rada spajanje radne kopije s glavnom kopijom obavlja po završetku implementacije funkcionalnosti. Što je veća funkcionalnost koja se implementira, to će duže radna kopija ostati izdvojena. Zbog navedenog je potrebno posao razdijeliti na male dijelove. Navedeno 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 napredniji pristupi verzioniranju pruža, kao što su lakše praćenje razvoja, zahtjevi za spajanjem i lakša objava projekta u produkciju nadilaze nešto duže vrijeme izdvojenosti radnih kopija.

U praktičnom dijelu rada koristim gitflow tijekom rada. Ovaj tijekom rada najbolje odgovara zahtjevima i tipu projekta. Dodatno, 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 ću izdvojiti u zasebnu granu. Navedeni proces omogućava lako praćenje testnih verzija te olakšava implementaciju procesa isporuke testne verzije.

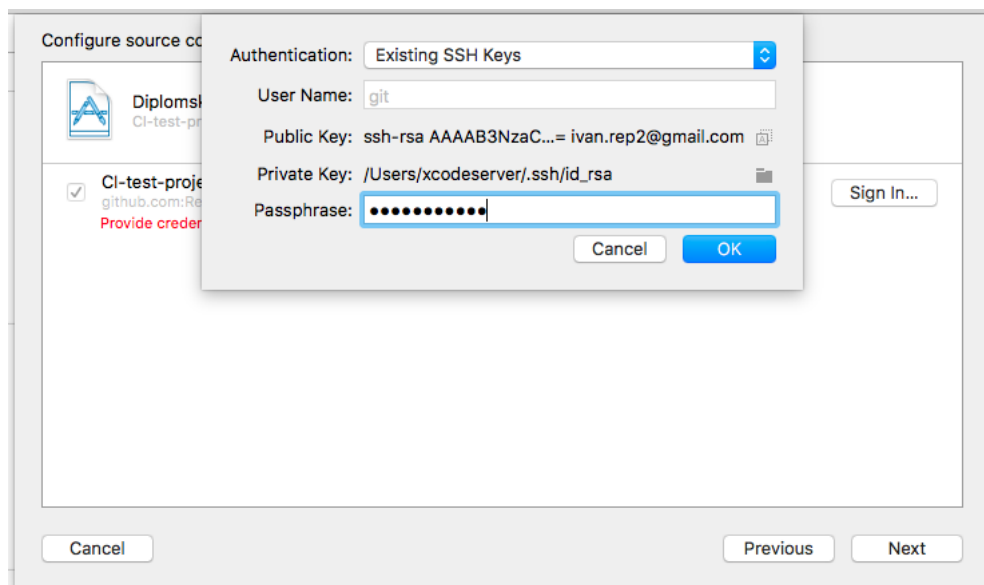
Verzioniranje u sklopu kontinuirane integracije Proces kontinuirane integracije započinje dohvatom željene verzije repozitorija. Kako bi dohvatio željenu verziju, proces kontinuirane integracije mora imati pristupiti repozitoriju. Xcode Server omogućava korištenje lokalnog ili udaljenog repozitorija 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 spremati 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. Ova tip autentifikacije je pogodniji za automatizaciju zbog čega ga koristim u sklopu ovog rada. Proces kreiranja i konfiguriranja SSH autentifikacije je detaljnije objašnjen u sljedećem odlomku.

Xcode Server automatski detektira alat kojim je repozitorij verzioniran. Nakon autentifikacije pristupa je potrebno odabrati granu za koju se kreira proces integracije. Pojedini bot integraciju obavlja za samo jednu granu zbog čega 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-

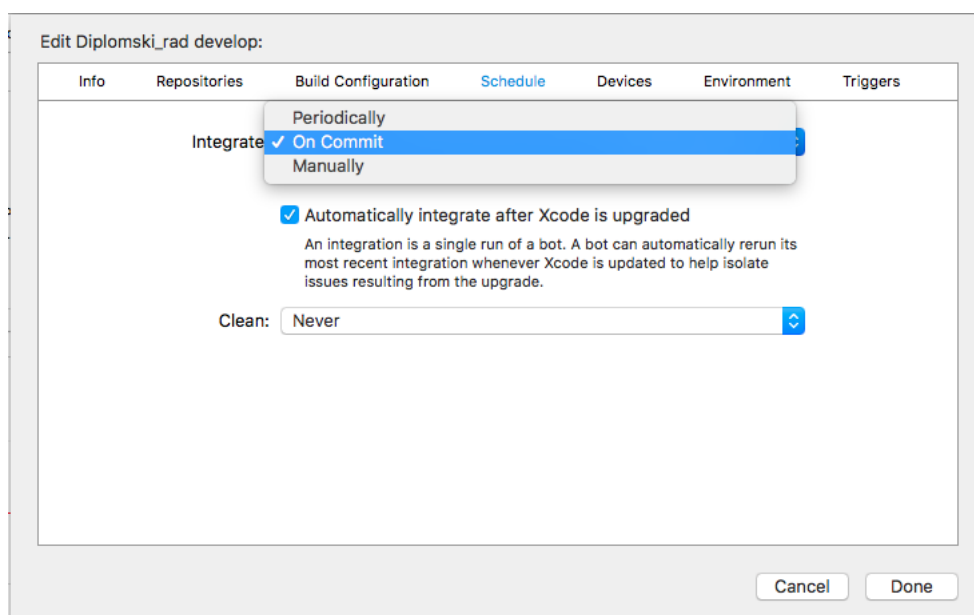


Slika 2.15: Dodavanje SSH ključa

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

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

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



Slika 2.16: Odabir načina pokretanja procesa integracije

Ako SSH ključ već ne postoji u navedenom direktoriju, onda potrebno je kreirati isti. Skripta 2.2 prikazuje proces generiranja ključa. Naredba pod #1 generira novi SSH ključ sa željenom e-mail adresom. Preporučeno je ključ u toku kreiranja 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 je potrebno registrirati na platformi koja hosta repozitorij. Naredba #4 kopira javni dio novo kreiranog ključa.

Skripta 2.2: Postavljanje SSH autentifikacije

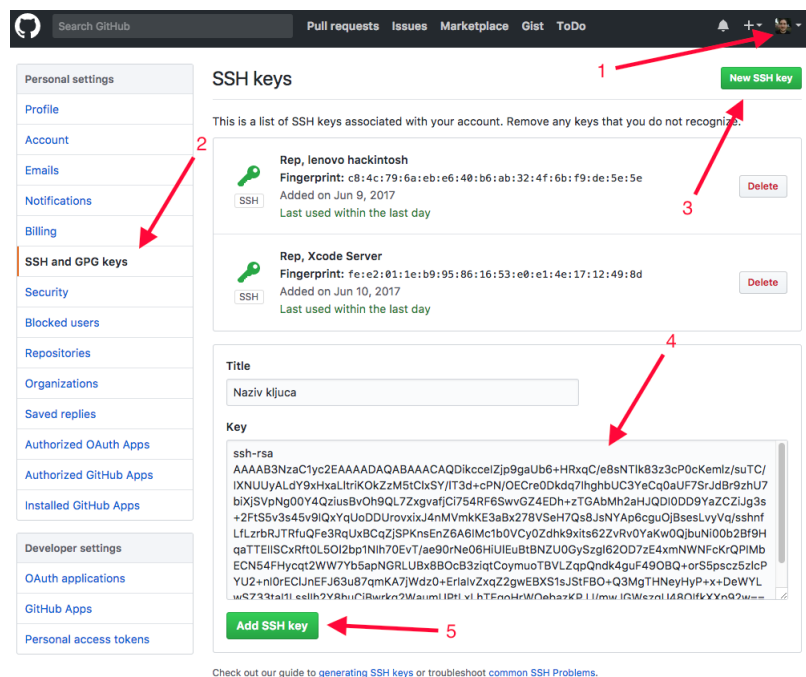
```
ssh-keygen -t rsa -b 4096 -C "{e-mail adresa}" #1
```

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

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

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

U sklopu rada koristim GitHub platformu. Slika 2.17 prikazuje proces registriranja SSH ključa na GitHub platformi. SSH ključ se dodaje odabirom opcije *Settings -> SSH and GPG keys -> New SSH key* te dodavanjem kopiranog javnog dijela ključa u polje



Slika 2.17: Dodavanje SSH ključa na GitHub platformu

za ključ. Nakon spremanja ključa isti je moguće koristiti za autorizaciju komunikacije s GitHub platformom.

2.2.2. Priprema sustava

Priprema sustava se sastoji 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 instalacija macOS operacijskog sustava. U slučaju nepostojanja alata isti je potrebno 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"

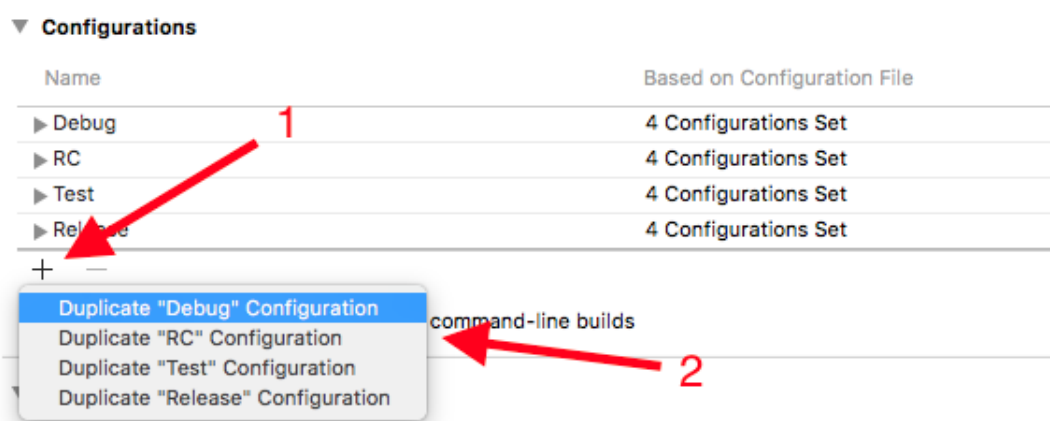
    brew install swiftlint
```

fi

Naredba pod #1 provjerava i instalira CocoaPods alat korištenjem alta `gem` dostupnog u sklopu instalacije macOS operacijskog sustava. Kako bi 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 je pohranjena unutar `.xcodeproj` datoteke. Datoteka je namijenjena za modifikaciju i čitanje korištenjem alata Xcode. Datoteka nije pogodna za račnu izmjenu i čitanje. Kako bi se olakšalo nadgledanje i modifikacija postavka projekta često se koriste `xcconfig` datoteke.

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



Slika 2.18: Kreiranje konfiguracije projekta

Korištenjem novo kreirane konfiguracije odabrati željenu `xcconfig` datoteku.

2.2.3. Upravljanje ovisnostima

Prije izgradnje projekta je potrebno dohvatiti ovisnosti koje projekt koristi. Za dohvaćanje ovisnosti se u iOS razvoju koriste dva alata: *CocoaPods* i *Carthage*. Oba sustava se široko koriste te izbor uvelike ovisi o osobnom ukusu. Zbog navedenog 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 dohvata ovisnosti. Alat samostalno kreira i konfigurira radno okruženje te time olakšava proces upravljanja ovisnostima.

Međutim, najveći problem alata je upravo 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. Navedeno otežava korištenje privatnih biblioteka i biblioteka u razvoju.

Inicijalizacija CocoaPods alata je prikazana 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 postojanja iste nastavlja s izvođenjem skripte. Naredba #2 provjerava postojanje CocoaPods alata. Ako alat nije instaliran, isti 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 dependancy 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.

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 objavljenu ovisnost Unbox objavljenu na Github platformi te privatnu ovisnost objavljenu na Gitlab platformi.

Dohvaćanje ovisnosti se 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 iste nisu već 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"

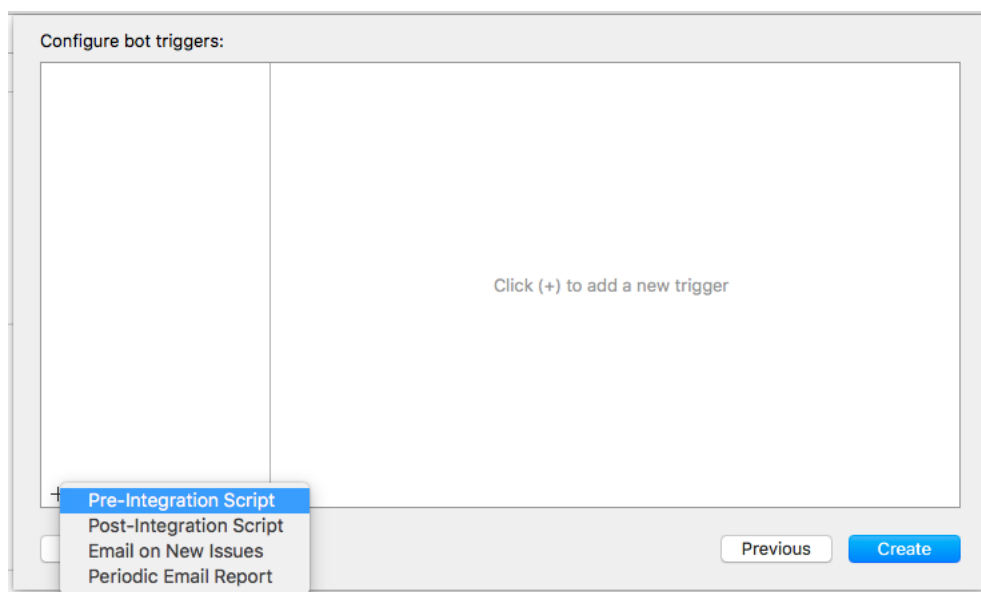
        brew install carthage
    fi

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

    echo "Finished dependancy fetch using Carthage"
fi
```

Automatizacija dohvaćanja ovisnosti Obje skripte je potrebno 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 novo kreirane skripte je potrebno pozvati sve naredbe koje se trebaju iz-



Slika 2.19: Dodavanje skripte koja se izvršava prije integracije botu

vršiti prije integracije. Radi jednostavnosti i fleksibilnosti je navedene naredbe korisno 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 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 prije definirane skripte. Na isti je način moguće dodati proizvoljan broj naredaba. Dodatno, 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 skripte se trebaju nalaziti u `scripts` direktoriju repozitorija.

2.2.4. Izgradnja

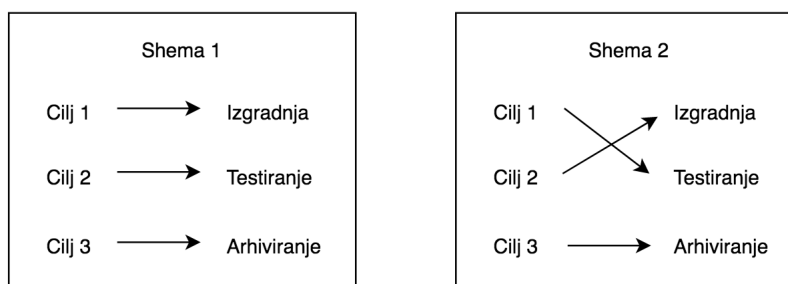
Izgradnja iOS aplikacija se obavlja korištenjem alat *xcodebuild*[1]. Alat je razvio Apple za izgradnju programske potpore za macOS operacijski sustav. Alat je vrlo moćan te pruža veliki broj funkcionalnosti i mogućih konfiguracija. Alat je proširen te danas podržava izgradnju aplikacija za iOS, tvOS i watchOS operacijske sustave. Alat izgradnju obavlja na korištenjem Xcode projekta. Prije definiranja procesa izgradnje se je potrebno upoznati 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 je Xcode daleko najkorišteniji. Svi alati koriste alat *xcodebuild* za izgradnju te zbog toga imaju vrlo sličnu strukturu projekta. Ovaj tip projekta se naziva Xcode projekt.

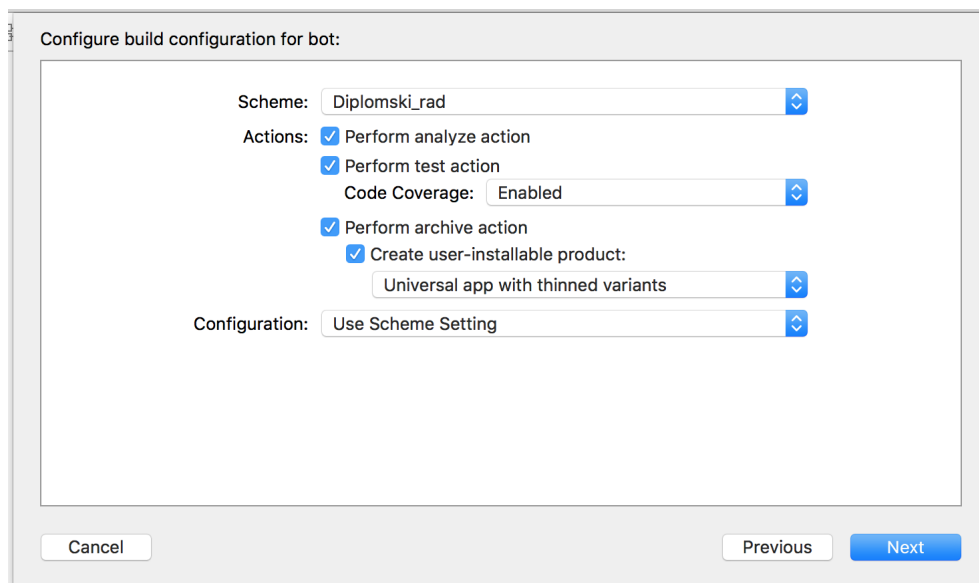
Xcode projekt sadrži jedan ili više ciljeva (engl. *target*) i jednu ili više shema (engl. *scheme*). Cilj definira postavke koji se koriste kod izvršavanja operacije za navedeni cilj. Jedan projekt može sadržavati više ciljeva. Pomoću ciljeva je moguće isti kod distribuirati za različite verzije operacijskog 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 je prikazan na slici 2.20. Projekt sadrži tri cilja i dvije sheme. Sheme različito definiraju koji se cilj koristi za koju operaciju.

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

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



Slika 2.20: Xcode projekt s tri cilja i dvije sheme

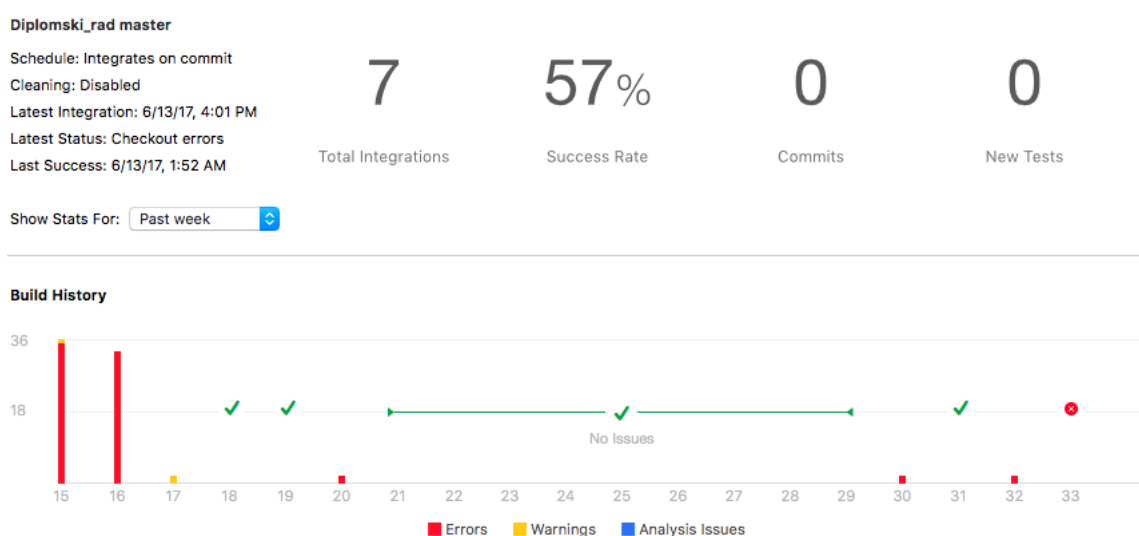


Slika 2.21: Konfiguracija osnovnih opcija integracije

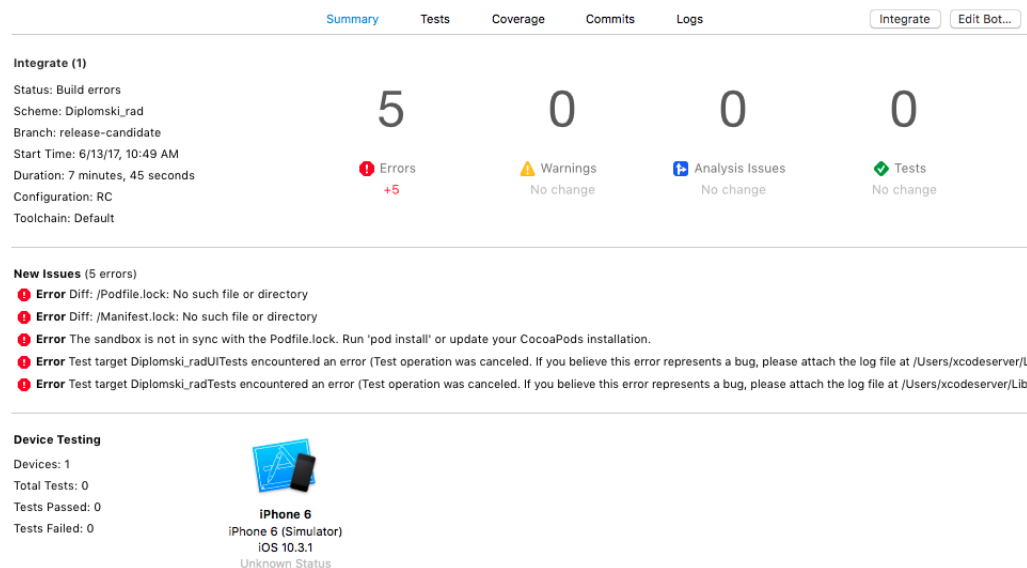
Xcode Server značajno olakšava korištenje alata xcodebuild. Nakon povezivanja bota s repozitorijem izvornog koda je moguće konfigurirati opcije integracije. Moguće je odabrati shemu projekta za koju se provodi integracija, operacije koje će se izvršavati u sklopu integracije te postavke koje se koriste za izgradnju projekta. Slika 2.21 prikazuje opcije koje se mogu konfigurirati.

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

Ovako konfiguriran bot provodi kontinuiranu integraciju za odabranu granu. Re-



Slika 2.22: Primjer prikaza rezultata kontinuirane integracije



Slika 2.23: Detaljan prikaz integracije

zultat integracije je moguće vidjeti na otvaranju početnog ekrana bota. Slika 2.22 prikazuje primjer rezultata integracije. Prvi redak slike prikazuje opće informacije o kontinuiranoj integraciji kao što broj obavljenih integracija, postotak uspješnosti te broj verzija repozitorija. Drugi redak slike prikazuje uspješnost izvođenja procesa izgradnje za svaku pojedinu integraciju.

Prve dvije prikazane integracije su završile 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 se otvara ekran s detaljima odabrane integracije. Detalji integracije su podijeljeni 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 novo pronađenih pogrešaka su prikazani 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 osiguravamo ispravan rad programske potpore, već sprječavamo nazadovanje koda (engl. *code regression*) i značajno smanjujemo potrebu za ručnim testiranjem ispravnosti[19].

Pogrešno je mišljenje da implementacija testova produljuje vrijeme razvoja. Svaku implementiranu funkcionalnost i obavljenju izmjeni je potrebno testirati. Jedino je pitanje hoće li se navedeno testiranje obavljati automatski. Jednom napisan kvalitetan test se može pokrenuti proizvoljan broj puta. S druge strane, provođenje ručnog testiranja svaki put zahtijeva vrijeme člana 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 se vjerojatnost pojave pogreške značajno smanjuje.

Ovaj odlomak ne ulazi u proces pisanja testova, već samo automatizira pokretanje istih. Implementacija kvalitetnih testova je vrlo 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 kroz ovu komunikaciju provjeravaju ispravnost izvođenja. Ovaj tip testa se pokreće 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 navede akcije. `UI` testovi pokreću dvije aplikacije: aplikaciju koju testiraju i aplikaciju koja simulira korisničku interakciju.

Oba tipa testova su implementirani kao testni ciljevi Xcode projekta. Kod testnog cilja nije dio produkcijskog 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.

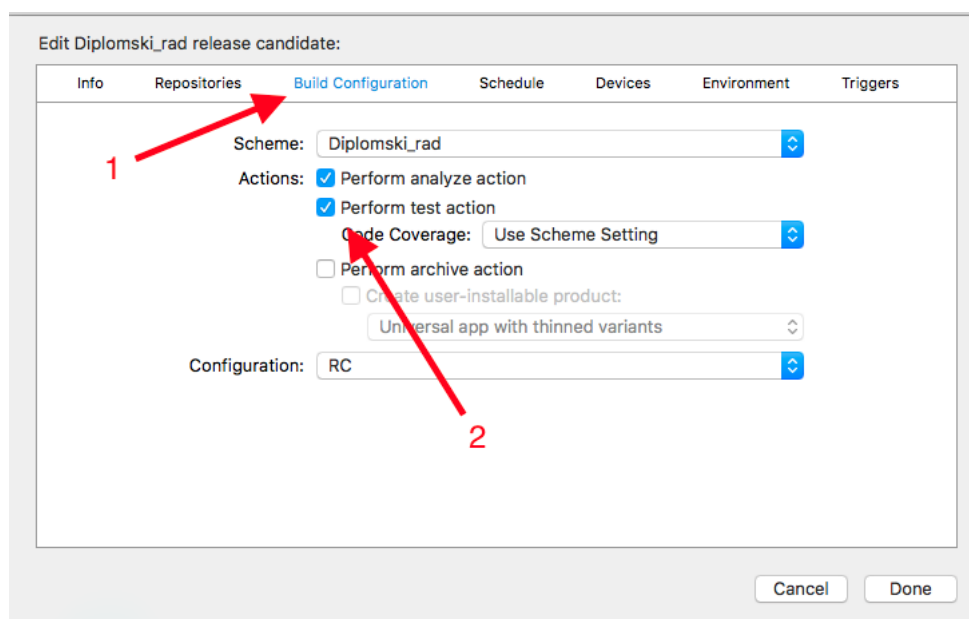
Proces testiranja se može pokrenuti na iOS Simulatoru, aplikaciji koja simulira iOS operacijski sustav na macOS operacijskom sustavu, ili stvarnom uređaju. Simulator se instalira u sklopu instalacije Xcode aplikacije.

2.3.1. Automatizacija testiranja

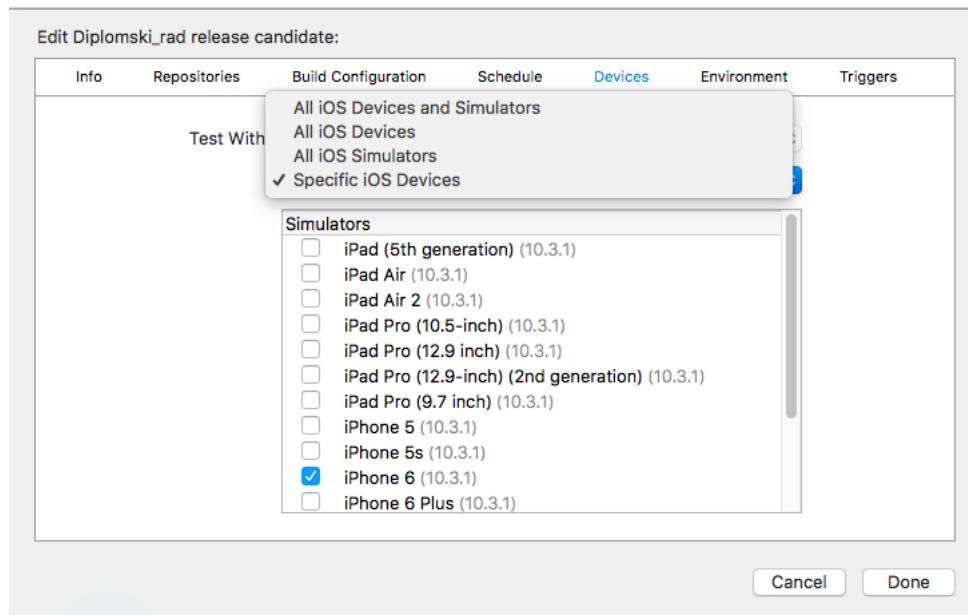
Xcode Server detektira testne ciljeve sheme za koju obavlja proces integracije. Za pokretanje testova u sklopu integracije odabrati opciju `Edit bot...` te u sekciji `Build configuration` označiti opciju `Perform test action`. Slika 2.24 prikazuje navedeni proces. Sljedeća obavljena integracija će pokrenuti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

Dodatno, 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 testni cilj je shemi moguće dodati korištenjem Xcode aplikacije. Pokrenuti iOS projekt te odabrati željenu shemu iz padajućeg izbornika u gornjoj alatnoj traci aplikacije. Odabirom opcije `Edit scheme` se otvara prozor prikazan na slici. U lijevom izborniku odabrati opciju `Test` te pritiskom na plus ikonu dodati željeni testni cilj postojećoj shemi. Nakon spremanja promjene integracija u sljedećem izvršavanju



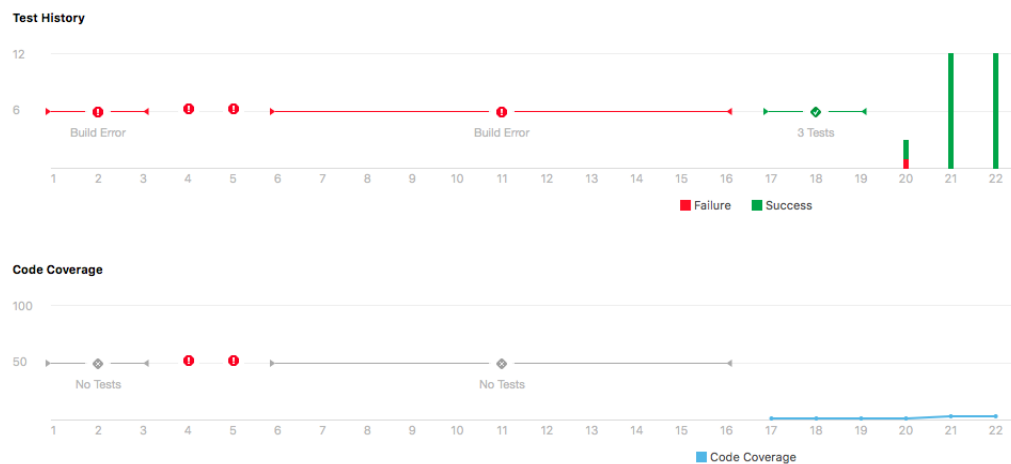
Slika 2.24: Uključivanje pokretanja testova u sklopu integracije



Slika 2.25: Odabir uređaja na kojima se pokreću testovi

obavlja i novo dodani testni cilj.

Rezultati obavljanja testova su također 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 se prikazuju detalji odabranog testiranja.



Slika 2.26: Rezultati procesa testiranja i osiguranja kvalitete

2.4. Osiguranje kvalitete

Osiguranje kvalitete (engl. *code quality*) je proces 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.

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


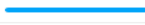


Pokrivenost koda testovima (engl. *code coverage*) je mjera koja govori u kolikom je postotku izvorni kod pokriven testovima. Svaki redak koda koji je barem jednom izvršen u procesu testiranja je pokriven testovima. Što je navedena mjera veća to je više koda testirano. Zbog toga možemo reći da veća pokrivenost koda testovima, u generalnom slučaju, vodi k boljoj kvaliteti 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[17].

Alat `xcodebuild` implementira mjerenje pokrivenosti koda testovima. Dovoljno je naredbi za pokretanje testova dodati argument `-showBuildSettings`. Ispis naredbe se pohranjuje 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 omogućava prikupljanje podataka o pokrivenosti koda testovima u sklopu integracije projekta. Prikupljanje podatka o pokrivenosti koda testovima se uključuje odabirom opcije `Code Coverage -> 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. Međutim, vrlo je teško automatizirati provjeru pokrivenosti koda testovima. Trenutno ne postoji mogućnost dohvata podataka prikazanih na slici izvan alata Xcode Server. Zbog navedenog provjeru pokrivenosti koda testovima obavljam ručno.

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

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

Slika 2.27: Pokrivenost datoteka testnog projekta testovima

koja nastoje izbjeći pojavu grešaka. Slični alati su kreirani za gotovo svaki programski jezik koji se koristi u praksi. Ovi alati se nazivaju alati za uređivanje koda (engl. *linting tools*). Navedene alate uglavnom kreira i održava zajednica.

Ne poštivanje 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 se parsira 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 je praksa 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 je produkt 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 je vremenski zahtjevan zbog čega se on ne provodi često. Rijetko provođenje procesa isporuke uzrokuje nekoliko problema. Prvo, rijetkom isporukom se veći 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. Dodatno, što duže funkcionalnost nije objavljena to je veći propušteni dobitak koji je funkcionalnost mogla donjeti. Svaka funkcionalnost se razvija s određenim ciljem. Nakon što je funkcionalnost razvijena a prije nego što je objavljena navedeni cilj ostaje bespotrebno ne ispunjen.

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 isto vrijeme poboljšati kvalitetu i osigurati ispravnost programskog produkta[18].

Kao što je navedeno u uvodu poglavlja, programsku potporu je 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 kreiranje arhive programske potpore. Arhiva omogućava instalaciju programske potpore na operacijskom sustavu za kojeg je programska potpora izgrađena.

Jednako tako je prije automatizacije isporuke produkta potrebno automatizirati izgradnju, testiranje, provjeru ispravnosti i arhiviranje programske potpore. Proces automatizacije izgradnje, testiranja i provjere ispravnosti se naziva kontinuirana integracija

te je definiran u poglavlju 2. Kontinuirana dostava obuhvaća proces kontinuirane integracije 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 se navedeni pristup koristi u ovom radu. Proces izgradnje, testiranja i provjere ispravnosti implementiram u sklopu kontinuirane integracije na koju dodajem automatizaciju isporuke i time ostvarujem kontinuiranu dostavu.

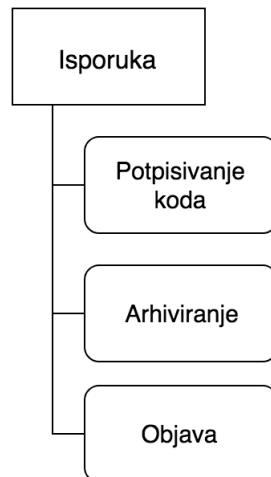
Važno je napomenuti da kontinuirana dostava ne isporučuje produkt automatski, već samo automatizira proces isporuke. Tim još uvijek mora pokrenuti proces isporuke za željenu verziju produkta. Automatska objava programske potpore je obrađena u sljedećem, 4 poglavlju.

Programsku potporu je moguće 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 se i automatizacije navedenih načina isporuke značajno 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 programske potpore 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 programske potpore 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 se nemože niti bi ga bilo pogodno automatizirati. Zbog sigurnosnih razloga Apple nije objavio proces direktne isporuke već ju je moguće obaviti isključivo korištenjem Xcode aplikacije.

Ad hoc i unutarnja isporuka su vrlo slične. Obje proizvode arhivu koju je potrebno samostalno dostaviti željenim korisnicima. Arhiva se može dostaviti ručno, na primjer korištenjem e-mail poruke ili se može objaviti na platformi za distribuciju programske potpore. Moguće je kreirati vlastitu platformu ili iskoristiti neku 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



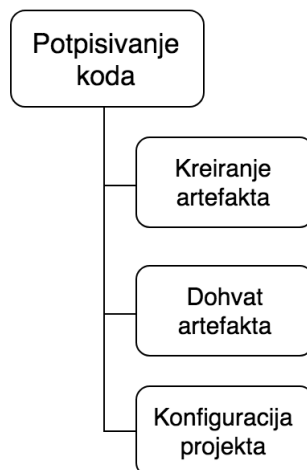
Slika 3.1: Proces isporuke

navedenog je korištenjem unutarne isporuke programsku potporu moguće isporučiti neograničenom broju korisnika. Za korištenje unutarne isporuke je potrebno posjedovati *Enterprise* Apple Developer račun, skuplju verziju običnog Apple Developer računa.

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

App Store isporuka se koristi 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 strog 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 nešto kroz 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 je definiran i automatiziran odvojeno u nastavku poglavlja. Slika 3.1 prikazuje navedenu podjelu.



Slika 3.2: Potpisivanje koda

3.1. Potpisivanje koda

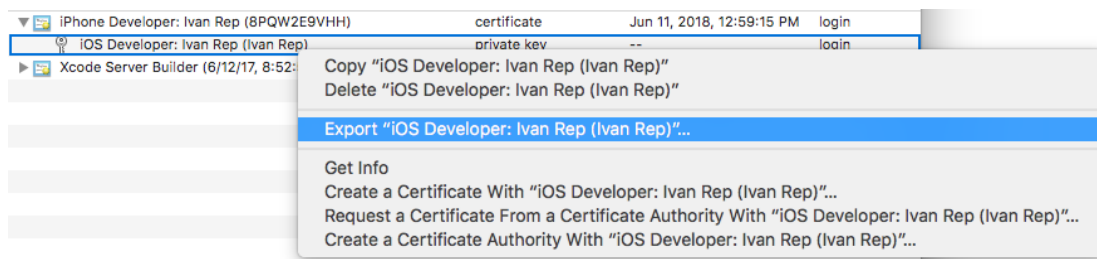
Kako bi osigurao autentičnost i neizmjenjenost programske potpore, iOS operacijski sustav implementira proces potpisivanja koda (engl. *code signing*). Proces korištenjem certifikata, identifikatora aplikacije (engl. *application identifier*) i pripremnih profila (engl. *provisioning profiles*) potpisuje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

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

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

Svi artefakti se kreiraju i dohvaćaju korištenjem `https://developer.apple.com` platforme. Potrebno se je prijaviti 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 nas zanimaju dva tipa certifikata: razvojni (engl. *development*) i produkcijski (engl. *production*) certifikati. Razvojni certifikati se koriste u razvoju i za direktnu isporuku programske potpore. Uobičajno je kreirati zaseban certifikat za svakog člana tima. Produkcijski certifikati se koriste 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



Slika 3.3: Izdvajanje privatnog ključa certifikata

korištenjem Keychain Access aplikacije.

Pokrenuti Keychain Access aplikaciju te u izborniku na vrhu ekrana odabrati opciju Certificate Assistant -> Request a Certificate. U novo otvorenom prozoru unijeti e-mail adresu Apple Developer računa, odabrati Save to disk opciju i lokaciju spremanja novo kreirane datoteke. Kreiranu datoteku dostaviti u formu za kreiranje certifikata na Apple Developer platformi te nakon toga obrisati datoteku.

Za korištenje certifikata je potrebno 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 posjeduje privatni ključ. Nakon kreiranja certifikata to je isključivo uređaj koji je kreirao CSR datoteku. Za dohvat privatnog ključa pokrenuti Keychain Access aplikaciju te pronaći željeni certifikat. Ako na uređaju postoji privatni ključ navedenog certifikata, isti je moguće izdvojiti korištenjem opcije *Export*. Slika 3.3 prikazuje izdvajanje privatnog ključa certifikata.

Ovako izdvojeni privatni ključ certifikata je moguće distribuirati. Nužno je osigurati tajnost privatnog ključa jer suprotno narušava sigurnost svih aplikacija koje koriste navedeni certifikat.

Identifikator aplikacije identificira i opisuje servise koje koristi aplikacija koja se isporučuje. Identifikator aplikacije je definiran pomoću identifikatora paketa (engl. *bundle identifier*), niza znakova jedinstvenog za svaku aplikaciju. Identifikator paketa je najčešće 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 koriste u razvoju programske potpore. Eksplicitni identifikatori jedinstveno identificiraju aplikaciju, na primjer `com.rep.testna_aplikacija` te se koriste za isporuku specifične aplikacije.



Slika 3.4: Postavljanje artefakta za potpisivanje koda

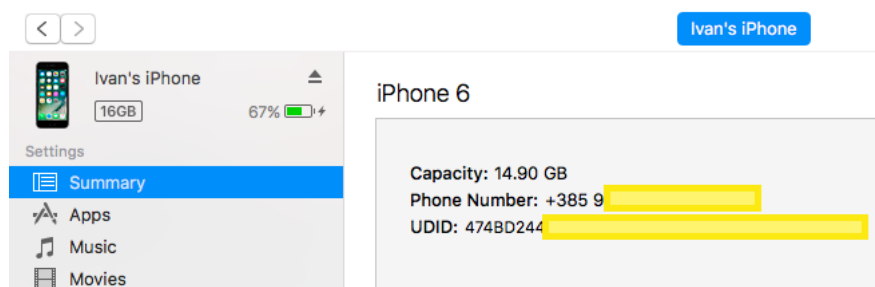
Za kreiranja identifikatora aplikacija je potrebno specificirati ime identifikatora, dodati identifikator paketa i odabrati servise koje aplikacija koristi. Nakon kreiranja je identifikator potrebno preuzeti i pokrenuti korištenjem Keychain Access aplikacije.

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

Novo kreirana artefakte je potrebno preuzeti i pripremiti 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 je potrebno konfigurirati projekt kako bi se u sklopu arhiviranja koristili ispravni artefakti. Otvoriti Xcode projekt, odabrati željen 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 nje-



Slika 3.5: Dohvat UUIDa uređaja korištenjem iTunes aplikacije

govog 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 kreiranje i održavanje certifikati i profila se 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. Artefakte se mogu slobodno distribuirati, ali je njihove privatne ključeve nužno održati tajnim. Do prije nekoliko godina se je cijeli proces odrađivao ručno. Nakon kreiranja artefakata isti su se zajedno 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 se oslanjaju na javan API Apple Developer platforme koji omogućuje kreiranje, modificiranje i dohvat svih artefakata navedenih u prošlom odlomku.

Dodatno, većina alata se 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 je nužno detaljno nadzirati. Preporuča se SSH protokol definirana u 2.2.1 odlomku. Dodatno, korisno je repozitorij pokrenuti unutar lokalne mreže kako on 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 `match` se inicijalizira pozivanjem naredbe `fastlane match init` u početnom direktoriju projekta. Proces inicijalizacije zahtijeva unos lokacije git repozitorija te autorizaciju pristupa repozitoriju. Lokacija repozitorija se zajedno s ostalim parametrima alata sprema u `Matchfile` datoteku gdje je iste moguće modificirati..

Match radi povećanja sigurnosti zahtijeva da SSH ključ korišten u procesu SSH autentifikacije bude vidljiv samo trenutnom korisniku operacijskog sustava. Skripta 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/{imekljuca}.pub
```

Dodatno, potrebno je konfigurirati SSH protokol kako bi isti koristi željeni ključ. 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/{imekljuca}
```

Alata match sve potrebne certifikate i profile kreira i 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')
```

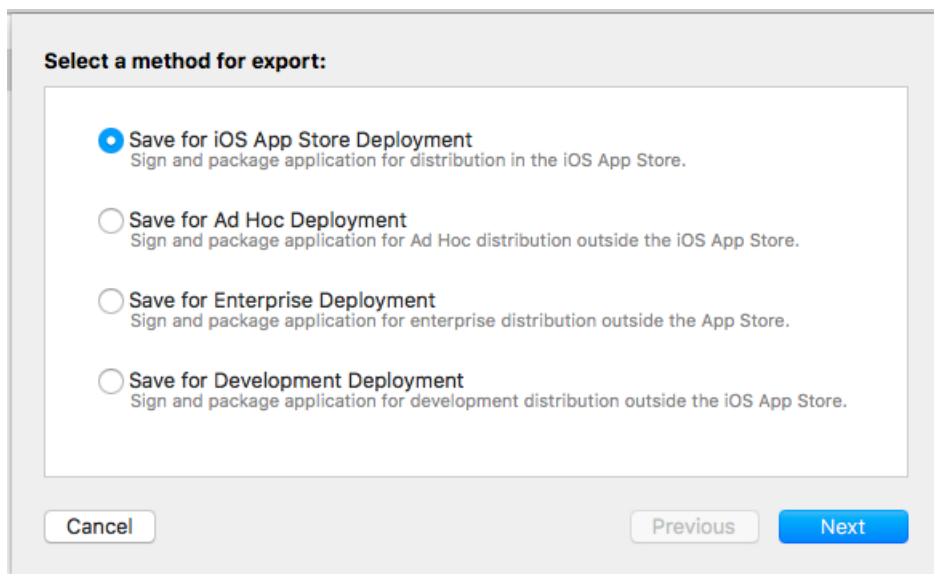
Alat ne obavlja konfiguraciju procesa arhiviranja, međutim, budući da je istu potrebno napraviti samo jednom ona se najčešće obavlja ručno. Pomoću match alata kreirati sve potrebne artefakte odnosno pozvati match naredbe za svaki tip isporuke koja se koristi. Navedene će naredbe kreirati i dohvatiti potrebne artefakte. Zatim otvoriti Xcode aplikaciju i odabrati željenu shemu. Na početnoj, `General` sekciji sheme u odlomku `Signing` za svaku konfiguraciju postaviti odgovarajući pripremljeni profil.

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

Proces arhiviranja se za ad hoc, unutarnju isporuku i App Store ispruku razlikuje jedino u odabiru tipa isporuke. Arhiviranje obavlja alat `xcodebuild`. Međutim, alat je



Slika 3.6: Odabir tipa isporuke u procesu kreiranja arhive

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 odnosno njegov dodatak gym. Proces arhiviranja korištenjem alata xcodebuild je prikazan u dodatku B.

Korištenjem aplikacije Xcode odabrati opciju `Product -> Archive` iz izbornika na vrhu ekrana. Nakon kraćeg perioda se otvara novi prozor koji prikazuje sve dosad kreirane arhive. Jedino što je potrebno je arhivu izdvojiti za objavu. U prozoru odabrati opciju `Export...` Otvara se 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 slijediti upute za dovršenje izdvajanja arhive.

Proces generira dvije datoteke 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 istu direktno instalira na uređaj. Dodatno, ovaj tip arhiviranja je moguće ostvariti isključivo korištenjem Xcode aplikacije. Apple zbog sigurnosnih razloga ne želi objaviti proces direktne instalacije aplikacije na uređaj. Arhiviranje i instalacija se obavlja 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 navedenog automatizaciju arhiviranja ostvarujem korištenjem fastlane dodatka `gym`. Alat za izgradnju interno koristi alat `xcodebuild` i na prvi pogled se 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. Dodatno, 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 se ostvaruje pozivanjem naredbe uz korištenje opcije `archive`. Shema i time cilj na temelju kojeg se izgrađuje arhiva se definira jedna kao i kod drugih opcija naredbe `xcodebuild`.

3.3. Objava

Za dovršetak isporuke je željenim korisnicima potrebno 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 je manifest i arhivu objaviti na generalnoj platformi za dijeljenje podataka kao što su Dropbox i Google Drive. Navedene platforme omogućavaju kreiranje URIa za pojedinu datoteku. Potrebno je kreirati URI za objavljenu arhivu i istu dodati manifestu. Iako je ovaj način isporuke na prvi pogled

jednostavan, on je vremenski vrlo zahtjevan. Svaku novu verziju je potrebno 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 arhivu obavljenu na 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 je moguće 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 je platforma vrlo sličan 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 je trenutno 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 je započet 2011. godine. Platforma je osnovana 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 je potrebno kreirati profil aplikacije na platformi. Kreiranje profila odrađuje Crashlytics biblioteka koju je potrebno dodati i pokrenuti zajedno s aplikacijom.

Prvo, 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 kreirati novu `Run script` 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}
```

API i tajni ključ je potrebno dohvatiti korištenjem Fabric platforme. Otvoriti poveznicu <https://www.fabric.io/settings/organizations>. Nakon prijave odabrati željenu organizaciju te preuzeti ključeve.

Na kraju, dodati naredbu `Fabric.with([Crashlytics.self])` na početak `application(application: didFinishLaunchingWithOptions:)` metode klase `AppDelegate`. Registracija aplikacije će se dogoditi prilikom prvog pokretanja aplikacije. Profil aplikacije se nakon pokretanja aplikacije nalazi na stranici <https://fabric.io/home>.

Nakon kreiranja profila je arhivu i manifest moguće 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 dostavje je potrebno pokrenuti naredbu nakon arhiviranja projekta. Skripta 3.7 prikazuje fastlane stazu koja izgrađuje, arhivira i objavljuje aplikaciju na Crashlytics platformi.

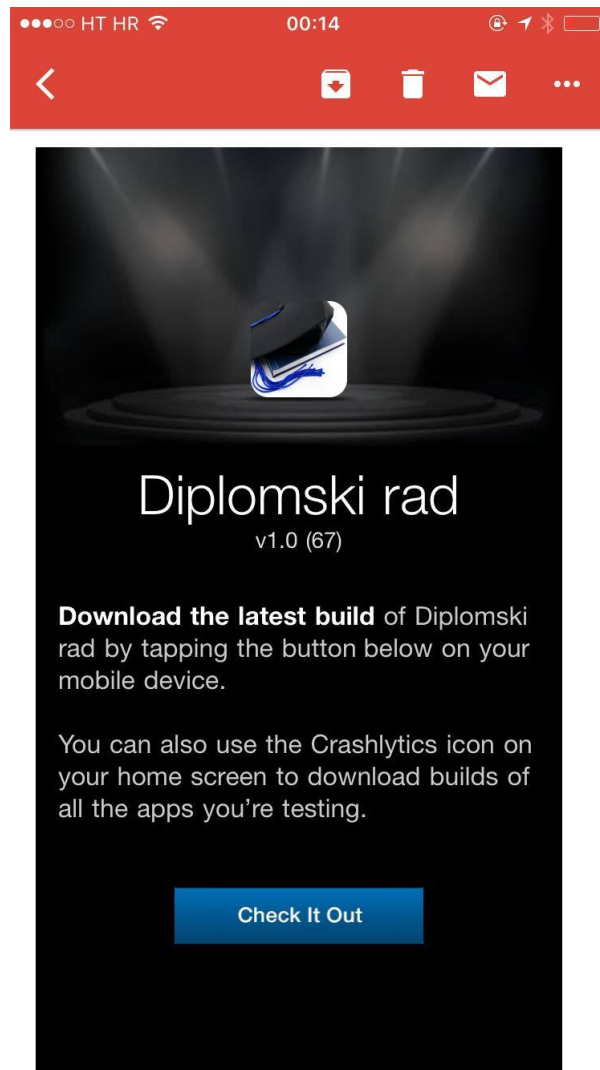
Skripta 3.7: Fastlane staza za isporuku korištenjem Crashlytics platforme

```
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 bi morao unositi crashlytics API i tajni ključ, iste je moguće specificirati kao argumente crashlytics naredbe.

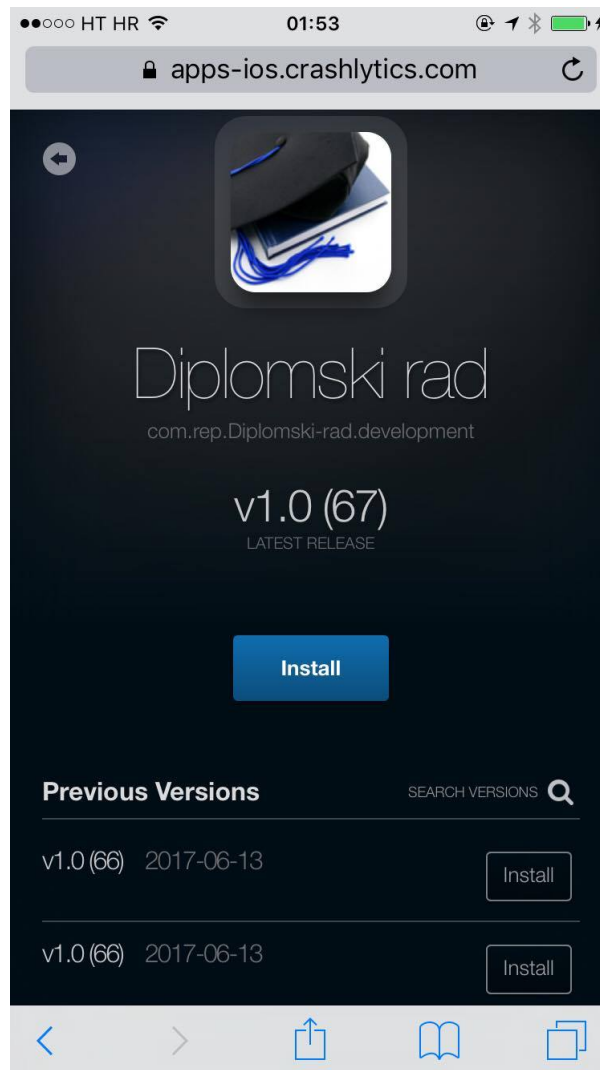


Slika 3.7: Crashlytics e-mail poruka izdana nakon objave nove verzije aplikacije

Nakon isporuke je aplikaciju moguće preuzeti direktno s Crashlytics platforme. Kako bi korisnik mogao preuzeti aplikaciju, potrebno ga je prvo dodati kao testera pojedine aplikacije. Na Crashlytics profilu aplikacije odabrati `Add testers` opciju te unijeti email adresu željene osobe. Na uređaju otvoriti dobivenu e-mail poruku te odabrati opciju `Let Me In`. Slika 3.7 prikazuje dobivenu e-mail poruku.

Crashlytics instalaciju aplikacija objavljenih na Crashlytics platformi obavlja pomoću aplikacije *Beta*. Ako ista nije instalirana poveznica pokreće proces instalacije i konfiguracije pomoćne aplikacije. Slika 3.8 prikazuje ekran za preuzimanje aplikacije. Odabirom opcije `Install` se pokreće instalacija novije verzije aplikacije. Moguće je također preuzeti neku od starijih verzija aplikacije.

Ako je aplikacija isporučena korištenjem ad hoc isporuke, uređaj korisnika mora biti registriran kao testni uređaj te dodan ad hoc profilu. Kako ne bi morali svakoj ver-



Slika 3.8: Preuzimanje aplikacije s Crashlytics platforme korištenjem aplikacije Beta

ziji aplikacije dodavati testere, grupa testera se može dodati kao argument Crashlytics naredbi `groups: {ime_grupe}`.

3.3.2. App Store

Objava javnih iOS aplikacija se ostvaruje 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 se obavlja korištenjem iTunes Connect web stranice. iTunes Connect omogućava kreiranje, modificiranje i praće-

nje objavljene programske potpore. Prije objave je potrebno kreirati profil aplikacije korištenjem iTunes Connect alata.

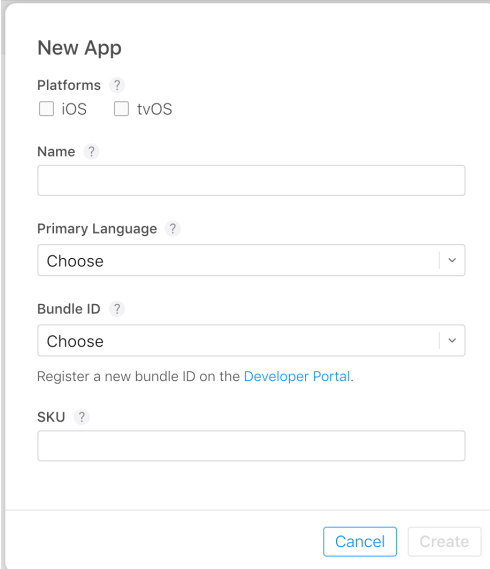
Kreiranje profila se obavlja na <https://itunesconnect.apple.com/> stranici. Nakon prijave je potrebno odabrati opciju My Apps te kreirati novi profil. Slika 3.9 prikazuje formu za kreiranje profila aplikacije.

Nakon kreiranja profila aplikacije je potrebno 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, ona donosi i brojne nedostatke. Ovaj tip isporuke je 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 je potrebno ponovno proći cijeli proces.

Za objavu nove verzije aplikacije je potrebno proći nešto blažu provjeru. Ona može trajati od nekoliko sati do nekoliko dana te najčešće završava odobrenjem promjena. Odbijanje promjena je vrlo rijetko te služi prvenstveno za prevenciju očitog narušavanja Appleovih politika.



The image shows a 'New App' form from the iTunes Connect interface. It contains the following elements:

- Title:** New App
- Platforms:** Two checkboxes for 'iOS' and 'tvOS', each with a help icon.
- Name:** A text input field with a help icon.
- Primary Language:** A dropdown menu with 'Choose' selected and a help icon.
- Bundle ID:** A dropdown menu with 'Choose' selected and a help icon.
- Link:** A text link that says 'Register a new bundle ID on the [Developer Portal](#)'.
- SKU:** A text input field with a help icon.
- Buttons:** 'Cancel' and 'Create' buttons at the bottom right.

Slika 3.9: Forma za kreiranje profila aplikacije na iTunes Connect web stranici

Ovaj proces isporuke je složen i dugotrajan. Iako se u zadnjih nekoliko godina proces značajno poboljšao, veliki broj članova industrije je još uvijek vrlo nezadovoljan. Međutim, Apple zabranjuje isporuku aplikacija za javnu uporabu korištenjem bilo koje druge metode. Ako navedeni proces usporedimo s procesom isporuke web aplikacija, onda su njegovi nedostaci jasno vidljivi. Nova verzija web aplikacije se isporučuje 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. Navedeno iziskuje dugoročnu podršku starijih verzija aplikacije što značajno otežava razvoj.

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

Automatizacija objave Isporuku programske potpore za App Store platformu implementiram pomoću fastlane dodatka `deliver`[9]. Dodatak korištenjem javnog APIa 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 je prikazana u skripti 3.8. Naredba projekt povezuje s iTunes Connect platformom kako bi ubuduće dodatak mogao samostalno obaviti objavu. Za autorizaciju pristupa je potrebno je korisničko ime i lozinku Apple Developer računa. Inicijalizaciju dovršiti unosom iTunes identifikatora profila aplikacije. Identifikator je moguće dohvatiti korištenjem iTunes Connect platforme. Nakon prijave na web stranicu `https://itunesconnect.apple.com` odabрати opciju `My Apps` te odabрати željenu aplikaciju. Slika 3.10 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

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 3.10: iTunes Connect profil aplikacije s označenim identifikatorom aplikacije

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` je moguće 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`

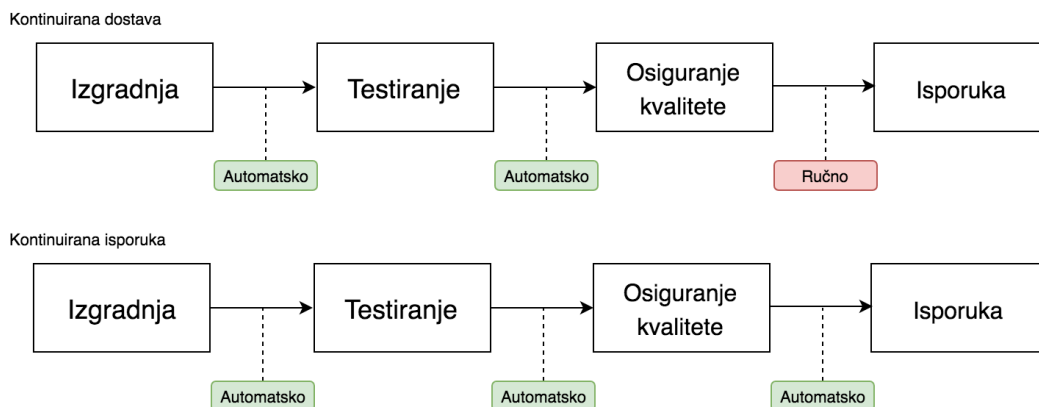
```
lane :release do
  match(type: "appstore")
  gym(scheme: "Production", export_method: "app-store")
  deliver
end
```

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

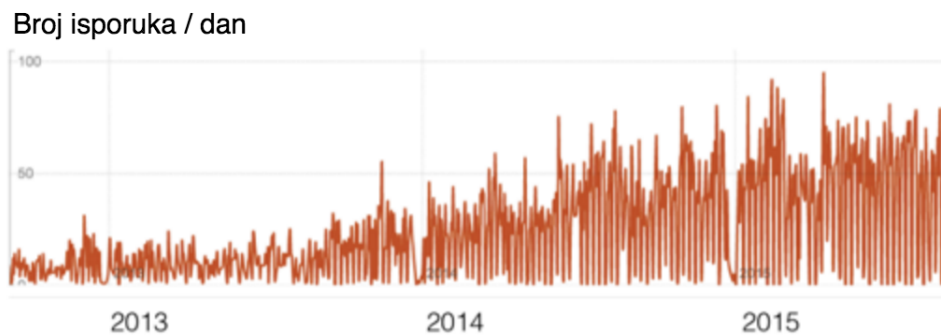
4. Kontinuirana isporuka

Kontinuirana isporuka (engl. *continuous deployment*) je praksa u programskom inženjerstvu koja nastoji smanjiti vrijeme između razvoja i objave funkcionalnosti u produkciju. Kontinuirana dostava automatizira proces isporuke, međutim, ne isporučuje programsku potporu automatski. Svaku isporuku nove verziju programske potpore je potrebno ručno pokrenuti. S druge strane, kontinuirana isporuka automatski isporučuje svaku novu verziju. Razlika između kontinuirane dostave i kontinuirane isporuke je prikazana na slici 4.1[4].

Kontinuirana isporuka automatskom isporukom promjena u produkciju nastoji smanjiti vrijeme između razvoja funkcionalnosti i njene objave u produkciju te samim time ukupno vrijeme trajanja razvoja. Kontinuirana isporuka ova obavlja na dva načina. Prvo, automatizacijom isporuke i automatskom isporukom promjena, kontinuirana isporuka značajno smanjuje vrijeme koje tim mora uložiti za obavljanje isporuke. Druga stavka je direktna posljedica prve. Zbog smanjenja opterećenja koje isporuka nanosi na tim, obavljanje isporuke postaje puno lakše. Lakše obavljanje isporuke dovodi do češćeg obavljanja isporuke. Irska kompanije *Intercom* u svom blogu bilježi stabilan porast broja dnevnih isporuka promjenu od produkcija. Od ispod 20 isporuka u 2012. godini, do preko 80 isporuka u 2015. godini. Slika 4.2 prikazuje broj dnevnih isporuka



Slika 4.1: Usporedba kontinuirane dostave i kontinuirane isporuke



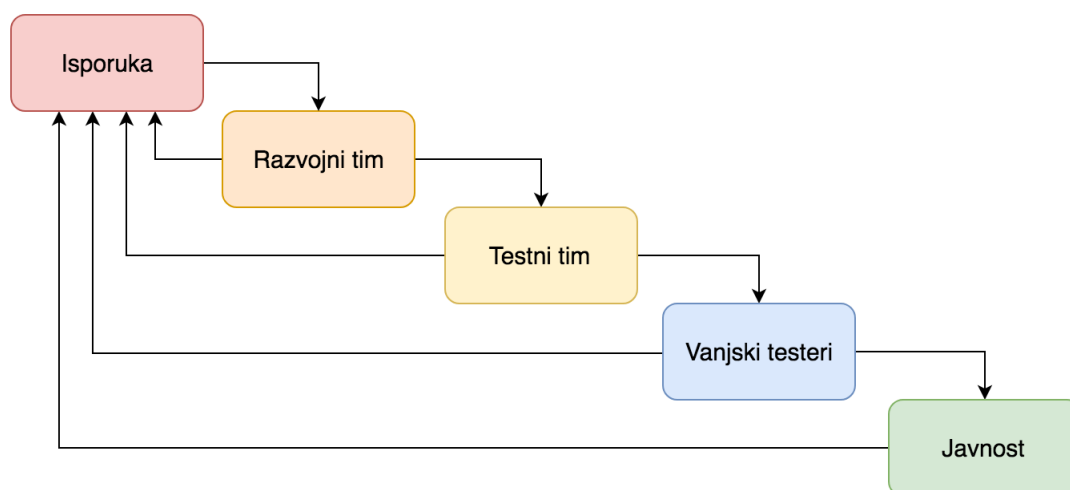
Slika 4.2: Broj dnevnih isporuka promjena u produkciju kompanije *Intercom*

promjena u produkciju kompanije *Intercom*[5].

Iako kontinuirana isporuka nastoji promjene automatski isporučiti u produkciju, navedeno ne znači da se svaka promjena koda automatski isporučuju direktno korisniku. Prvo, svaka promjena prolazi cijeli proces automatskog testiranja i provjere kvalitete. Dodatno, promjene u praksi često prolaze brojna ručna testiranja prije objave u produkciju. Ručna testiranja možemo generalno podijeliti na testiranja razvojnog tima, testiranja tima za osiguranje kvalitete i testiranja vanjskih korisnika koja po broju korisnika dijelimo na *alpha* i *beta* testove.

Dok se *alpha* testiranja provode na vrlo malom broju korisnika, *beta* testiranja mogu sadržavati i do 10

Umjesto da se programska podrška isporuči direktno korisnicima, ona prolazi niz ručnih testiranja. Primjer procesa isporuke produkta različitim skupinama korisnika je prikazana na slici 4.3.



Slika 4.3: Postupna isporuka programske potpore skupovima korisnika

Termin kontinuirana isporuka je prvi puta iskoristio poznati autor Kent Beck u razgovoru o agilnom razvoju. Kent Beck je neisporučenu funkcionalnost usporedio s inventarom koji nije prikazan korisniku. Termin je prvi puta definiran u članku *The Deployment Production Line* autora Jez Humblea, Chris Reada i Dan Northa. Praksa je postala posebno prihvaćena zajednici. Danas praksu razvija i unaprjeđuje uglavnom zajednica web programera koji ju, 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, isporuka iOS aplikacija prvo mora proći provjeru Apple tima te je korisnik mora preuzeti na uređaj. Zbog navedenog se kontinuirana isporuka programske potpore za iOS operacijski sustav značajno razlikuje od tradicionalne kontinuirane isporuke.

4.1. Kontinuirana isporuka za iOS operacijski sustav

Kontinuirana isporuka za iOS operacijski sustav se značajno razlikuje od tradicionalne prakse kontinuirane isporuke. Glavni razlog ove razlike je App Store platforma. Dok se nova verzija web aplikacije može isporučiti u nekoliko minuta, nova verzija iOS aplikacije mora proći službenu provjeru Apple tima te ju potom korisnik mora preuzeti. Dodatno, App Store je zatvorena platforma s ograničenim skupom funkcionalnosti. Iako Apple konstantno dodaje nove funkcionalnosti, platforma je u velikom zaostatku u usporedbi s web razvojem. Navedena ograničenost App Store platforme je glavni razlog zbog kojeg veliki broj poznatih članova iOS zajednice prognozira njenu postupnu eliminaciju. Iako App Store platforma ima navedena ograničenja, još uvijek je moguće automatizirati veliki dio procesa.

Isporuka programske potpore za ograničene skupove korisnika je puno jednostavnija. Budući da za ovaj tip isporuke nije potrebno koristiti App Store platformu, moguće je ostvariti puno veći set funkcionalnosti. U sklopu ovog rada koristim Crashlytics platformu koja pruža brojne funkcionalnosti kao što su automatsko osvježanje i postupna isporuka aplikacije korisnicima.

Vratimo se sada na projekt za koji implementiram kontinuiranu isporuku. U sklopu razvoja isporučujem četiri različite verzije aplikacije pod nazivima: razvoja (engl. *develop*), testna (engl. *test*), pripremna (engl. *release candidate*) i produkcijska (engl.

production) aplikacija.

Razvojna aplikacije prikazuje trenutno stanje aplikacije. Koriste je članovi tima kako bi mogli isprobati novo razvije funkcionalnosti te kako bi mogli pratiti napredak projekta. Svaka završena izmjena treba biti u što kraćem roku vidljiva na ovoj verziji aplikacije.

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

Pripremna aplikacije služi za pripremu aplikacije za izdavanje u produkciju. Sve izmjene obavljene od zadnje verzije se još jednom provjeravaju kako bi izbjegli pojavu grešaka u produkcijskoj aplikaciji. Osim tima za osiguranje kvalitete, pristup ovom tipu aplikacije često imaju i vanjski korisnici. Nova verzija ovog tipa aplikacije se kreira kada je potrebno isporučiti aplikaciju u produkciju.

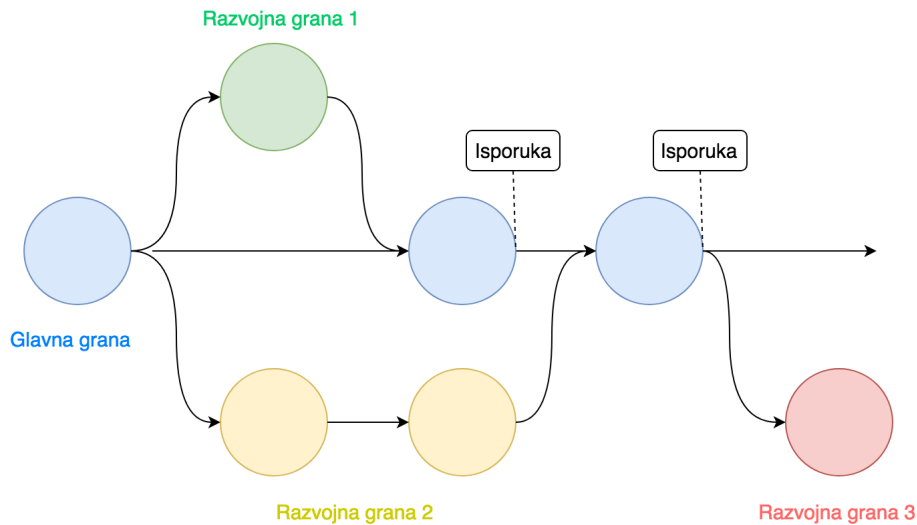
Produkcijska aplikacija je aplikacija trenutno dostupna korisniku. Verzija se osvježava jednom do dva puta u tri mjeseca.

Iz navedenog je jasno da trenutni proces ne slijedi principe kontinuirane isporuke. Između dvije produkcijske verzije može proći i tri mjeseca. Ovaj 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. Pokretanje isporuke

Iako je cilj kontinuirane integracije svaku promjenu što brže isporučiti, nema smisla za svaku promjenu linije koda graditi novu verziju aplikacije. Ne samo da to ne bi bilo resursno isplativo, već većina verzija ne bi prošla ni proces automatskog testiranja i provjere kvalitete. Pametnije je verziju prilikom spremanja promjena u repozitoriju izvornog koda.

Dodatno, u praksi se često jedna funkcionalnost razdvaja na više manjih promjena. Svaka od ovih promjena sama prolazi automatska testiranja i osiguranje kvalitete, međutim često dovodi aplikaciju u neupotrebljivo stanje. Zbog navedenog nema smisla osvježavati verziju kada se ona nalazi u navedenom među stanju. Dodatno, razvojni tim može istovremeno raditi na nekoliko različitih funkcionalnosti.



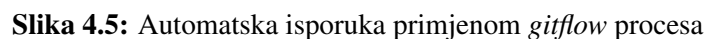
Slika 4.4: Automatska isporuka nakon promjene glavne grane

Za rješavanje navedenog problema se u praksi najčešće koristi proces verzioniranja poznatim nazivom grananje funkcionalnosti (engl. *feature branch*) opisana u odlomku 2.2.1. Umjesto pokretanja procesa isporuke aplikacije prilikom svake promjene repozitorija izvornog koda, proces isporuke se pokreće samo prilikom izmjene glavne grane. Ovaj pristup smanjuje opterećenje sustava te rezultira isporukom upotrebljivog produkta. Dodatno, prilikom spajanja razvojne grane je moguće obaviti dodatne kontrole ispravnosti kao što su timski pregled koda. Slika 4.4 prikazuje proces automatskog pokretanja isporuke nakon spajanja razvojne grane.

Ovaj pristup je vrlo jednostavan te omogućava automatiziranje isporuke promjena na produkciju. Ako projekta ne zahtijeva testnu i pripremnu verziju aplikacije, isporuku u produkciju je moguće ostvariti korištenjem oznaka (engl. *tag*). Oznake dodaju dodatno značenje stanju u sustavu za kontrolu verzija. Umjesto isporuke na temelju novog stanja, isporuku u produkciju je moguće pokrenuti nakon kreiranja oznake.

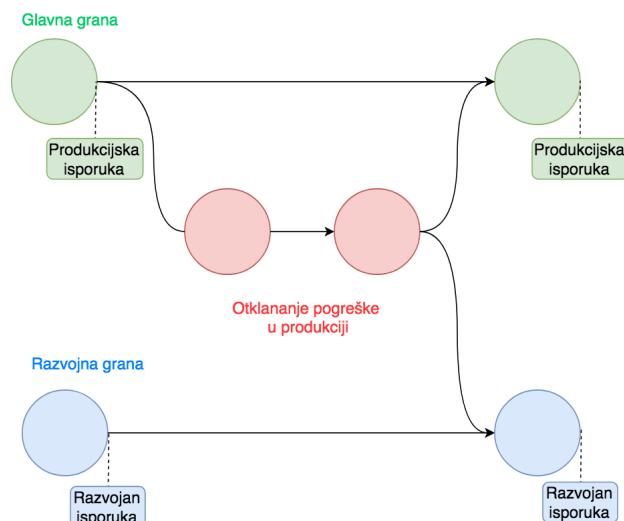
Međutim, ovaj tip automatske isporuke je vrlo limitiran. Glavna mana je limitiranost u broju tipova aplikacija. Za otklanjanje navedenog problema možemo koristiti *gitflow* proces verzioniranja. Umjesto postojanja jedne glavne grane, *gitflow* proces koristi nekoliko grana s posebnim značenjem.

Glavna grana (engl. *master branch*) predstavlja trenutno stanje programske potpore u produkciji. Svaka verzija ove grane je obilježena 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 se obavlja pomoću pomoćne grane koja se kreira iz razvojne grane. Nakon



Za potrebe našeg projekta koristim modificiranu verziju *gitflow* procesa. Uz glavnu i razvojnu granu koristim još dvije posebne grane. Testna grana služi za isporuku testne aplikacije timu za osiguranje kvalitete. U testnu se granu promjene dodaju iz razvojne grane. Nakon pronalaska pogreške iste se otklanjaju direktno na ovoj grani. Nakon potvrde ispravnosti funkcionalnosti započinje testiranje novonastalih funkcionalnosti.

U praksi se često javljaju novi, neočekivani zahtjevi. Jedan od primjera ovog tipa zahtjeva je 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 sljedeći standardnu praksu. Međutim, moguće je granu 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 se nalazi na slici 4.6. Važno je prilikom pojave zahtjeva ovog tipa iskoristiti postojeće procese isporuke.



Slika 4.6: Automatska isporuka otklananja pogreške u produkciji

4.3. Implementacija kontinuirane isporuke

Za ostvarenje kontinuirane isporuke je potrebno pokrenuti odgovarajući proces kontinuirane dostave ako se proces kontinuirane integracije uspješno izvrši. Budući da su kontinuirana integracija i kontinuirana dostava već implementirani, potrebno je samo detektirati uspješnost izvršenja kontinuirane integracije i pokrenuti odgovarajući proces kontinuirane dostave.

Provjeru ispravnog izvršenja kontinuirane integracije obavljam korištenjem varijabli okruženja koje Xcode Server generira u procesu integracije.

Ispravnost izgradnje provjeravam korištenjem varijabli `XCS_ERROR_COUNT`, `XCS_WARNING_COUNT` i `XCS_WARNING_CHANGE`. `XCS_ERROR_COUNT` i `XCS_WARNING_COUNT` varijable definiraju broj pogrešaka i upozorenja generiranih u procesu integracije. Za ispravnost izgradnje broj grešaka mora biti 0. Poželjno je da i broj upozorenja bude 0, ali zbog vanjskih biblioteka to često nije moguće ostvariti. Zbog navedenog se umjesto broja upozorenja često koristim `XCS_WARNING_CHANGE` varijablu koja definira promjenu u broju upozorenja. U sklopu ovog projekta definiram da je izgradnja uspješna ako se broj upozorenja nije povećao.

Ispravnost testiranja provjeravam korištenjem `XCS_TEST_FAILURE_COUNT` varijable. Testiranje smatram uspješnim isključivo ako je broj neuspješnih testova 0.

Ispravnost provjere ispravnosti ovisi o definiranim provjerama ispravnosti. Swiftlint nepoštivanje pravila dojavljuje koristeći standardne pogreške i upozorenja. Trenutno nije moguće razaznati je li pogreška ili upozorenje generirano izgradnjom ili Swiftlint provjerom. Zbog navedenog se držim pravila definiranog u sklopu isprav-

nosti izgradnje. Pokrivenost koda trenutno nije dostupna putem varijable okruženja. Međutim, moguće ju je dohvatiti korištenjem javnog Xcode Server APIja pozivom GET HTTPS metode:

```
/integration/{id}/coverage{?include_methods}
```

Naredba vraća podatke o pokrivenosti koda u JSON formatu koje je moguće iskoristiti za provjeru dovoljne pokrivenosti koda testovima.

Provjeru ispravnost cijele integracije implementiram koristeći sljedeću naredbu:

```
if XCS_ERROR_COUNT == 0 &&  
    XCS_WARNING_CHANGE <= 0 &&  
    XCS_TEST_FAILURE_COUNT == 0  
    {isporuka}  
fi
```

Tip isporuke koji se koristi je moguće definirati direktno u botu koji koristimo, na temelju imena bota ili imena izvorne grane. Definiranje tipa isporuke u samom botu je najjednostavnije, ali uzrokuje razlikom u implementaciji botova. U radu koristim ime bota pohranjeno u varijabli okruženja XCS_BOT_NAME. Na temelju imena odlučujem koji proces isporuke pokrenuti.

```
if XCS_BOT_NAME == '*Test'  
    fastlane test  
elif XCS_BOT_NAME == '*ReleaseCandidate'  
    fastlane release_candidate  
elif XCS_BOT_NAME == '*Production'  
    fastlane production  
else  
    fastlane develop  
fi
```


5. Zaključak

Zaključak.

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] Apple. Code signing, 2017. URL <https://developer.apple.com/support/code-signing/>. [Online; accessed 1-May-2017].
- [3] Atlassian. Comparing workflows, 2016. URL <https://www.atlassian.com/git/tutorials/comparing-workflows>. [Online; accessed 15-March-2017].
- [4] 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].
- [5] 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].
- [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] Open source. Ruby, 2017. URL <https://www.ruby-lang.org/en/>. [Online; accessed 11-Jun-2017].
- [15] supermarin. xcpretty, 2017. URL <https://github.com/supermarin/xcpretty>. [Online; accessed 22-March-2017].
- [16] Wikipedia. Booch method — Wikipedia, the free encyclopedia, 2015. URL https://en.wikipedia.org/wiki/Booch_method. [Online; accessed 10-Feb-2017].
- [17] Wikipedia. Code coverage — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Code_coverage. [Online; accessed 23-Apr-2017].
- [18] Wikipedia. Continuous delivery — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Continuous_delivery. [Online; accessed 28-Apr-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

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.

Appendices

A. Tehnički detalji implementacije kontinuirane integracije, dostave i isporuke

A.0.1. Ručna isporuka

Objava

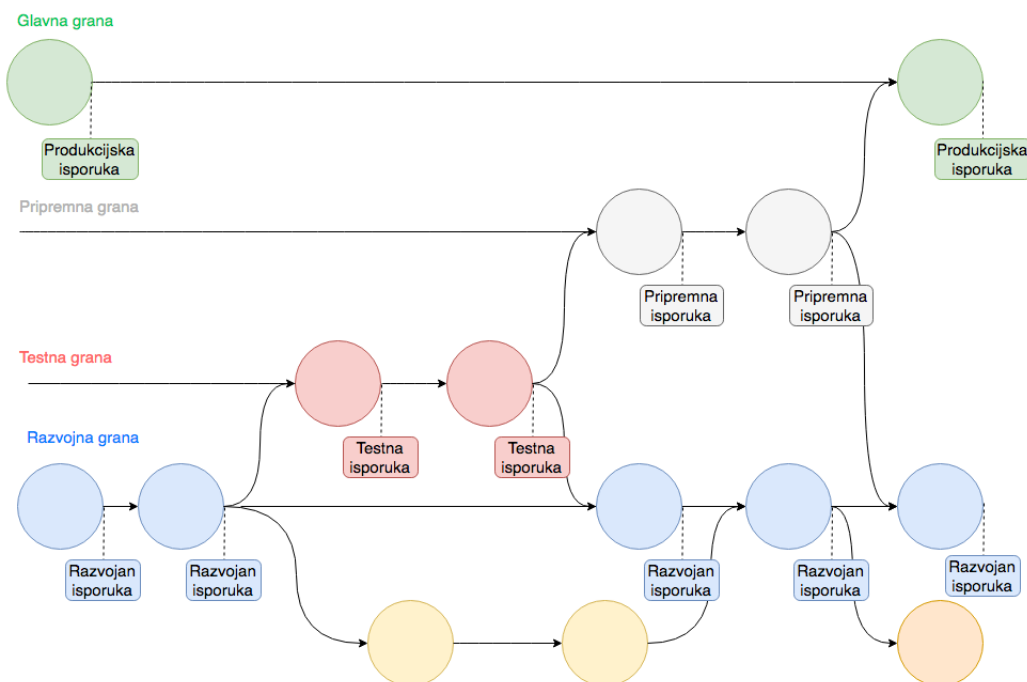
Dostava Kontinuirana isporuka automatski isporučuje novo izgrađene funkcionalnosti. Funkcionalnosti se isporučuju u četiri faze. Prvo se izdaju samo članovima tima, zatim u dva navrata timu za osiguranje kvalitete te na kraju na javno na App Store platformu. Funkcionalnost se prvi puta isporučuje testnom timu radi provjere novo razvijenih funkcionalnosti, a drugi puta radi verifikacije ispravnosti verzije aplikacije koja će se izdati za javnost.

Za pokretanje procesa isporuke koristim modificirani *Gitflow* tok rada. Samu isporuku ostvarujem kombinacijom procesa definiranih u prijašnja dva poglavlja.

A.0.2. Gitflow radni tok

Kako bi ostvario isporuku četiri različitih verzija aplikacija koristim modificirani *Gitflow* tok rada. Standardan *Gitflow* tok rada koristi tri specijalne grane: glavnu, razvojnu i pripremnu granu. Glavna grana predstavlja trenutno stanje projekta dok razvojna grana služi za dodavanje novih funkcionalnosti. Za svaku funkcionalnost koja se razvija se kreira nova grana iz trenutnog stanja razvojne grane. Nakon završetka razvoja funkcionalnosti kreirana grana se spaja u razvojnu granu.

Glavna grana i time cijeli projekt se osvježava spajanjem promjena razvojne grane s glavnom granom. Kako bi se osigurala ispravnost funkcionalnosti koje se dodaju projektu, spajanje razvoje i glavne grane se obavlja korištenjem pripremne grane. Pripremna grana se kreira iz radne grane te se nakon provjere ispravnosti spaja s glavnom



Slika A.1: Modificirani Gitflow radni tok

granom. U slučaju pronalaska pogrešaka iste se otklanjaju na pripremljenoj grani. Pogreške ispravljene na ovaj način je potrebno na kraju dodati i razvojnoj grani. Dodatno, pripremljena grana se može kreirati iz glavne grane kad je potrebno otkloniti pogrešku u trenutnoj verziji aplikacije.

Uz navedene grane koristim i testnu granu koja služi za isporuku testne aplikacije timu za osiguranje kvalitete. Navedena se grana kreira iz razvojne grane. Na grani se otklanjaju pronađene pogreške nakon čega se ista spaja s razvojnom granom.

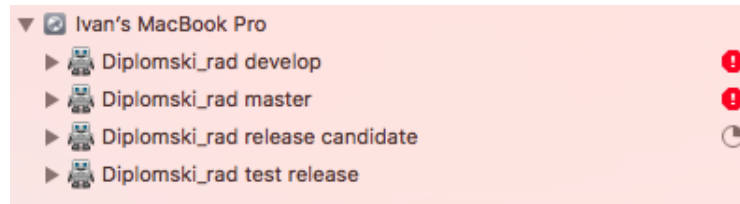
Isporuku produkcijske aplikacije obavljamo s glavne grane, isporuku razvojne aplikacije s razvojne grane, testne aplikacije s testne grane, a pripremljene aplikacije s pripremljene grane. Slika A.1 prikazuje modificirani Gitflow radni tok.

Sada je vrlo jednostavno odrediti koju isporuku treba koristiti. Ukoliko se izmjena nalazi na razvojnoj grani potrebno je osvježiti razvojnu verziju aplikacije, ako se izmjena nalazi na testnoj grani potrebno je osvježiti testnu verziju, u slučaju izmjene pripremljene grane je potrebno osvježiti pripremljenu verziju dok je u slučaju izmjene glavne grane potrebno osvježiti javnu verziju aplikacije na App Store platformi.

Skripta A.1 prikazuje proces kreiranja potrebnih grana.

Skripta A.1: Kreiranje potrebnih grana Gitflow radnog toka

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



Slika A.2: Novo kreirani botovi

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

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

Za svaku granu je potrebno kreirati jedan bot. Najjednostavnije je duplicirati postojeći bot te promijeniti granu na kojoj bot pokreće integraciju. Slika A.2 prikazuje novo kreirane botove.

A.0.3. Konfiguracija projekta

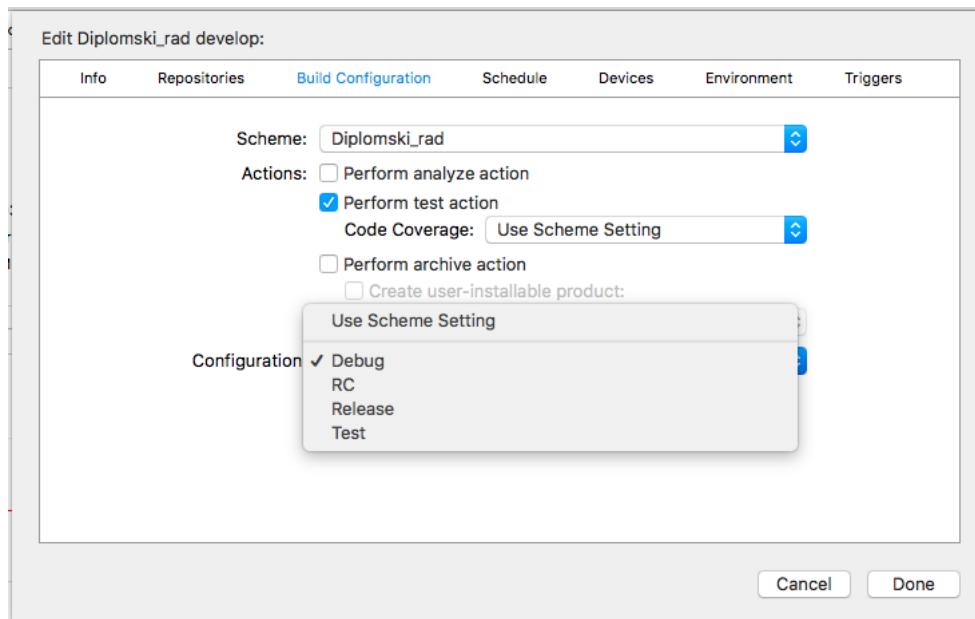
iOS operacijski sustav aplikacije razlikuje na temelju identifikatora aplikacije (engl. *Bundle Identifier*). Kako bi na istom uređaju mogli instalirati sve četiri verzije aplikacije, navedene verzije moraju imati različite identifikatore aplikacije. Navedeno ostvarujem kreiranjem više konfiguracija sheme i korištenjem `.xcconfig` datoteke.

U sklopu projekta kreiram četiri `.xcconfig` datoteke, jednu za svaki tip isporuke. Sadržaj datoteka je prikazan u skripti A.2. Naredba pod #1 dodaje konfiguraciju CocoaPods projekta novo kreiranoj konfiguraciji. Naredba pod #2 postavlja vrijednost identifikatora aplikacije. Svaki od tipova isporuke mora imati jedinstven identifikator. Jedinstvenost ostvarujem korištenjem sufiksa.

Skripta A.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
```

Slika A.3: Odabir konfiguracije u postavkama bota

A.0.4. Automatsko pokretanje isporuke

Za svaki tip isporuke je potrebno kreirati zaseban bot. Navedeno je najlakše ostvariti dupliciranjem postojećeg bota. Botovi se razlikuju u dva pogleda - u konfiguraciji koju koriste te u procesu isporuke koji pokreću.

Konfiguraciju koju projekt koristi je moguće postaviti u konfiguraciji bota. Odabrati opciju *Edit Bot...* -> *Build Configurations* te odabrati odgovarajuću konfiguraciju. Konfiguraciju koju projekt koristi je moguće postaviti u konfiguraciji bota. Odabrati opciju *Edit Bot...* -> *Build Configurations* te odabrati odgovarajuću konfiguraciju.

Za implementaciju isporuke koristim fastlane skriptu definiranu u prijašnjem odlomku. Skriptu pokrećem nakon obavljanja integracije. Odabrati opciju *Edit Bot...* -> *Triggers* te dodati novu fazu nakon integracije. Skripta A.3 prikazuje sadržaj koji je potrebno dodati novo kreiranoj fazi.

Skripta A.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
```

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

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

Promjene obavljene na bilo kojoj od posebnih staza sada pokreću vlastiti proces isporuke.

B. Alat Xcodebuild

Za izgradnju, testiranje i arhiviranje iOS aplikacija koristim *xcodebuild* alat. Alat je razvio Apple za izgradnju macOS aplikacija. U međuvremenu je alat proširen te danas podržava razvoj programske potpore za iOS, tvOS i watchOS operacijske sustave. Xcode i Xcode Server alati koriste *xcodebuild* za obavljanje svih operacija vezanih uz projekt. Iako se u procesu automatizacije nećemo direktno susretati s alatom, korisno je znati što se dešava u pozadini.

Alat je vrlo jednostavan za uporabu. Dovoljno je pokrenuti naredbu *xcodebuild* u početnom direktoriju projekta. Ako u direktoriju postoji samo jedan projekt, naredba pokreće proces izgradnje za predodređenu shemu projekta.

Projekt i cilj se je moguće odabrati korištenjem sljedećih parametara:

```
xcodebuild [-project imeprojekta] [-target imecilja]
```

Shema projekta se odabire *scheme* parametrom:

```
xcodebuild [-project imeprojekta] -scheme imesheme
```

Naredba prima operaciju kao argument. 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

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

B.0.1. Testiranje

Xcode projekt implementira dvije vrste testova: *Unit* i *UI* testove. Oba tipa testa su implementirani kao ciljevi unutar projekta koji pokazuju na cilj aplikacije. Unit testovi služe za testiranje unutarnje implementacije projekta. Ovaj tip testa se pokreće 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 se pokreću na iOS simulatoru. Zbog navedenog je potrebno imati barem jedan simulator prihvatljive verzije operacijskog sustava. Simulator je moguće dohvatiti pomoću Xcode alata. Za prikaz dostupnih simulatora je moguće iskoristiti naredbu:

```
instruments -s devices
```

Testiranje se pokreće naredbom:

```
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 se radi jednako kao i kod izgradnje. Za pokretanje drugog testnog cilja je potrebno kreirati novu shemu te joj kao cilj testne operacije postaviti željeni testni cilj.

B.0.2. 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. Primjer naredbe:

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

Naredba podatke o pokrivenosti koda sprema u `~/Library/Developer/Xcode/DerivedData/{ime_projekta+slučajan_identifikator}/`

Build/Intermediates/CodeCoverage direktoriju. Generirani dokumenti su teško 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 je sličan onome `xcodebuild` alata. Za lakše praćenje pogrešaka i pokretanje naredbe kod svakog procesa izgradnje je moguće Xcode projektu dodati novu Run Script fazu s naredbom:

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

B.0.3. Isporuka

```
xcodebuild -scheme {imesheme} archive
```

Naredba je interaktivna te od korisnika zahtijeva unos nekoliko parametara. Dovoljno je slijediti upute naredbe. Naredba generira dvije datoteke. Arhivu aplikacije s `.ipa` nastavkom i manifest aplikacije s `.plist` nastavkom. Arhiva aplikacije se koristi za instalaciju aplikacije na mobilni uređaj, dok se manifest može koristiti za jednostavnije preuzimanje arhive. Unutar manifesta je moguće specificirati lokaciju arhive. Budući da je manifest vrlo mala xml datoteka, jednostavno ju je moguće podijeliti sa svim potrebnim osobama. Na temelju preuzetog manifesta sustav preuzima povezanu arhivu i instalira aplikaciju.

C. Fastlane

Automatizaciju isporuke programske potpore za iOS operacijski sustav obavljam pomoću alat fastlane. Danas je alat dio Fabric familije koju je u siječnju 2017. godine kupio Google. Fastlane je kolekcija manjih alata od kojih je svaki zadužen za automatizaciju pojedine operacije. Ove alate nazivamo dodaci. Većina dodataka je razvijena od strane zajednice te se kontinuirano razvijaju novi dodaci. Fastlane specificira i jednostavan način ulančavanja i pokretanja dodataka što značajno olakšava automatizaciju. Iako fastlane koristim samo za automatizaciju isporuke, alat podržava i automatizaciju ostalih operacija kao što su izgradnja i testiranje[8].

Instalaciju alata obavljam koristeći Homebrew. Naredba je prikazana u nastavku.

```
brew cask install fastlane
```

Inicijalizacija fastlane alata se obavlja pozivajući naredbu `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 je moguće definirati staze (engl. *lane*). Svaka staza je sastavljena od proizvoljnog broja naredaba koje se izvršavaju slijedno. Dodatno, moguće je definirati naredbe koji se izvršavaju prije ili poslije željenih staza. 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 `Cartfile` ili `Podfile` datoteku, onda fastlane dodaje pozive respektivno `carthage` ili `cocoapods` dodataka.

Primjer staze je prikazan u nastavku.

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

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

Appfile sadrži postavke projekta kao što su korisničko ime i id Apple Developer računa, korisničko ime i ime projekta na Crashlytics platformi te druge varijable koje se koriste u automatizaciji procesa.

Uz zavedene datoteke, fastlane direktorij može sadržavati i brojne druge tekstualne dokumente koji specificiraju postavke pojedinog dodatka. Datoteke slijede istu shemu imenovanja. Prvi dio je skraćena imena dodatka na koju je dodan `file` sufix - `{skraćena}file`.

C.1. Dohvat ovisnosti

Za dohvat ovisnosti koristim alate Carthage i CocoaPods. Fastlane podržava oba alata koristeći dodatke `carthage` i `cocoapods`. Dodaci pružaju jednostavno sučelje prema pojedinom alatu, podržavaju sve potrebne funkcionalnosti te jednostavno formatiraju ispis.

Staza koja pokreće dohvat ovisnosti alatom Carthage je prikazana u nastavku. Naveo sam dvije opcije: `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 uređaju.

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

Za dohvat ovisnosti pomoću `cocoapods` dodatke ne koristim dodatne argumente. Dovoljno je u stazi pozvati `cocoapods` naredbu.

```
lane :cocoapodsCustom do
  cocoapods
end
```

Budući da dohvat ovisnosti želimo pozivati za svaku stazu, navedene se naredbe mogu specificirati u `before_all` stazi.

```
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` je puno jednostavnije i kompatibilnije sa stilom fastlane alata. Dodatno, 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 se obavlja pozivom naredbe `fastlane gym`. Ako projekt sadrži više shema, naredba korisnika traži odabir željene sheme.

C.3. Testiranje

Fastlane testiranje projekta obavlja korištenjem dodatka `scan`[12]. Scan se ponaša slično dodatku `gym`. Operaciju testiranja obavlja korištenjem `xcodebuild` alata. Dodatak samostalno detektira projekte i sheme za koje pokreće testove. U slučaju postojanja više sheme, dodatak korisnika traži odabir željene sheme. Dodatka zatim pokreće sve testne ciljeve koje definira shema.

Dodatak rezultate pohranjuje u `fastlane\test_output\report.junit` datoteci. Junit format je široko prihvaćen način zapisa rezultata testova te postoji veliki broj alata za njegov vizualan prikaz.

D. Usporedba alata za implementaciju kontinuirane integracije