

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1385

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

Ivan Rep

Zagreb, svibanj 2017.

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

SADRŽAJ

1. Uvod	1
2. Kontinuirana integracija	2
2.1. Izgradnja	3
2.1.1. Verzioniranje	5
2.1.2. Implementacija verzioniranja	7
2.1.3. Priprema sustava	10
2.1.4. Upravljanje ovisnostima	11
2.1.5. Izgradnja	11
2.2. Testiranje	13
2.3. Osiguranje kvalitete	14
2.4. Implementacija kontinuirane integracije	15
3. Kontinuirana dostava	19
3.1. Ručna isporuka	19
3.2. Automatizacija isporuke	22
4. Kontinuirana isporuka	24
5. Zaključak	25
Literatura	26
Appendices	29
A. Ručna integracija i isporuka iOS aplikacija	30
A.1. Alati	30
A.2. Priprema	31
A.2.1. Uvod u git	32
A.2.2. SSH ključ	35

A.2.3. Upravljanje ovisnostima	35
A.3. Integracija	37
A.3.1. Izgradnja	37
A.3.2. Testiranje	38
A.3.3. Osiguranje kvalitete	38
A.4. Isporuka	39
B. Xcode Server	43
B.1. Priprema	43
B.2. Kontinuirana integracija	44
B.2.1. CocoaPods	46
B.2.2. Carthage	47
B.2.3. Testiranje i osiguranje kvalitete	48
B.3. Kontinuirana dostava	48
C. Fastlane	50
C.1. Dohvat ovisnosti	51
C.2. Izgradnja	52
C.3. Testiranje	52
C.4. Isporuka	52
C.4.1. Potpisivanje koda	53
C.4.2. Crashlytics platforma	54
D. Usporedba alata za implementaciju kontinuirane integracije	57

1. Uvod

Uvod rada. Nakon uvoda dolaze poglavlja u kojima se obrađuje tema.

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*)[11].

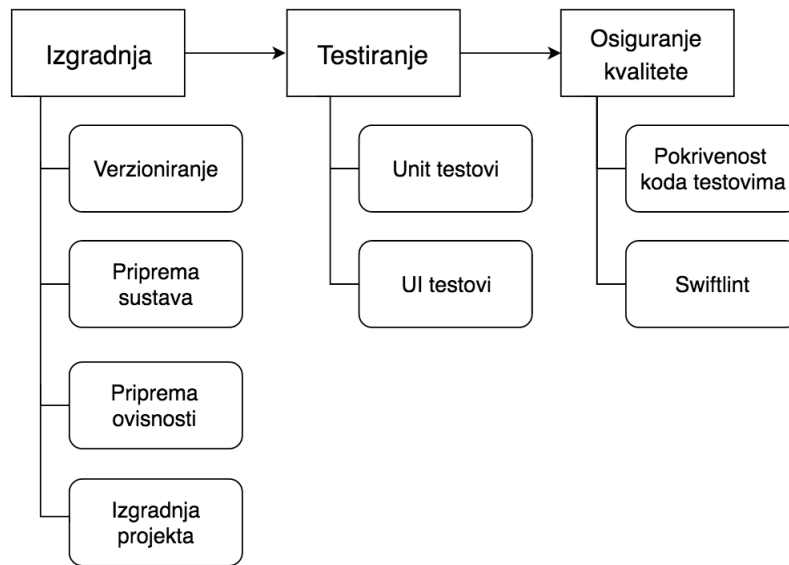
Glavni cilj metode je smanjivanje broja konflikata prilikom spajanja različitih verzija koda. Tijekom razvoja programeri preuzimaju zajedničku (engl. *master*) kopiju izvornog koda (engl. *source code*) te nad njom obavljaju promjene. Lokalnu kopiju izvornog koda nazivamo *razvojnom kopijom izvornog koda*. Svaki programer ima vlastitu razvojnu kopiju izvornog koda.

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 na vrh zajedničke kopije. Međutim, ako je zajednička kopija izvornog koda izmijenjena, onda je potrebno na neki način spojiti lokalne i već spojene promjene.

Čim je duže programerova kopija izdvojena to je veća vjerojatnost izmijene izvorne kopije. 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 modificirale isti dio izvornog koda. Programer u takvom slučaju prije integracije prvo mora preuzeti glavnu kopiju, ručno otkloniti konflikte koje prouzrokuju njegove promjene, te tada obaviti integraciju.

Nakon nekog vremena kopije 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 ova 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.

Danas je kontinuirana integracija standardna praksa u razvoju programske potpore. Međutim, ona se značajno razlikuje od prakse koju je 1991. godine predložio Grady



Slika 2.1: Faze kontinuirane integracije

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.

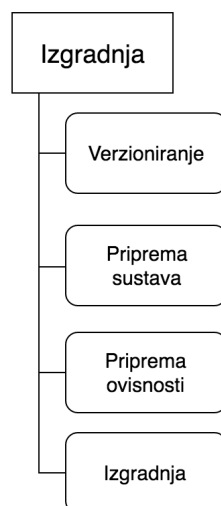
Kontinuiranu integraciju dijelim na tri faze: izgradnju, testiranje i osiguranje kvalitete. Podjelu kontinuirane integracije na faze s podfazama je prikazana na slici 2.1. Svaka od faza je obrađena zasebnim odlomkom u nastavku poglavlja.

Proces verzioniranja se u praksi smatra dijelom procesa izgradnje. Zbog navedenog ću u sklopu odlomka 2.1 razmotriti i problem učestalosti obavljanja integracije.

Zadnji odlomak poglavlja prikazuje proces automatizacije faza korištenjem alata Xcode Server.

2.1. 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 izvorne funkcionalnosti. Kod se prilikom prevođenja često 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



Slika 2.2: Generalna podjela procesa izgradnje

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 na računalu. Danas gotovo niti jedan jezik nije u cijelosti kompajliran ili interpretiran, već koristi kombinaciju obje metode s ciljem optimizacije 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, ovisnosti i postavke projekta. Dohvat i priprema ovisnosti osigurava postojanje ispravnih ovisnosti korištenih u izvornom kodu. Ovisnosti dijelimo na dva tipa, službene ovisnosti koje smatramo dijelom razvojne okoline i vanjske (engl. *third party*) ovisnosti, 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. Proces izgradnje je prikazan na slici 2.2.

Osim kreiranja željenog artefakta, izgradnja provjerava i je li zadana verzija iz-

vornog koda izgradiva. Kod je izgradiv ako se u procesu izgradnje ne desi pogreška, odnosno ako se isti ispravno izvrši. Pogrešku može izazvati neispravnost u izvornom kodu, neispravna konfiguracija sustava, nepostojanje potrebnog alata ili neki drugi nedostatak. Izgradivost sustava je preduvjet za testiranja i isporuku. Samim time je automatizacija izgradnje preduvjet za automatizaciju testiranja i automatizaciju isporuke.

Nastavak odlomka promatra svaki dio izgradnje zasebno. Dodatak A.2 dublje ulazi u tehničke aspekte pojedinog procesa.

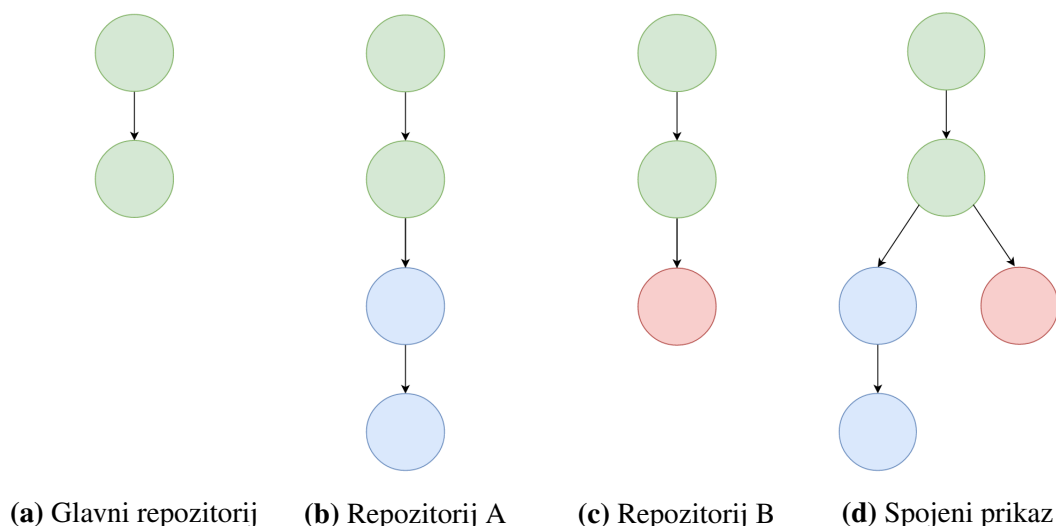
2.1.1. Verzioniranje

Verzioniranje je proces dodjele jedinstvene oznake stanju izvornog koda[15]. Ova oznaka omogućava jedinstvenu identifikaciju pojedinog stanja programske potpore. Verzionirano stanje izvornog koda se naziva verzija.

Uz jedinstvenu oznaku i stanje izvornog koda, proces verzioniranja pohranjuje i dodatne podatke kao što su autor, datum i lokacija kreiranja verzije. Navedeni podaci omogućavaju izgradnju stabla promjena. Ako verzije poredamo kronološki po datumu obavljanja izmjena, onda svaka pojedina verzija ne treba sadržavati cijelo stanje izvornog kod. Dovoljno je samo navesti sve promjene koje su obavljene nakon zadnjeg verzioniranja. Navedeni proces ne samo da značajno smanjuje veličinu repozitorija već olakšava i praćenje izmjena. Slika 2.3 prikazuje primjer stabla promjena. Glavni repozitorij sadrži dvije verzije izvornog. Repozitorij u navedenom stanju preuzimaju dva programera čime kreiraju lokalne repozitorije nad kojima obavljaju izmjene. Globalno stablo promjena se kreira spajanjem stabla promjena pojedine osobe.

U razvoju programske potpore se najčešće koriste dvije različite sheme verzioniranja. Unutarnje, inkrementalno verzioniranje koje se osvježava i nekoliko puta dnevno te vanjsko verzioniranje koje obilježava specifičnu verziju koda, na primjer verziju koda spremnu za objavu. Kod unutarnjeg verzioniranja sam identifikator nije bitan dokle god je on jedinstven. S druge strane, kako vanjsko verzioniranje nosi neko značenje, proces dodjele identifikatora je puno složeniji i ovisi o svrsi koje se pokušava postići. Često se za postizanje vanjskog verzioniranja dodjeljuju posebne oznake (engl. *tag*) unutarnjim verzijama izvornog koda.

Unutarnje verzioniranje koda se naziva kontrola verzija[16]. 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 ne zamisliv bez korištenja jednog od ovih alata.



Slika 2.3: Stablo promjena

Danas se u praksi koriste dva alata: Apach Subversion i git. Apache Subversion, poznat i pod skraćenicom svn, je kreiran 2000. godine u sklopu projekta Apache Software Foundation zajednice. Alat je otvorenog koda, centraliziran, siguran i jednostavan za korištenje. Obično postoji jedan glavni repozitorij kojeg programeri 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 tijek verzioniranja. Bio to centralizirani repozitorij nalik na svnov pristup, pristup s osobama zaduženim za odobravanje promjena, distribuirani model i drugi. Najvažnije, git je jednostavan ali vrlo moćan alat. Implementacija osnovnih funkcionalnosti je jednostavna dok istovremeno postoji podrška za vrlo kompleksne pothvate.

Svn je stariji, međutim još je uvijek široko korišten sustav. Koristi ga veliki broj starijih kompanija i projekata otvorenog koda. Git je značajno dobio na popularnosti u zadnjem desetljeću. Njegova jednostavnost i fleksibilnost ga čine lakšim za upoznavanje i korištenje. Danas git preuzima tržište te se koristi u većini novih projekata. Zbog navedenog u ostatku rada koristim git. Detaljan pregled osnovnih funkcionalnosti alata git se nalazi u dodatku A.2.1. Sve se funkcionalnosti mogu, uz manju modifikaciju, implementirati i u svnu.

2.1.2. Implementacija 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 procesa verzioniranja (engl. *versioning workflows*). Ovaj odlomak obrađuje centralizirani proces (engl. *centralized workflow*), proces grananja funkcionalnosti (engl. *feature branch workflow*), *gitflow* proces (engl. *gitflow workflow*) i proces forkanja (engl. *forking workflow*). Svaki od navedenih pristupa ima svoje prednosti i mane te se koristi u različitim tipovima projekta[3].

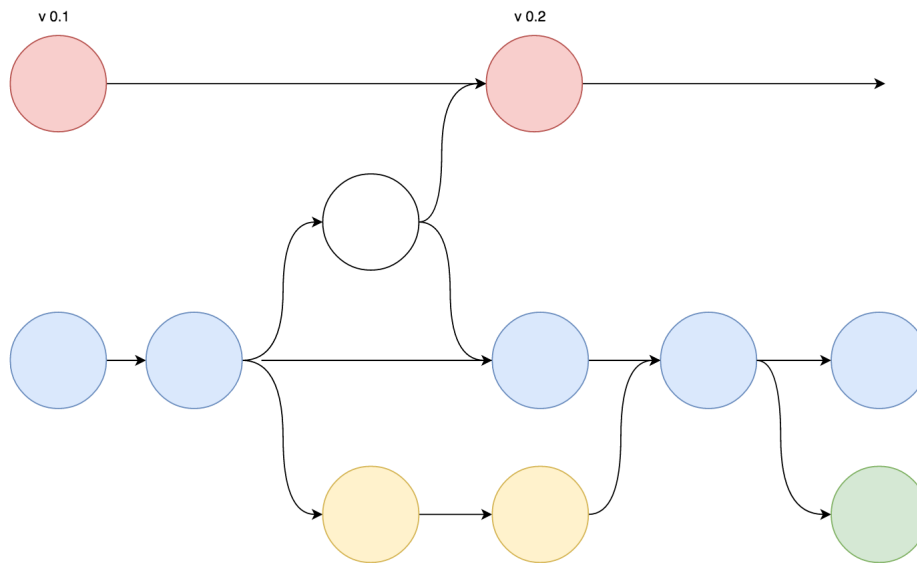
Centralizirani proces koristi jedan glavni i više lokalnih repozitorija. Najčešće se koristi samo jedna, glavna grana. Svaki programer kreira lokalnu kopiju glavnog repozitorija na kojoj obavlja promjene. Nakon obavljanja željenih promjena iste spaja s glavnim granom centralnog repozitorija. Na pojedinom 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 vrlo 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.

Proces grananja funkcionalnosti nastoji otkloniti glavni nedostatak centraliziranog procesa, učestalo preplitanje različitih tokova razvoja. Proces grananja 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 završetku implementacije funkcionalnosti programer granu spaja s glavnim 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*

Gitflow proces također koristi jedan centralni i više lokalnih repozitorija. Za razliku od prijašnja dva procesa, gitflow proces 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.

Navedeni pristup olakšava upravljanje objavom projekta. Buduće da je ključno objaviti ispravan produkt sam proces objave treba biti kontroliran i a produkt temeljito testiran. Izdvajajući proces objave na glavnu granu omogućava istovremeno testiranje produkcijske verzije i nastavak rada na novim funkcionalnostima.

8



Slika 2.5: Primjer gitflow procesa

mer vlastiti javni repozitorij kreira kopiranjem nekog 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 promjena nekom drugom repozitoriju. Navedeni se proces naziva zahtjev za povlačenjem (engl. *pull request*).

Forking proces 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 procesi rjeđe obavljaju integraciju radnih kopija, prednosti koje pružaju nadilaze navedeni nedostatak. Na primjer, gitflow proces integraciju 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 proces. Ovaj proces najbolje odgovara zahtjevima i tipu projekta. Dodatno, gitflow proces 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.

Proces kontinuirane integracije će se obavljati na svim granama. Dodatno, spajanje bilo koje dvije grane neće biti moguće ako rezultat spajanja ne prolazi sve kriterije kontinuirane integracije.

2.1.3. Priprema sustava

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

Dohvat i konfiguracija alata uvelike ovisi o alatima koji se koriste u procesu izgradnje. Cilj je instalaciju i konfiguraciju što većeg broja alata automatizirati kako bi olakšali proces implementacije kontinuirane integracije. Dodatno, automatizacija procesa instalacije i konfiguracije nam omogućava instalaciju alata samo ako je to potrebno.

MacOS je vrlo siguran, i time zatvoren, operacijski sustav. Gotovo svi alati zahtijevaju korisničko dopuštenje te šifru računa koji obavlja instalaciju. Ako se instalacija obavlja za cijeli sustav ili više računa potrebno je instalaciju autorizirati računom s administracijskim privilegijama. Postoji nekoliko načina za automatski upis lozinke, ali navedeni procesi narušavaju sigurnost operacijskog sustava te ne rade za alate s grafičkim sučeljem. Zbog navedenog nije moguće u potpunosti automatizirati proces izgradnje.

Popis alata korištenih u izgradnji se nalazi u dodatku A.1. Priprema alata čiju instalaciju nije moguće automatizirati se nalazi u dodatku B.1 dok je priprema ostalih alata specificirana u ostatku B dodatka.

Xcode projekt, projekt pomoću kojeg implementiram iOS aplikaciju, omogućava specifikaciju postavka izgradnje korištenjem vizualnog sučelja. Postavke izgradnje Xcode sprema u teško čitljivu formatiranu datoteku. S ciljem lakše specifikacije i praćenja postavka izgradnje se mogu iskoristiti konfiguracijske datoteke. Prilikom izgrad-

nje postavke definirane u datotekama nadjačavaju postavke specificirane u projektu.

Za korištenje konfiguracijskih datoteka je dovoljno iste dodati projektu i repozitoriju izvornog koda. Ova je funkcionalnost posebno korisna kad je potrebno izdati više različitih verzija projekta. Na primjer, verziju za korisnika i verziju za testiranje.

2.1.4. Upravljanje ovisnostima

Prije izgradnje projekta je potrebno dohvatiti sve ovisnosti i alate koje projekt zahtjeva. Alat *xcodebuild*, kojeg koristimo za izgradnju programske potpore za iOS operacijske sustave, sadrži sve službene biblioteke i alate potrebne za izgradnju. Međutim, ako projekt koristi vanjske biblioteke iste je potrebno dohvatiti i pripremiti za korištenje. Za dohvaćanje vanjskih biblioteka se u iOS razvoju koriste dva alata: *CocoaPods* i *Carthage*.

CocoaPods je vrlo jednostavan centralizirani sustav za upravljanje ovisnostima. Za dohvaćanje ovisnosti je potrebno samo specificirati iste u datoteci imena *Cartfile*. Alat samostalno kreira i konfigurira novo radno okruženje u koje dodaje osnovni projekt i sve ovisnosti. Jedino što programer mora napraviti je razvoj obavljati u radnom okruženju a ne u osnovnom projektu.

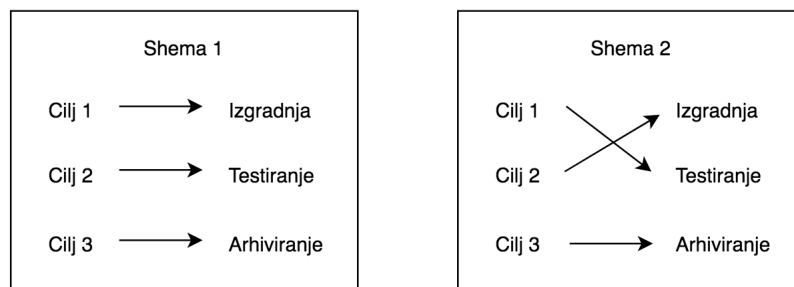
Najveći problem alata je učestala modifikacija radnog okruženja i njegovih konfiguracija. Drugim riječima, alat mijena postavke izgradnje što može uzrokovati neželjeno ponašanje. Dodatno, budući da je alat centraliziran, sve korištene biblioteke moraju biti registrirane u *CocoaPods* sustavu. Navedeno otežava korištenje privatnih biblioteka i biblioteka u razvoju.

Carthage je decentralizirani alat za upravljanje ovisnostima. Sustav omogućava jednostavno dohvaćanje i izgradnju biblioteke. Za razliku od *CocoaPods* alata, *Carthage* ne modificira radno okruženje. Navedeno znači da je ovisnost potrebno samostalno uključiti u projekt ali u istom trenutku otklanja neželjene posljedice koje nosi *CocoaPods*.

Oba sustava se široko koriste te izbor uvelike ovisi o osobnom ukusu. Zbog navedenog u radu koristim oba alata.

2.1.5. 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



Slika 2.6: Xcode projekt s tri cilja i dvije sheme

izgradnju obavlja na temelju Xcode projekta. Prije definiranje 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 implementirati više shema kako bi objedinio operacije za pojedinu distribuciju. Odnos cilja i sheme je prikazan na slici 2.6. 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.

Pri razvoju programske potpore, operacije pokrećemo korištenjem alata Xcode. Međutim, kod automatizacije procesa naredbe možemo pozivati samo korištenjem `bash` ljuske. Zbog navedenog moramo koristiti alat `xcodebuild`. Pokretanje alata je vrlo jednostavno. Cijeli postupak izgradnje projekta pomoću alata `xcodebuild` alata je prikazan u dodatku A.3.1.

Ispis `xcodebuild` alata je vrlo detaljan. Ispis prikazuje sve izvršene naredbe te dojavljuje veliki broj međurezultata. Ovakav oblik ispisa je vrlo informativan ali je istovremeno vrlo teško čitljiv. Xcode Server, alat za automatizaciju koji koristim u ovom radu, već implementira parsiranje ispisa u lako razumljiv prikaz. U slučaju odabiri alata koji ne obavlja parsiranje moguće je iskoristiti jedan od alata namijenjen za par-

siranje xcodebuild ispisa. Trenutno je najpopularniji *xcpretty* alat[10].

Proces izgradnje ne rezultira izvršivim artefaktom već samo provjerava je li sustav izgradiv. Projekt koji je izgradiv je moguće instalirati na uređaju korištenjem xcode-build alata uz specificiranje operacije `install`.

2.2. 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[14].

Pogrešno je mišljenje da implementacija testova produljuje vrijeme razvoja. Svaku implementiranu funkcionalnosti i obavljenju izmjenu je potrebno testirati. Jedino je pitanje hoće li se navedeno testiranje automatizirati ili ne. Jednom napisan kvalitetan test može se pokrenuti beskonačan broj puta. Ručano testiranje, s druge strane, svaki put zahtijeva vrijeme zaposlenika. Bilo to u sklopu razvoja ili s ciljem provjere ispravnosti, ručna provjera ispravnosti zahtijeva više vremena 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.

Alat xcodebuild omogućuje implementaciju 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, cilj može specificirati koji se testni ciljevi pokreću prilikom pokretanja operacije testiranja. Na ovaj način jedan cilj može specificirati više testnih ciljeva te pokretanjem operacije testiranja pokrenuti sve navedene testne ciljeve. Navedeni proces olakšava pokretanje testova korištenjem `xcodebuild` alata.

U sklopu razvoja testiranja pokrećemo korištenjem Xcode alata. Alat obavlja sve potrebne operacije i dojavljuje rezultate procesa. Ručno pokretanje testiranja se također obavlja korištenjem alat `xcodebuild`.

Proces testiranja se pokreće na iOS Simulatoru, aplikaciji koja simulira iOS operacijski sustav na macOS operacijskom sustavu. Zbog navedenog je potrebno preuzeti barem jedan iOS simulator željene verzije.

Primjer naredbe koja pokreće testni proces se nalazi u dodatku A.3.2. Ispis naredbe je sličan ispisu izgradnje te je preporučeno koristiti jedan od parsera ispisa.

2.3. Osiguranje kvalitete

Osiguranje kvalitete služi za provjeru dodatnih restrikcija koje samostalno namećemo na izvorni kod. Navedene restrikcije nastoje poboljšati kvalitetu koda, konačnog produkta i procesa razvoja. Osiguranje kvalitete može sadržavati jednostavna pravila, kao što su zahtijevanje postojanja komentara za svako spajanje, ili vrlo složena pravila koje provjeravaju specijalizirani alati.

Dodatno, moguće je obavljati različite provjere za različite tipove konačnog produkta. Na primjer, produkt koji izdajemo u produkciju mora biti vrlo kvalitetan i strogo istestiran. S druge strane, kod isporuke produkta za testiranje želimo samo isporučiti trenutno stanje programske potpore ma kakva ona bila. Zbog navedenog će ova dva primjera postavljati značajno različite kriterije osiguranja kvalitete.

Navedenu je fleksibilnost moguće ostvariti na nekoliko načina. Jedan od jednostavnijih načina je unaprijed automatizirati procese osiguranja kvalitete za sve željene slučajeve te ih pozivati na temelju neke predodređene oznake, na primjer imena grane na kojoj se obavlja integracija.

U sklopu osiguranja kvalitete implementiram dvije provjere: pokrivenost koda testovima i provjeru 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[12].

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 prikazivanje pokrivenosti koda testovima u alatu Xcode. Uz sam postotak pokrivenosti koda testovima, Xcode prikazuje i pokrivenost pojedinog dokumenta te broj poziva svake pojedine linije koda. Navedene datoteke nisu namijenjene za ljudsko čitanje. Primjer naredbe se nalazi u dodatku A.3.3.

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 pri pisanju koda u Swiftu[4]. Većina pravila se odnosi na izgled i format koda, ali postoje i pravila koja nastoje izbjeći pojavu grešaka. Ne poštivanje pravila izaziva dojavu upozorenja (engl. *warning*) ili greške (engl. *error*) jednake onima procesa izgradnje. Pravila je moguće modificirati, u potpunosti isključiti ili dodati nova.

Swiftlint je vrlo koristan alat. Pridržavanje strogog formata pisanja koda olakšava timski rad, poboljšava čitljivost koda i izbjegava pojavu lako izbjegnutih grešaka.

Alat se pokreće pozivanjem naredbe `swiftlint` u početnom direktoriju projekta. Dodatno, moguće je Xcode projektu dodati novu Run Script fazu koja će alat pozivati prilikom svake izgradnje te će upozorenja i greške alata Swiftlint dojaviti zajedno s ostalim upozorenjima i greškama. Navedenim postupkom izbjegavamo ručnu provjeru ispisa alata. Primjer korištenja alata je dostupan u dodatku A.3.3.

2.4. Implementacija kontinuirane integracije

Za ostvarenje kontinuirane integracije je potrebno automatizirati procese navedene u ovom poglavlju. Automatizaciju procesa možemo ostvariti ručno. Potrebno bi bilo implementirati proces koji osluškuje javni git repozitorij te prilikom detekcije promjene pokreće proces integracije.

Proces integracije bi preuzeo novu verziju izvornog koda, pomoću alata CocoaPods i Carthage dohvatio ovisnosti te pomoću alata `xcodebuild` obavio izgradnju, testiranje i provjeru pokrivenosti koda testovima. Rezultat integracije je moguće formatirati korištenjem alata `xcpretty` te poslati zainteresiranim osobama korištenjem email poruke

ili nekog drugog oblika komunikacije.

Međutim, na tržištu već postoji veliki broj alata koji već implementiraju navedene funkcionalnosti. Dodatno, navedeni alati pružaju i brojne funkcionalnosti kao što su formatirani prikaz rezultata, udaljeno kreiranje i konfiguriranje procesa te testiranje na stvarnim uređajima. Korištenjem ovih alata olakšavamo implementaciju kontinuirane integracije.

Za implementaciju kontinuirane integracije, dostave i isporuke koristim alat Xcode Server. Na tržištu postoji vrlo velik broj sličnih alata. Dodatak D objašnjava zašto sam od svih alata odabrao upravo Xcode Server.

Xcode Server je izgrađen specifično za implementaciju kontinuirane integracije za Xcode projekte. Zbog navedenog je proces implementacije značajno jednostavniji u usporedbi s većinom sličnih alata.

Xcode Server je kombinacija dva alata, Xcodea i macOS Servera. Xcode je alat za razvoj programske podrške za iOS, macOS, tvOS i watchOS operacijske sustave. Alat Xcode već koristim u razvoj programske potpore. macOS Server je alat za automatizaciju procesa na macOS operacijskom sustavu. Prvenstveno je izgrađen za lakšu automatizaciju procesa, ulančavanje poziva skripta i upravljanje udaljenim pristupom računalu. Danas brojni alati koriste macOS Server za lakše ostvarenje automatizacije. Među njima je i Xcode.

Pomoću alata Xcode kreiramo, konfiguriramo i pratimo automatizirane procese. Procesi se pohranjuju i izvršavaju na macOS Serveru. macOS Server na kojem se pokreću procesi može biti na lokalnom ili udaljenom računalu.

Oba alata razvija Apple zbog čega su aplikacije usklađene te su nove funkcionalnosti dostupne na dan njihovog izdavanja. Dodatno, inzistiranje na jednostavnosti korištenja, bogatstvo funkcionalnosti i laka proširivost čine ovaj alat jednim od najboljih za ostvarenje kontinuirane integracije, dostave i isporuke za razvoj iOS aplikacija.

Glavni manjak Xcode Servera je ograničenost na isključivo Xcode projekte. Kako razvoj mobilnih aplikacija gotovo uvijek uključuje iOS i Android operacijske sustave, timovi se češće odluče na implementaciju zajedničkog rješenja. Dodatno, kako je svijet mobilnih aplikacija vrlo mlad, većina se timova još uvijek drži općih rješenja kao što su Jenkins ili CircleCI.

Za početak implementacije kontinuirane integracije je potrebno preuzeti alate macOS Server i Xcode. Alate je moguće preuzeti korištenjem aplikacije App Store. Cijena macOS Servera je \$25 ali je besplatan za osobe s Apple Developer računom. Navedeni račun je potreban i za izgradnju iOS aplikacija zbog čega ga već ima većina iOS programera.

Nakon instalacije je potrebno povezati alate. Povezivanje se ostvaruje odabirom opcije Xcode u macOS Server alatu. Odabir opcije otvara novi prozor u kom je potrebno locirati i odabrati željenu verziju Xcode alata. Ovime je kreiran alat Xcode Server.

Moguće je odabrati račun operacijskog sustava na kojem će se izvršavati integracije. Zbog sigurnosti je preporučeno Xcode Server pokrenuti na zasebnom račun koji nema administrativna prava. Budući da se Xcode Serveru može pristupiti iz vanjske mreže, važno je ograničiti prava koja alat posjeduje. Izdvajanjem procesa na zasebni račun osiguravamo da alat ima samo ona prava koja su mu potrebna. Moguće je kreirati račun proizvoljnog imena, ali je standardno koristiti ime *xcodeserver*. Računu na kojem se obavlja integracija je potrebno dati potrebna prava te osigurati postojanje potrebnih alata.

Budući da želimo automatizirati što veću količinu posla, poželjno je što više potrebnih alate instalirati u sklopu integracije, a ne zahtijevati njihovo prethodno postojanje na računalu. Međutim, što više funkcionalnosti nastojimo automatizirati to se implementacija integracije više komplicira. Dodatno, brojni alati za instalaciju zahtijevaju posebno dopuštenje i interakciju korisnika. Instalaciju takvih alata nije moguće automatizirati bez značajnog narušavanja sigurnosti operacijskog sustava.

U procesu integracije koristimo tri alata za olakšavanje dohvaćanja i konfiguracije drugih alata: ruby, gem i Homebrew. Alati ruby i gem su dostupni na svim instalacijama macOS operacijskog sustava. Međutim, ako su lokacije njihove instalacije promijenjena, onda je potrebno istu dodati u `PATH` varijablu računa operacijskog sustava ili izmijeniti skripte kako bi iste pokazivale na ispravnu lokaciju. Alat Homebrew je potrebno ručno dohvatiti i konfigurirati. Proces je prikazan u dodatku B.1.

Za olakšavanje implementacije kontinuirane integracije koristim alate CocoaPods, Carthage i Swiftlint. Instalacija i konfiguracija navedenih alata je prikazana u dodatku B.

Automatizaciju izgradnje, testiranja i osiguranja kvalitete implementiram korištenjem alata `bot` koji je dio Xcode alata. Bot se kreira u alatu Xcode odabirom opcije `Product -> Create bot...` Za kreiranje je potrebno specificirati repozitorij izvornog koda te odabirati opcije integracije. Sve potrebne funkcionalnosti su već implementirane te je potrebno samo uključiti njihovo korištenje. Proces kreiranja bota je prikazan u dodatku B.2.

Budući da dohvat ovisnosti implementiram korištenjem alata CocoaPods i Carthage, instalaciju, konfiguraciju i pokretanje alata je potrebno samostalno automatizirati. Za automatizaciju naredba možemo iskoristiti funkcionalnost `bot` alata, akciju

koja se obavlja prije izgradnje (engl. *pre-build action*). Ove se akcije pozivaju u sklopi integracije nakon dohvaćanja izvornog koda a prije obavljanja izgradnje.

Sve naredbe koje se pozivaju u sklopu akcije je korisno izdvojiti u zasebnu skriptu. Drugim riječima, akcija jednostavno poziva skriptu koja zatim definira naredbe koje se izvršavaju. Navedeni pristup olakšava izmjenu i nadopunu naredba te ne zahtijeva izmjenu procesa integracije.

3. Kontinuirana dostava

Kontinuirana dostava je praksa koja nastoji automatizacijom procesa isporuke programske podrške omogućiti čestu i pouzdanu isporuku ispravnog produkta u produkciju. Automatizacija procesa isporuke značajno smanjuje količinu vremena koje tim mora uložiti na proces i time potiče češće izdavanje. Dodatno, implementacija jednostavnog i automatiziranog procesa isporuke smanjuje mogućnost pojave ljudske pogreške. Proces u isto vrijeme nastoji smanjiti trošak, rizik i vrijeme uloženo u razvoj te poboljšati ispravnost produkta[13].

Prije isporuke produkta u produkciju je isti potrebno izgraditi i testirati. Jednako tako je prije automatizacije isporuke produkta u produkciju potrebno automatizirati izgradnju i testiranje produkta. Proces automatizacije izgradnje i testiranja se naziva kontinuirana integracija te je definiran u poglavlju 2.

Zbog navedenog kontinuirana dostava obuhvaća proces kontinuirane integracije na koji dodaje automatizaciju isporuke. U praksi se pojam kontinuirana dostava češće koristi kao dodatak na kontinuiranu integraciju. Drugim riječima, pod pojmom kontinuirana dostava se implementira samo automatizacija izdavanja produkta u produkciju, dok se automatizacija izgradnje i testiranja prepušta kontinuiranoj integraciji.

Navedeni pristup koristim u radu. U sklopu kontinuirane dostave implementiram samo automatizaciju isporuke te za procese koji prethode koraku isporuke koristim kontinuiranu integraciju. Dodatno, važno je napomenuti da kontinuirana dostava ne isporučuje produkt automatski, već samo automatizira proces isporuke. Osoba zadužena za isporuku produkta još uvijek mora pokrenuti proces isporuke za željenu verziju produkta.

3.1. Ručna isporuka

Programsku potporu za iOS operacijski sustav možemo isporučiti na nekoliko načina: direktna isporuka (engl. *direct distribution*), *ad hoc* isporuka (engl. *ad hoc distribution*), unutarnja isporuka (engl. *in-house distribution*) i isporuka korištenjem App

Store platforme. Odabir načina uglavnom ovisi o klijentu za koji obavljamo isporuku. Na primjer, za isporuku aplikacije unutar tima možemo koristiti direktnu isporuku. Jednostavno spojimo mobilni uređaj s računalom i instaliramo aplikaciju. Isto naravno nije moguće te ne bi bilo praktično raditi za stvarne korisnike aplikacije. Njima aplikaciju isporučujemo u korištenjem App Store platforme odakle je oni mogu samostalno preuzeti. Primjer implementacije svakog od pristupa je dotupan u dodatku A.4.

Direktna isporuka programsku potporu instalira direktno na uređaj, odakle i dobiva svoje ime. Za obavljanje direktne isporuke je potrebno uređaj povezati s računalnom koje obavlja isporuku USB kabelom. Uređaj je potrebno registrirati na Apple Developer računu kao testni uređaj. Na računalu koje obavlja isporuku je potrebno dohvatiti željenu verziju izvornog koda, dohvatiti potrebne certifikate za izgranjtu te aplikaciju izgraditi i instalirati na uređaju. Izgradnju i instalaciju obavlja već spomenuti alat `xcodebuild`. Instalaciju pokrećemo odabirom operacije `install` i specificiranje uređaja na kojem obavljamo instalaciju. Ako za razvoj koristimo Xcode onda je instalaciju moguće obaviti odabirom uređaja pomoću padajućeg izbornika i odabirom opcije `Run`.

Budući da direktna isporuka zahtijeva direktnu vezu mobilnog uređaja i računala, ona se koristi isključivo za isporuku produkta članovima tima. Dodatno, navedeni tip isporuke ne optimizira produkt. S ciljem ubrzanja izgradnje, alat `xcodebuild` izgradnju može obaviti na nekoliko načina. Ako se način izgradnje ne specificira onda se koristi ne optimiziranja izgradnja. Ovaj tip izgradnje je brz ali rezultira lošijim performansama produkta. Zbog tog navedeni tip isporuke nije preporučeno koristiti za isporuku klijentima ili za prezentaciju programske podrške.

Ad hoc isporuka se koristi za isporuku produkta unutar kompanije u kojoj se obavlja razvoj produkta. Za razliku od direktne isporuke, ad hoc isporuka ne zahtijeva povezanost mobilnog uređaja i računala koje obavlja isporuku. Osoba koja je zadužena za isporuku dohvaća, izgrađuje i arhivira željenu verziju programske potpore.

Arhiviranje je proces kreiranja artefakta koji se koristi za instalaciju aplikacije na korisničkom uređaju. Artefakt se naziva arhiva aplikacije. Za instalaciju aplikacije je uz arhivu potrebna i datoteka koja opisuje i verificira arhivu. Navedene je artefakte potrebno objaviti negdje odakle ih ovlaštene osobe mogu preuzeti i instalirati na mobilni uređaj. Za objavu je moguće koristiti neku od brojnih postojećih platformi ili implementirati vlastito rješenje. Mobilni uređaj na kojem se instalira aplikacija mora biti registriran kao testni uređaj na Apple Developer računu firme. Dodatno, ovaj tip isporuke je optimiziran zbog čega su performanse aplikacije bolje od onih isporučenih korištenjem direktne isporuke.

Ovaj tip isporuke se koristi u slučajevima kad je on jednostavniji ili brži način is-

poruke u usporedbi s direktnom isporukom. Na primjer, isporuka aplikacije udaljenom timu za osiguranje kvalitete. Direktna isporuka aplikacije udaljenom timu nije moguća jer nije moguće doći do uređaja koje tim koristi. Dodatno, ad hoc isporuka omogućava samostalnu instalaciju aplikacije na uređaj. U slučaju postojanja velikog broja uređaja direktna isporuka može biti vrlo zamorna. Omogućavanje instalacije klijentu po potrebi rješava ovaj problem.

Unutarnja isporuka se koristi za isporuku produkta ograničenom skupu vanjskih klijenata. Unutarnja isporuka se od ad hoc isporuke razlikuje u tome da ne zahtijeva prethodnu registraciju mobilnih uređaja kao testnih uređaja. Jednako kao i kod ad hoc isporuke, osoba zadužena za isporuku priprema i objavljuje produkt. Sve osobe s potrebnim pravima pristupa mogu preuzeti i instalirati produkt. Unutarnja isporuka zahtijeva Apple Enterprise račun koji je nešto skuplja verzija Apple Developer računa zbog čega se ne koristi za isporuku unutar kompanije.

Ovaj tip isporuke se koristi kada programska potpora ne smije biti javno dostupna, a u isto vrijeme mora biti lako dostupna određenom skupu osoba. Ovi se zahtjevi najčešće javljaju kod programske potpore koja se koristi u internom poslovanju kompanije. Zabranjeno je aplikacije za javno tržište objaviti korištenjem ovog načina isporuke. Apple strogo nadzire i kažnjava sve aplikacije koje su objavljene na ovaj način.

Za objavu aplikacija koje su javno dostupne korisnicima se koristi App Store platforma. Isporuka započinje slično kao i prijašnja dva tipa isporuke. Osoba zadužena za isporuku priprema aplikaciju za isporuku te izgrađuje arhivu aplikacije. Za razliku od prijašnja dva tipa isporuke, ovaj tip isporuke ne zahtijeva implementaciju vlastitog sustava za distribuciju aplikacije, već se za distribuciju koristi App Store platforma. Nakon objave aplikacije na App Store platformi korisnik ju može pronaći, preuzeti i instalirati korištenjem App Store aplikacije za iOS operacijski sustav koja se nalazi na svakom iOS mobilnom uređaju. App Store pruža i brojne druge prednosti kao što su jednostavno dodavanje novih verzija aplikacije te komentari i ocjene korisnika.

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. Apple vrlo strogo regulira aplikacije koje se nalaze na App Store platformi. Aplikaciju je prvo potrebno registrirati i dodati prvu verziju. Nakon registracije Apple provjerava aplikaciju. Ovaj proces zna trajati od tjedan dana 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.

Ovaj proces isporuke je vrlo složen i dugotrajan. Iako se u zadnjih nekoliko godina 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 ubrzu zamjenu App Store platforme nekim boljim načinom isporuke. Međutim, Apple nije objavio nikakvu naznaku ovog te smo za sad ograničeni sustavom koji imamo.

3.2. Automatizacija isporuke

Direktnu isporuku nije moguće automatizirati zbog dva razloga. Prvo, za obavljanje direktne isporuke mobilni uređaj i računalo koje obavlja isporuku moraju biti direktno povezani što u stvarnom svijetu nije praktično. Čak ako i dedikamo određeni broj mobilnih uređaja specifično za testiranje zaposlenici će ih često koristiti za testiranje te tada neće biti povezani s računalom koje obavlja integraciju. Zbog navedenog pomoću direktne isporuke ne možemo sa sigurnošću tvrditi da se je verzija aplikacije na mobilnom uređaju osvježila. Drugo, automatizaciju direktne isporuke je vrlo teško ostvariti. Apple, zbog sigurnosnih razloga, proces direktne instalacije aplikacije na uređaj drži tajnim. Trenutno je jedini službeno podržani način direktne isporuke istu obaviti korištenjem Xcodea te ovaj proces nije moguće automatizirati.

S druge strane, automatizaciju ad hoc i unutarnje isporuke je vrlo jednostavno os-

tvariti. Kreiranje arhive i manifesta se može obaviti korištenjem xcodebuild alata uz specificiranje operacije `archive`. Novo kreiranu arhivu je potrebno objaviti korisnicima. Proces objave ovisi o načinu distribucije aplikacije. Ako se koristi privatni poslužitelj, onda je dovoljno staru arhivu poslužitelja zamijeniti novom. Alat MDM trenutno nema jednostavan način automatiziranja isporuke. Slično kao i s direktnom isporukom, ne postoji službeno podržani proces. Postoji nekoliko alata razvijenih od strane zajednice koji implementiraju ovaj proces, ali niti jedan nije široko podržan. Budući da smatram ovaj način objave lošijim od ostala dva neću ga detaljnije obraditi.

U sklopu rada koristim treći pristup objave aplikacije, vanjske alate specijalizirane za objavu. Odabrao sam alat Crashlytics, trenutno najpopularniji alat u navedenoj skupini. Alat olakšava objavu i distribuciju aplikacija te omogućava automatizaciju cijelog procesa. Dodatno, postoje brojni alati koji olakšavaju ovaj proces. Za olakšanje automatizacije isporuke koristim alat fastlane.

Za automatizaciju isporuke koristim alat fastlane. Fastlane je sastavni dio alata Fabric koji olakšava cijeliokupni proces razvoja, isporuke i praćenja programske potpore za mobilne operacijske sustave. Razvoj Fabric alata je započeo u novo osnovanoj Crashlytics kompaniji 2011 godine. Kompaniju je u siječnju 2013 godine kupio Twitter te ju je u siječnju 2017 godine preuze Google. Uz podršku ovako velikih imena jasno je zašto je alat popularan.

Fastlane omogućuje automatizaciju velikog broja procesa vezanih uz razvoj aplikacija.

4. Kontinuirana isporuka

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] realm. Swiftlint, 2017. URL <https://github.com/realm/SwiftLint>. [Online; accessed 24-Apr-2017].
- [5] Open source. Fastlane code signing, 2017. URL <https://codesigning.guide/>. [Online; accessed 1-May-2017].
- [6] Open source. fastlane, 2017. URL <https://github.com/fastlane/fastlane/tree/master/fastlane>. [Online; accessed 29-Apr-2017].
- [7] Open source. gym, 2017. URL <https://github.com/fastlane/fastlane/tree/master/gym>. [Online; accessed 1-May-2017].
- [8] Open source. match, 2017. URL <https://github.com/fastlane/fastlane/tree/master/match>. [Online; accessed 1-May-2017].
- [9] Open source. scan, 2017. URL <https://github.com/fastlane/fastlane/tree/master/scan>. [Online; accessed 1-May-2017].
- [10] supermarin. xcpretty, 2017. URL <https://github.com/supermarin/xcpretty>. [Online; accessed 22-March-2017].

- [11] Wikipedia. Booch method — Wikipedia, the free encyclopedia, 2015. URL https://en.wikipedia.org/wiki/Booch_method. [Online; accessed 10-Feb-2017].
- [12] Wikipedia. Code coverage — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Code_coverage. [Online; accessed 23-Apr-2017].
- [13] Wikipedia. Continuous delivery — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Continuous_delivery. [Online; accessed 28-Apr-2017].
- [14] Wikipedia. Software testing — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Software_testing. [Online; accessed 23-Apr-2017].
- [15] Wikipedia. Software versioning — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Software_versioning. [Online; accessed 18-Feb-2017].
- [16] 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. Ručna integracija i isporuka iOS aplikacija

Ovaj dodatak je tehnički usmjeren pregled ručne implementacije procesa integracije i isporuke iOS aplikacije. Glavni cilj dodatka je upoznati se s alatima koje koristimo kod razvoja, integracije i isporuke iOS aplikacija. Sam postupak automatizacija donosi svoje prepreke, zbog čega je lakše prvo definirati postupak koji automatiziramo.

Prvi dio dodatka specificira alate koje koristim u razvoju, integraciji i isporuci te njihovu instalaciju i pripremu. Drugi dio obrađuje integraciju a treći isporuku.

Primjeri su napisani korištenjem operacijskog sustava *macOS Sierra 10.12.4*.

A.1. Alati

Sve naredbe su definirane za *bash* ljusku. Ovoj ljusci je najjednostavnije pristupiti korištenjem *termianl* aplikacije dostupne na svakoj instalaciji macOS operacijskog sustava. Naravno, moguće je koristiti bilo koji drugi tekstualno sučelje s pristupom *bash* ljusci.

Za provjeru postojanja alata koristim sljedeći naredbu. Naredbu ću koristiti za provjeru postojanja alata i izbjegavanje nepotrebne reinstalacije.

```
if which {ime_naredbe} >/dev/null; then
    echo "{ime_naredbe} installed"
else
    echo "{ime_naredbe} not installed"
fi
```

Za razvoj iOS aplikacija koristim alat Xcode. Na tržištu postoji nekoliko sličnih alata, ali je Xcode daleko najpopularniji te ga koristimo i za ostvarenje automatizacije integracije. Alat je moguće instalirati korištenjem App Store aplikacije. Za testiranje procesa koje ću implementirati u ostatku ovog dodatka kreiram jednostavan testni

projekt. Projekt je iOS aplikacija imena *Diplomski_rad* te sadrži jednu shemu istog imena. Projekt sadrži tri cilja: iOS aplikaciju, *unit* i *ui* testove.

Hombrew je alat za instaliranje i upravljanje alatima za macOS operacijski sustav. Alat olakšava instalaciju i konfiguraciju nekoliko alata koje koristimo u sklopu razvoja. Mana ovog alata je zahtijevanje *sudo* pristupa zbog čega instalaciju alata nije moguće automatizirati.

Homebrew je moguće preuzeti korištenjem sljedeće naredbe. Naredba prvo provjerava postojanje Homebrew aplikacije. Ako aplikacija ne postoji, onda ju preuzima sa službene stranice. Ako aplikacija postoji, onda naredba osvježava njeno stanje. Za dohvaćanje alata koristim naredbu *ruby*. Navedena naredba je prisutna na svim trenutnim instalacijama macOS operacijskog sustava.

```
if which brew >/dev/null; then
    brew update
else
    ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
fi
```

Za upravljanje ovisnosti koristim alate CocoaPods i Carthage. Navedene alate dohvaćam respektivno naredbom pod (1) i naredbom pod (2). Naredba pod (1) osvježava postojeću instalaciju ako alat već postoji zbog čega ne provjeravam njegovo postojanje.

```
gem install cocoapods --user-install (1)
```

```
if ! which carthage >/dev/null; then
    brew install carthage (2)
fi
```

Swiftlint je alat za statičku analizu jezika Swift (engl. *linting tool*). Alat je moguće preuzeti korištenjem Homebrew alata.

```
brew install swiftlint
```

A.2. Priprema

Integracija započinje verzioniranjem. U sklopu ovog rada koristim alat git a repozitorij objavljujem na Github stranici. Za kontrolu pristupa repozitoriju koristim *ssh* protokol.

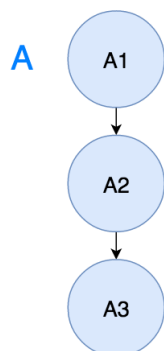
A.2.1. Uvod u git

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 repozitori te sadrži informacije o repozitoriju.

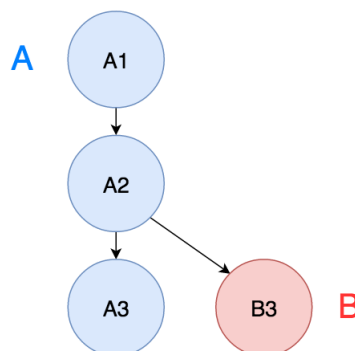
Repozitoriji se mogu klonirati na istom ili drugom uređaju. Klonirani repozitorij je novi repozitorij koji sadržava svu povijest kloniranog repozitorija. Promjene koje se obavljaju u kloniranom repozitoriju nemaju nikakvog utjecaja na izvorni repozitorij. Međutim, promjene obavljene u kloniranom repozitoriju 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 promjena koje su primijenjene na njoj. Modificiranje dokumenata u repozitorija ne uzrokuje samo po sebi stvaranje nove verzije. Obavljene modifikacije je potrebno potvrditi (engl. *commit*). Potvrđivanje modifikacija kreira novu verziju izvornog koda na trenutnoj grani te osvježava njeno stanje. Potvrđivanje promjena je prikazano na slici A.1. Repozitorij koda se stvara kreiranjem glavne grane i obavljanjem inicijalnog potvrđivanja (engl. *initial commit*). Glavna grana je označena slovom A. Inicijalno potvrđivanje je označeno s A1, dok su naknadna potvrđivanja označena s A2 i A3.

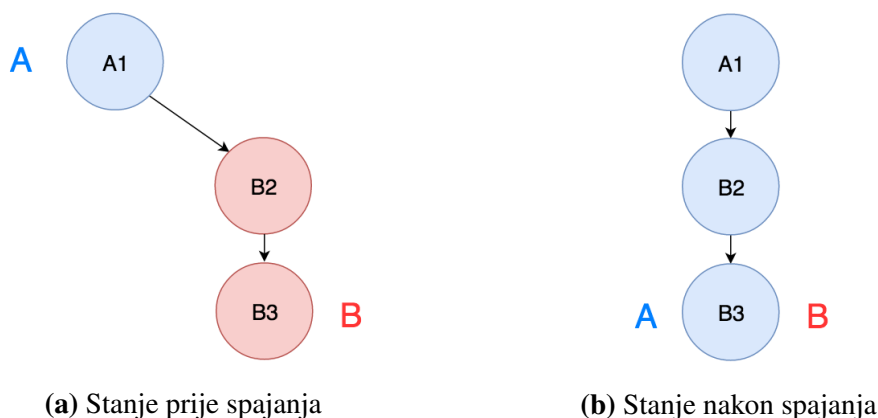
Nova grana se može kreirati iz bilo kojeg stanja postojeće granje. Ovaj se postupak naziva grananje (engl. *branching*). Izvorna i klonirana grana dijele zajedničku povijest do trenutka grananja. Daljnje promjene se primjenjuju samo na izvornu ili kloniranu granu. Slika A.2 prikazuje postupak grananja. Grana B se kreira iz stanja A2 grane A. Grane A i B dijele dva zajednička stanja A1 i A2. Ova stanja nazivamo zajednička



Slika A.1: Grana s tri potvrde



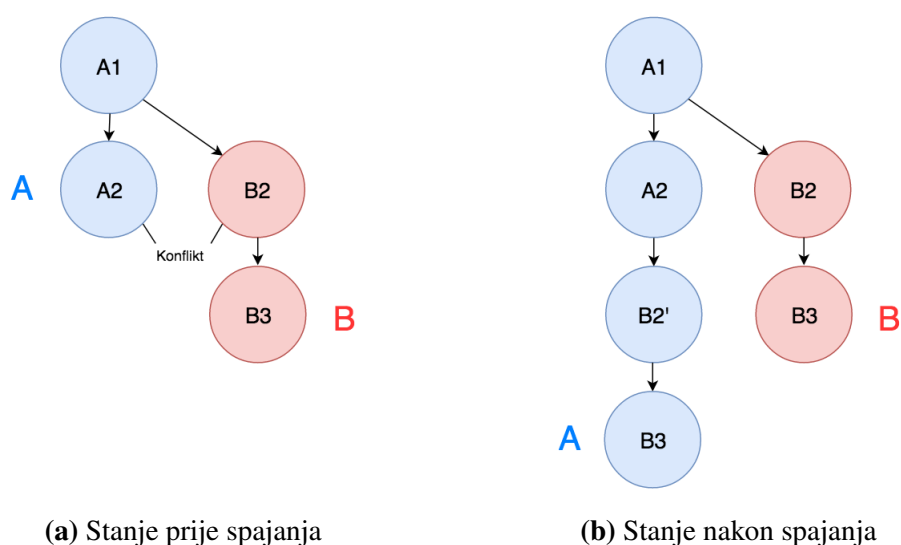
Slika A.2: Grananje



Slika A.3: Spajanja dodavanje promjena

povijest grana. Nakon grananja na granu B dodajemo dva nova stanja B2 i B3.

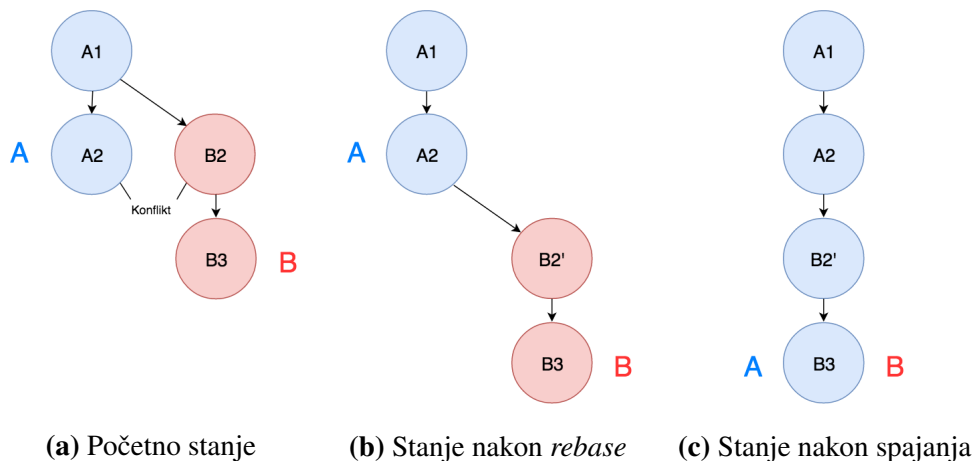
Grane je također moguće spojiti. Spajanje grana dodaje promjene obavljene na izvornoj (engl. *source*) grani u odredišnu (engl. *destination*) granu. Spajanje je moguće obaviti na nekoliko načina ovisno o odnosu dviju grana koje se spajaju. Slika A.3 prikazuje najjednostavniji odnos dviju grana kod spajanja. Nakon grananja grane B iz stanja A2 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*). Slike A.3a prikazuje stanje prije spajanja dok slika A.3b prikazuje stanje nakon spajanja. Dodatno, samo spajanje je moguće označiti dodavanjem dodatnog stanja na granu na kojoj se obavlja spajanje.



Slika A.4: Spajanja otklananjem konflikta

Postupak se komplicira ako je odredišna grana modificirana nakon grananja. U navedenom slučaju nije moguće promjene obavljene u izvorišnoj grani 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 A.3, jednostavno dodavanjem promjena na vrh odredišne grane.

Međutim, ako promjene izazivaju konflikte, onda je te konflikte potrebno ručno razriješiti. Otklanjanje konflikata uzrokuje izmjenom verzija jedne ili obje grane. Proces otklanjanja konflikata se najčešće odrađuje dodavanjem jedne po jedne verzije izvorne 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 A.4 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.



Slika A.5: Spajanje *rebase* postupkom

Gore naveden slučaj se često rješava postupkom pod nazivom *rebase*. Postupak prije spajanja u povijest izvorišne grane dodaje sve verzije nastale u odredišnoj grani nakon grananja. Verzije se dodaju odmah nakon stare točke grananja čime se točka grananja pomiče na zadnju verziju A grane. Slika A.5b prikazuje stanje nakon obavljanja *rebase* postupka na grani B. Sada su grane u stanju jednakom onom na slici A.3 te je spajanje moguće obaviti dodavanjem promjena na vrh odredišne grane. Stanje B2 se još uvijek mijenja, međutim sada je povijest repozitorija linearna.

A.2.2. SSH ključ

Novi ssh ključ se kreira naredbom u nastavku.

```
ssh-keygen -t rsa -b 4096 -C "email@primjer.com"
```

Korisno je ključ dodati ssh agentu kako ne bi morali svaki put unositi šifru ključa.

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

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

Ključ je potrebno dodati na Github. Sljedeća naredba kopira javni dio ključa.

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

A.2.3. Upravljanje ovisnostima

Koristim dva alata za upravljanje ovisnostima - CocoaPods i Carthage. CocoaPods je stariji i široko prihvaćen alat za upravljanje ovisnostima iOS projekata. Alat je centraliziran. Ovisnosti moraju biti unaprijed registrirane na CocoaPods platformi zbog čega je složeno koristiti privatne ovisnosti i ovisnosti koje su još u razvoju. Carthage je noviji alat. Za razliku od CocoaPods alata, Carthage ne izmjenjuje datoteke projekta te je decentraliziran. Ovisnost nije potrebno registrirati te ih je moguće dohvatiti iz bilo kojeg repozitorija.

Kako bi osigurao ispravnost procesa dohvate ovisnosti, projektu dodajem četiri vanjske ovisnosti od kojih je jedna privatna.

CocoaPods CocoaPods alat se inicijalizira korištenjem naredbe u početnom direktoriju projekta.

```
pod init
```

Ovisnosti se dodaju u novokreiranu datoteku *Podfile*. Ovisnosti se dodaju željenom cilju. Ako su ciljevi ugniježđeni, onda se ovisnosti i postavke roditelja odnose i na djecu. Definiram dvije ovisnosti, *LayoutKit* u glavnom cilju i *Nimble* u testnom cilju. Sadržaj datoteke *Podfile* je prikazan u nastavku.


```
use_frameworks!
```

```
target 'Diplomski_rad' do  
  pod 'LayoutKit'  
end
```

```
target 'Diplomski_radTests' do  
  inherit! :search_paths  
  pod 'Nimble'  
end
```

Ovisnosti se dohvaćaju naredbom:

```
pod install
```

Naredba kreira novo radno okruženje pod imenom *Diplomski_rad.xcworkspace*. Radnom okruženju su dodana dva projekta: glavni projekt i novo kreirani *Pods* projekt koji sadrži sve ovisnosti. Daljnji razvoj je potrebno nastaviti korištenjem kreiranog radnog okruženja.

Carthage Korištenjem alata Carthage također dohvaćam dvije ovisnosti ali je jedna od ovisnosti privatna ovisnost podignuta na privatnom *gitlab* poslužitelju.

U početnom direktoriju repozitorija je potrebno kreirati datoteku pod nazivom *Cartfile*. U datoteci se specificiraju sve javne ovisnosti. Privatne ovisnosti je moguće specificirati u *Cartfile.private* datoteci. Sadržaj datoteke *Cartfile* je prikazan u nastavku.

```
git "git@naziv_poslužitelja:ime_projekta.git" "ime_grane"
```

```
github "facebook/facebook-sdk-swift"
```

Ovisnosti se dohvaćaju pokretanjem naredbe:

```
carthage update --platform ios
```

Dohvaćene ovisnosti je potrebno ručno uključiti u projekt. Potrebno je odabrati željeni cilj te u *General* -> *Linked Frameworks and Libraries* sekciju dovući željene biblioteke iz *Carthage/Build* direktorija. U *Build phases* sekciji dodati novu *Run script* fazu s naredbom:

```
/usr/local/bin/carthage copy-frameworks
```

U polje *Input Files* dodati sve željene ovisnosti u obliku:

```
$(SRCROOT)/Carthage/Build/iOS/{ime_biblioteke}.framework
```

A.3. Integracija

A.3.1. Izgradnja

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.

A.3.2. 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.

A.3.3. 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.

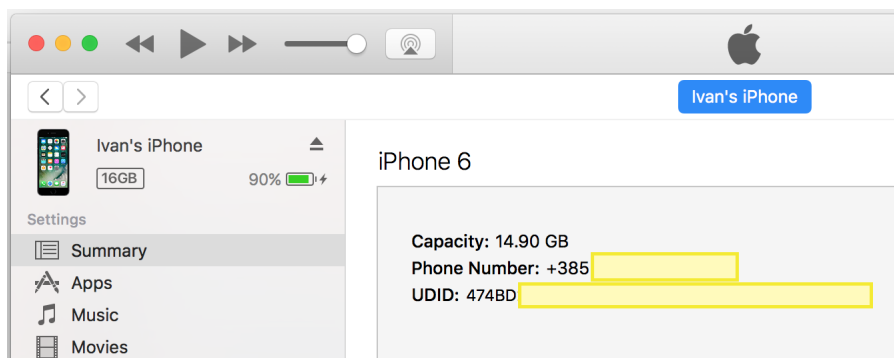
Alat se pokreće pozivanjem naredbe `swiftlint` u početnom direktoriju projekta. Ispis alata je sličan onome `xcodebuild` alata. Za lakše praćenje pogrešaka i pokretanje naredbe kod svakog procesa izgradnje je moguće Xcode projektu dodati novu Run Script fazu s naredbom:

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

A.4. Isporuka

Kao što je navedeno u odlomku 3.1, isporuka programske potpore za iOS operacijske sustave dolazi u četiri tipa: direktna isporuka, *ad hoc* isporuka, unutarnja isporuka i isporuka korištenjem App Store platforme. U nastavku je prikazana ručna implementacija svakog tipa.

Direktna isporuka aplikaciju instalira na mobilni uređaj koji je direktno povezan s računalom koje obavlja isporuku. Uređaj je prije isporuke potrebno registrirati kao testni uređaj na Apple developer računu. Za pristup Apple Developer računu se je potrebno u isti prijaviti na <https://developer.apple.com/> stranici. Nakon prijave odabrati opciju Certificates, Identifiers & Profiles. U lijevom bočnom izborniku odabrati opciju Devices te u gornjem desnom kutu klikom na *plus* ikonu pokrenuti proces registriranja novog testnog uređaja. Za registraciju je potrebno dohvatiti UUID uređaja. UUID se može dohvatiti korištenjem iTunes aplikacije dostupne na macOS operacijskom sustavu. Potrebno je povezati mobilni uređaj s računalom korištenjem USB kabela. U iTunes aplikaciji odabrati povezani mobilni uređaj te kopirati UUID uređaja. Slika A.6 prikazuje lokaciju UUID broja u iTunes aplikacije.



Slika A.6: Dohvat UUIDa uređaja korištenjem iTunes aplikacije

Nakon dohvata UUIDa je isti potrebno kopirati u formu za registriranje testnog uređaja.

Dodatno, prije isporuke aplikacije je potrebno kreirati certifikat za razvoj i certifikat za osobu koja obavlja isporuku. Svaki certifikat se sastoji od privatnog i javnog dijela. Javan dio je slobodno dijeliti te se uvijek može preuzeti s Apple Developer računa. S druge strane, tajni dio certifikata se mora održati strogo tajnim te instalirati samo na uređajima koji zahtijevaju navedeni certifikat. Proces kreiranja i održavanja certifikata je poprilično komplicirao proces razvoja i isporuke. Zbog navedenog je Apple ugradio podršku za automatsko upravljanje certifikatima. Ovim postupkom je gotovo u potpunosti eliminirana potreba ručnog kreiranja i održavanja certifikata. Za uključanje opcije je potrebno odabrati `Automatically manage signing` opciju na željenom cilju projekta.

Certifikate je moguće kreirati ručno korištenjem Apple Developer platforme. Proces je sličan registraciji testnog uređaja. Nakon prijave potrebno je odabrati opciju `Certificates, Identifiers & Profiles -> Identifiers`. Kreiranje certifikata je dobro definirano te je potrebno samo slijediti uputstva.

Instalaciju aplikacije na uređaj ostvarujem korištenjem Xcode alata. Potrebno je odabrati željeni spojen uređaj i pokrenuti `run` operaciju na željenoj shemi. Operacija izgrađuje i instalira aplikaciju na odabranom uređaju. Apple zbog sigurnosnih razloga drži tajnim proces isporuke aplikacije na uređaj zbog čega je navedeni proces moguće ostvariti samo korištenjem Xcode alata.

Implementacija ad hoc i unutarnje isporuke je vrlo slična. Procesi se razlikuju u certifikatima koje koriste i u tome da ad hoc isporuka zahtijeva prethodnu registraciju testnog uređaja. Proces isporuke započinje arhiviranjem aplikacije. Arhiviranje aplikacije je moguće ostvariti korištenjem alata `xcodebuild` uz specificiranje `archive` operacije i željene sheme. Navedena operacija prvo izgrađuje projekt uz korištenje

opcija optimizacije. Primjer naredbe je prikazan u nastavku.

```
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.

Arhivu i manifest je moguće objaviti na nekoliko načina. Osnovan način je korištenjem vlastitog poslužitelja distribuirati potrebne datoteke. Najjednostavnije je napraviti jednostavnu HTML stranicu koja sadrži poveznicu na manifest. Manifest je moguće distribuirati i korištenjem drugih metoda dijeljena kao što su email ili Dropbox, ali je arhivu potrebno objaviti na poslužitelju.

Sljedeći način objave koristi službeni Appleov alat *Mobile Device Management*, *MDM*. Alat je namijenjen za jednostavnu isporuku i distribuciji aplikacija koje nisu namijenjene za javnu objavu. Po funkcionalnosti je vrlo sličan App Store platformi. MDM omogućava jednostavnu objavu aplikacije i novih verzija te omogućava automatsku izgradnju i osvježanje verzije aplikacije. Međutim, alat je vrlo skup te ga koriste gotovo isključivo velike kompanije.

Danas je na tržištu dostupan veliki broj alata koji implementiraju funkcionalnosti slične MDMu za značajno manju cijenu. Najpopularniji od ovih alata je alat Crashlytics. Kompanija je osnovana 2011 godine s ciljem praćenja i dojava pogreška i rušenja mobilnih aplikacija. Kompaniju je 2013 godine kupio Twitter a početkom 2017 kupio Google. Danas alat uz praćenje pogrešaka olakšava distribuciju aplikacija i automatizaciju procesa.

Isporuka aplikacija pomoću Crashlytics alata je vrlo jednostavna. Potrebno je samo registrirati aplikaciju i dodati arhivu. Svi korisnici kojima je aplikacija vidljiva ju mogu jednostavno preuzeti i instalirati na uređaj. Alat je detaljnije obrađen u poglavlju B.3 gdje ga koristim za automatizaciju procesa isporuke.

Za objavu javnih aplikacija je isporuku potrebno obaviti korištenjem App Store platforme. Objava aplikacija na App Store platformu se obavlja korištenjem iTunes Connect alata. Za pristup alatu je potrebno posjedovati Apple Developer račun. Proces započinje arhiviranjem aplikacije i registracijom iste korištenjem iTunes Connect alata.

Nakon prijave u <https://itunesconnect.apple.com/> stranicu potrebno je odabrati opciju Apps. Proces registracije aplikacije je poprilično jednostavno pratiti. Prvo je potrebno specificirati osnovne informacije o aplikaciji kao što su ime, opis i kategorija aplikacije. Nakon toga je potrebno dostaviti sve potrebne grafike koje uključuju nekoliko verzija ikone aplikacije koja je prikazana na početnom ekranu uređaja te slike potrebne za App Store platformu. Na kraju je potrebno dostaviti konačnu verziju aplikacije i time započeti proces provjere aplikacije. Ako proces završi uspješno aplikacija se izdaje na App Store platformu. U suprotnom je potrebno razriješiti razlog odbijanja aplikacije ponovno započeti proces provjere.

B. Xcode Server

Za automatizaciju procesa navedenih u odlomku A koristim alat Xcode Server. Na tržištu postoji veliki broj alata koji omogućuju automatizaciju procesa integracije. Razlog odabira alata Xcode Server objašnjavam u dodatku D.

B.1. Priprema

MacOS je, s ciljem sigurnosti, vrlo zatvorena platforma. Brojni alati zahtijevaju `sudo` lozinku, korisničku interakciju te ih nije moguće instalirati korištenjem ljsuke. Zbog navedenog instalaciju ovih alata nije moguće automatizirati već oni moraju biti unaprijed instalirani na računalu. Među njima su macOS Server, Xcode i Homebrew alati.

```
su - xcodeserver
```

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

Za dohvat brojnih alata koristim *Homebrew* alat. Instalacija alata zahtijeva korištenje računa s administracijskim pravima (engl. *sudo user*). Budući da novokreirani račun nije administrator, proces instalacije je potrebno obaviti na računu koji posjeduje navedena prava. Nakon instalacije je potrebno Xcode Server računu omogućiti korištenje alata i dati pristup `/usr/local` direktoriju u kojem Homebrew obavlja izmjene. Naredba pod (1) instalira alat *Homebrew*, naredba pod (2) kreira `ci` grupu, naredba pod (3) dodaje trenutni i Xcode Server račun grupi, dok (4) daje potrebna prava grupi.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" (1)
```

```
sudo dseditgroup -o create ci (2)
```

```
sudo dseditgroup -o edit -a {current_user} -t user ci (3)
```



```
sudo dseditgroup -o edit -a xcodeserver -t user ci
```

```
sudo chgrp -R ci /usr/local (4)
```

```
sudo chmod -R g+wr /usr/local
```

B.2. Kontinuirana integracija

Kreiranje, konfiguraciju i praćenje automatiziranih procesa obavljam korištenjem alata Xcode. Procesi se pokreću i izvršavaju na macOS Serveru. Računalo na kojem se nalazi macOS Server se naziva poslužitelj. macOS Serveru je moguće pristupiti korištenjem proizvoljnog Xcode alata, sve dok je server vidljiv te postoje odgovarajuća prava pristupa. Za spajanje na udaljeni macOS Server je potrebno pokrenuti lokalni macOS Server i u ponuđenim opcijama odabrati udaljeni poslužitelj.

Automatizacija Xcode procesa se ostvaruje korištenjem *bota*. Bot je namijenjen specifično za automatizaciju Xcode projekata te je kreiranje i konfiguriranje kontinuirane integracije jednostavna u usporedbi s drugim alatima.

Bot se kreira korištenjem Xcode alata, *Product -> Create bot...*. Potrebno je imenovati bot i odabrati macOS Server na kojem će se bot izvršavati.

Bot je potrebno povezati s repozitorijem izvornog koda. Ova veza omogućava pokretanje integracije nakon izmjene stanja repozitorija. Moguće je koristiti lokalno ili javno hostani repozitorij verzioniran alatom git ili svn.

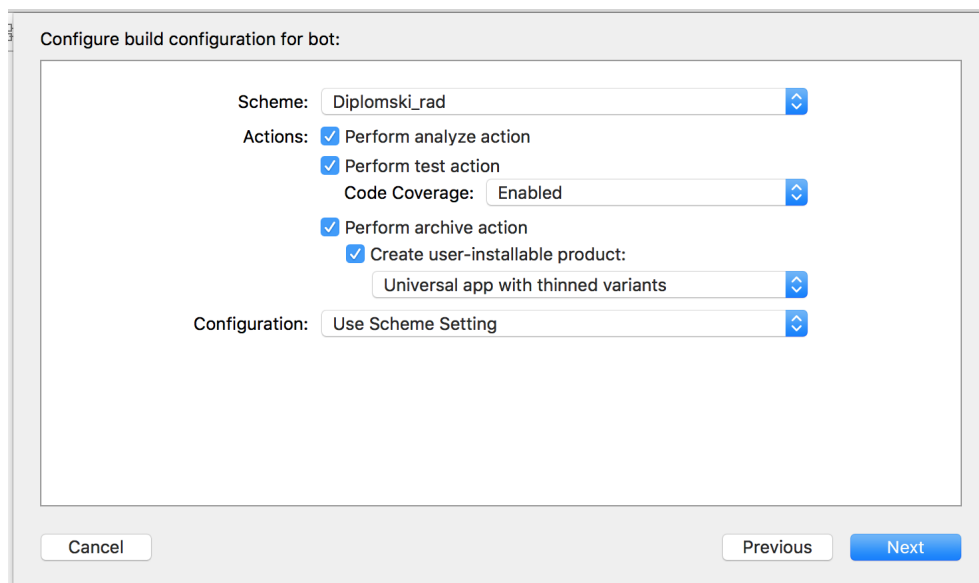
Nakon povezivanja s repozitorijem izvornog koda je moguće konfigurirati opcije integracije. Moguće je odabrati željenu shemu te operacije koje će se izvršavati. Shema mora biti javna da bi se mogla automatizirati. Opcije su prikazane na slici B.1. Testove je moguće obaviti na iOS simulatoru i na stvarnim uređajima povezanim sa serverom.

Integraciju je moguće pokretati periodički, nakon promijene stanja ili ručno.

Dodatno, moguće je konfigurirati okolinu u kojoj će se integracija izvršavati te pomoću *lshl* ljuške specificirati akcije koje se obavljaju prije i poslije obavljanja integracije. Okolina se konfigurira postavljanjem proizvoljnog broja ključ - vrijednost parova. Akcije su proizvoljne skripte definirane i pokrenute na *bin/sh* ljuhci. Obje funkcionalnosti se ekstenzivno koriste za ostvarenje kontinuirane dostave i isporuke.

Integracija započinje odmah nakon kreiranja bota. Proces je moguće ručno započeti odabirom *Integrate* opcije u gornjem desnom kutu.

Prva integracija završava pogreškom koju izaziva ne postojanje potrebnih ovisnosti. Xcode Server kreira privatnu verziju repozitorija izvornog koda te na njemu obavlja integraciju. Problem ovisnosti je moguće riješiti na dva načina.



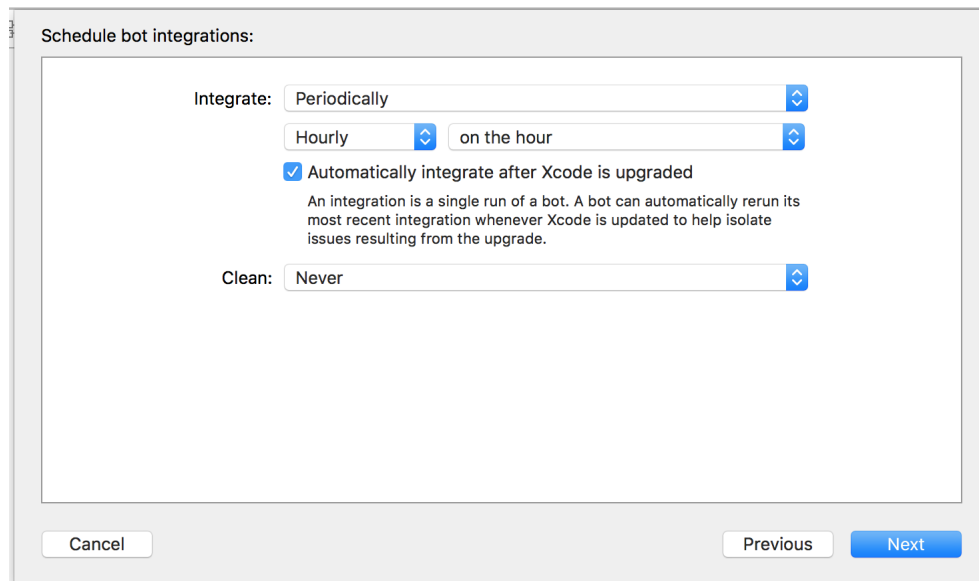
Slika B.1: Konfiguracija osnovnih opcija integracije

Sve potrebne ovisnosti je moguće dodati repozitoriju izvornog koda. Korištenjem navedenog postupka su sve ovisnosti prisutne u repozitoriju odmah nakon preuzimanja repozitorija zbog čega ih nije potrebno dohvaćati. Navedeni postupak olakšava i ubrzava proces integracije. Međutim, dodavanje svih ovisnosti u izvorni repozitorij nosi i značajne probleme. Prvo, repozitorij koda postaje poprilično veći. Pregledom vlastitih projekata ustanovio sam da su ovisnosti od 2 do 50 puta veće od projekta. Čak su i u najvećem projektu ovisnosti bile značajno. Povećanje repozitorija usporava preuzimanje i otežava praćenje promjena. Iako navedeni postupak ima svojih prednosti, generalno se izbjegava u praksi.

Drugi pristup je potrebne ovisnosti specificirati u izvornom repozitoriju te ih dohvatiti po potrebi. Ovaj proces je vremenski zathjevniji od prošlog. Dohvaćanje i ponovna izgradnja svih ovisnosti može značajno usporiti proces integracije. Zbog navedenog je vrlo važno dohvaćati samo potrebne ovisnosti. Ostvarivanje optimalnog ponašanja značajno ovisi o alatu koji koristimo.

Implementacija prvog pristupa je vrlo jednostavna. Potrebno je sve ovisnosti dodati u repozitorij, odnosno maknuti ih iz *.gitignore* dokumenta.

Implementacija drugog pristupa je nešto složenija. Za upravljanje ovisnostima u iOS razvoju se koriste dva alata: CocoaPods i Carthage. Oba alata imaju svoje prednosti i mane zbog čega demonstriram korištenje oba. Detaljan opis oba alata i razlog njihova korištenja se nalazi u dodatku A.



Slika B.2: Konfiguracija perioda izvršavanja integracije

B.2.1. CocoaPods

Instalaciju CocoaPods možemo automatizirati korištenjem akcije koja se ostvaruje u sklopu integracije. Akcije se botu dodaju odabirom opcije Edit bot -> Triggers te pritiskom na *plus* ikonu u donjem lijevom kutu. Moguće je kreirati akcije koje se izvršavaju prije ili poslije integracije. U ovom slučaju odabrati akciju koja se izvršava prije integracije te joj dodati naredbu u nastavku.

```
if which pod >/dev/null; then
    echo "CocoaPods found"
else
    echo "CocoaPods not found, proceeding with installation"

    if which gem >/dev/null; then
        gem install cocoapods --user-install

        pod repo update
    else
        echo "gem not found, quitting CocoaPods instalation"
    fi
fi
```

U slučaju ne postojanja CocoaPods alata, naredba ga instalira i konfigurira. Za instalaciju CocoaPods alata koristim `gem` alat koji je dostupan na svim instalacijama

macOS operacijskog sustava. U slučaju promjene lokacije alata je istu potrebno izmijeniti u naredbi. Naredba `pod` (1) dohvaća i instalira CocoaPods alat. Naredba `pod` (2) dodaje direktorij u kom se nalazi CocoaPods alat u `PATH` varijablu.

CocoaPods ovisnosti definira u `Podfile` dokumentu. Za dohvaćanje i pripremu ovisnosti je dovoljno pozvati naredbu `pod install`. Naredba dohvaća specifičnu verziju ovisnosti definiranu u `Podfile.lock` datoteci. Ako ovisnost odgovarajuće verzije već postoji u lokalnom repozitoriju onda se ista ne dohvaća ponovno.

```
if [ -f Podfile ]; then
    pod install
fi
```

B.2.2. Carthage

Za instalaciju Carthage alata koristim Homebrew alat. Proces je sličan instalaciji CocoaPods alata.

```
if which carthage >/dev/null; then
    echo "Carthage found"
else
    echo "Carthage not found, proceeding with installation"

    if which brew >/dev/null; then
        brew install carthage
    else
        echo "Homebrew needs to be installed"
    fi
fi
```

Carthage ovisnosti specificira u `Cartfile` datoteci. Ako datoteka postoji u direktoriju, ovisnosti se mogu dohvatiti pozivajući `carthage update` naredbu. 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.

```
if [ -f Cartfile ]; then
    echo "Fetching dependencies using Carthage"
```

```
carthage update --platform ios --cache-builds
else
    echo "Skipped fetching dependencies using Carthage"
fi
```

Nakon dodavanja navedene četiri naredbe kao akcije koje se izvršavaju prije integracije projekt bi se trebao uspješno izgraditi. Ispis procesa koje obavlja integracija se nalazi u `Logs` sekciji bota. U slučaju postojanja pogreške korisno je pogledati ispis procesa.

B.2.3. Testiranje i osiguranje kvalitete

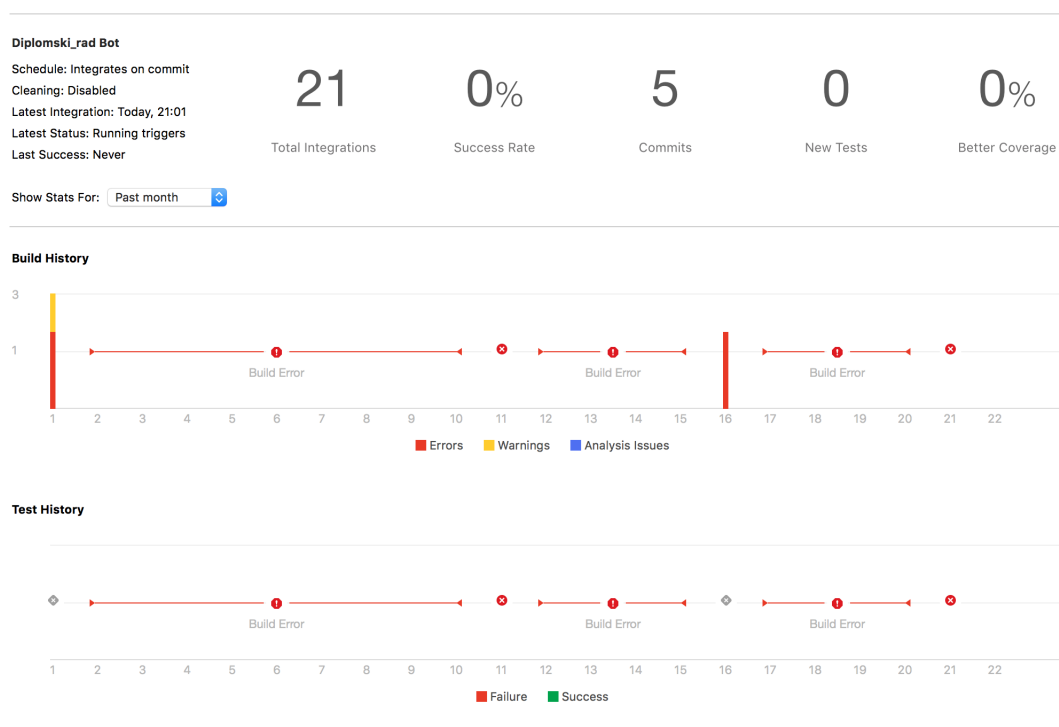
Testovi se pokreću odabirom sheme koja za testnu operaciju koristi željene testne ciljeve i uključivanjem opcije `Perform test action` u postavkama integracije. Predodređeno shema za testove pokreće oba testna cilja (UI i Unit) testove. Za isključivanje pojedinog testnog cilja ili dodavanje novih je potrebno modificirati testne postavke sheme.

Prikupljanje podatka o pokrivenosti koda testovima se uključuje odabirom opcije `Code Coverage -> Enabled` u postavkama integracije.

Ako je poziv `swiftlint` naredbe dodan kao `Run script` faza u projektu, onda se isti poziva prilikom svake izgradnje i nije potrebno obavljati nikakve modifikacije. Ispis naredbe se parsira zajedno s ispisom `xcodebuild` alata te se rezultati pojavljuju zajedno. Ako naredba nije dodana projektu onda je poziv potrebno ručno implementirati nakon obavljanja integracije. Kod korištenja ovog pristupa je potrebno ispis samostalno parsirati.

Rezultate integracije je moguće vodjeti odabirom željenog bota korištenjem `Xcode` alata. Primjer rezultata je prikazan na slici B.3.

B.3. Kontinuirana dostava



Slika B.3: Rezultati procesa integracije

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[6].

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`[7]. 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`[9]. `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.

C.4. Isporuka

U poglavlju 3.1 sam definirao četiri tipa isporuke: direktna isporuka, *ad hoc* isporuka, unutarnja isporuka i isporuka korištenjem App Store platforme.

Kao što je navedeno u poglavlju, direktnu isporuku trenutno nije moguće automatizirati. Dodatno, unutarnja isporuka i *ad hoc* isporuka su vrlo slične. *Ad hoc* isporuka zahtijeva registraciju uređaja na koji se instalira aplikacija kao testnog uređaja na Apple Developer platformi. Unutarnja isporuka nema ovaj zahtjev, međutim, onda je dostupna samo za korisnike s Apple Enterprise računom, skupljom verzijom Apple Developer računa. Zbog navedenog ove dvije isporuke možemo automatizirati istim procesom. Produkt oba tipa isporuke ćemo obaviti korištenjem Crashlytics platforme. Crashlytics je trenutno najbolja i najjednostavnija platforma za objavu programske pot-

pore za iOS operacijski sustav za ograničen skup korisnika. Dodatno, fastlane podržava navedenu platformu što automatizaciju objave čini vrlo jednostavnom.

Fastlane također omogućava automatizaciju isporuke programske potpore za javnost korištenjem App Store platforme.

Prije implementacije isporuke programske potpore je potrebno riješiti pitanje potpisivanja koda.

C.4.1. Potpisivanje koda

Kako bi osigurao autentičnost i ne izmijenjenost aplikacija, Apple koristi sustav potpisivanja koda (engl. *code signing*). Sustav se sastoji od dva dijela: certifikata i profila. Certifikati služe za potpisivanje i kriptiranje aplikacija. Profili se dodaju izvornom kodu te sadrže informacije o aplikaciji i razvojnom timu. Za kreiranje arhive, a samim time i isporuku programske potpore, je potrebno posjedovati odgovarajuće certifikate i profile. Potrebno je imati dva certifikata i jedan profil: certifikat programera koji obavlja isporuku, certifikat aplikacije koja se isporučuju te profil za aplikaciju koji odgovara tipu isporuke koji se obavlja[2]. Postoje četiri tipa profila od kojih svaki odgovara svom tipu isporuke: razvojni, ad hoc, unutarnji i App Store profil.

Navedene profile je potrebno spremati na računalu koje obavlja isporuku u direktoriju `~/Library/MobileDevice/Provisioning Profiles/` a certifikate je potrebno spremati korištenjem `keychain` aplikacije. Certifikate i profile je moguće ručno kreirati i održavati, međutim, u velikom timu se proces može vrlo brzo zakomplicirati.

Zbog navedenog je korisno automatizirati navedeni proces. Automatizacija se može implementirati na nekoliko načina. Trenutno najpopularnije rješenje je sve certifikate i profile držati u jednom tajnom repozitoriju[5]. Dodatno, preporučeno je koristiti samo jedan programerski (engl. *developer*) certifikat po timu. Umjesto kreiranja certifikata za svakog pojedinog člana tima, kreira se samo jedan certifikat kojeg svi članovi koriste za isporuku. Prije obavljanja isporuke potrebno je kreirati certifikat aplikacije te profile za sve potrebne tipove isporuke. Kreiranje certifikata i profile se obavlja korištenjem Apple Developer platforme. Nakon prijave je potrebno odabrati `Certificates, Identifiers & Profiles` opciju odakle je moguće kreirati sve potrebne artefakte.

Nakon kreiranja certifikata i profila je potrebno iste zajedno s tajnim ključevima certifikata dodati u privatni repozitorij i dati prava pristupa članovima tima. Sve podatke je preporučeno enkriptirati.

Umjesto ručnog kreiranja i održavanja certifikata i profila je moguće koristiti fastlane dodatak `match`[8]. Alat samostalno kreira, dohvaća i priprema certifikate i profile čime značajno olakšava proces potpisivanja koda.

Inicijalizacija `match` dodatka se obavlja naredbom `fastlane match init`. Potrebno je unijeti URL repozitorija te autorizirati pristup. URL repozitoriji se zajedno s ostalim parametrima sprema u `Matchfile` datoteku.

Preporučeno je za autentifikaciju repozitorija koristiti SSH protokol. Za automatsko korištenje SSH ključa je isti potrebno dodati alatu `ssh-agent` te podesiti SSH protokol. Za dodavanje ključa `ssh` agentu pokrenuti naredbu u nastavku.

```
chmod 600 ~/.ssh/{imeključa}.pub (1)
```

```
ssh-add -K ~/.ssh/{imeključa} (2)
```

Naredba pod (1) ograničava pristup ključu na samo njegovog vlasnika. Navedeno je potrebno radi sigurnosti ključa. Naredba pod (2) dodaje ključ `ssh` agentu.

Dodatno, potrebno je konfigurirati SSH protokol da koristi željeni ključ. Kreirati novu datoteku `.ssh/config` sa sadržajem:

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

Nakon inicijalizacije dodatka je sve potrebne certifikate i profile moguće dohvatiti automatski. Dovoljno je pozvati `match` dodatak i specificirati tip isporuke koji se koristi. Opcije su `development`, `adhoc`, `enterprise` i `appstore` od kojih svaka odgovara pojedinom tipu isporuke. Naredba u nastavku dohvaća certifikate i profil za ad hoc isporuku.

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

C.4.2. Crashlytics platforma

Crashlytics platformu je moguće iskoristiti za ad hoc i unutarnju isporuku. Prije implementacije isporuke je potrebno registrirati aplikaciju na platformi. Registraciju automatski odrađuje Crashlytics biblioteka koja se dodaje i pokreće zajedno s aplikacijom.

Za dodavanje biblioteke je potrebno preuzeti i slijediti upute Fabric aplikacije dostupne na <https://fabric.io/downloads/apple> stranici. Nakon pokretanja aplikacije odabrati željeni projekt te slijediti upute za dodavanje Crashlytics biblioteke.

Prvo, dodati sljedeće ovisnosti Podfile datoteci:

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

Sljedeće, kreirati novu Run scrip fazu za željeni cilj te joj dodati naredbu u nastavku:

```
"${PODS_ROOT}/Fabric/run" {api_ključ} {tajni_ključ}
```

API i tajni ključevi su dostupni putem Fabric aplikacije.

Zadnje, dodati naredbu `Fabric.with([Crashlytics.self])` u datoteku `AppDelegate`. Registracija aplikacije se odvija prilikom prvog pokretanja aplikacije. Profil aplikacije se nalazi na stranici <https://fabric.io/home>.

Prije isporuke aplikacije je potrebno izgraditi arhivu. Za izgradnju arhive koristim dodatak `gym`. Kod pokretanja `gym` naredbe je potrebno odabrati shemu koja kreira ispravan tip arhive za željeni tip isporuke, te arhivu objaviti korištenjem `crashlytics` dodatka. Nužno je za potpisivanje koda odabrati ispravne certifikate i profil. Staza u nastavku obavlja proces isporuke uz objavu na Crashlytics platformi.

```
lane :beta do
  increment_build_number
  gym(scheme: "Diplomski_rad", export_method: 'ad-hoc')
  crashlytics
end
```

Dodatno, kako prilikom pokretanja naredbe ne bi morao unositi Crashlytics API i tajni ključ, iste je moguće dodati Crashlytics naredbi.

```
crashlytics(
  api_token: {api_ključ},
  build_secret: {tajni_ključ}
)
```

Nakon isporuke je aplikaciju moguće preuzeti direktno s Crashlytics platforme. Kako bi korisnik mogao preuzeti aplikaciju, potrebno ga je prvo dodati kao testera pojedine verzije aplikacije. Na Crashlytics profilu aplikacije odabrati `Add testers` opciju te unijeti email adresu željene osobe. Na uređaju osobe otvoriti poslanu email poruku te odabrati opciju `Let Me In`. Po potrebi instalirati `Beta` aplikaciju pomoću koje se obavlja instalacija. 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 verziji aplikacije dodavati testere, grupa testera se može dodati kao argument Crashlytics naredbi `groups: {ime_grupe}`.

D. Usporedba alata za implementaciju kontinuirane integracije