Životni ciklus programskog proizvoda

Barkijević, Filip

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

FILIP BARKIJEVIĆ

ŽIVOTNI CIKLUS PROGRAMSKOG PROIZVODA

ZAVRŠNI RAD

Pula, 19.rujna 2021.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

FILIP BARKIJEVIĆ

ŽIVOTNI CIKLUS PROGRAMSKOG PROIZVODA

ZAVRŠNI RAD

JMBAG: 0303062572 (4370-E), izavnredni student
Studijski smjer: Informatika

Predmet: Programsko Inženjerstvo
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor:izv. prof. dr. sc. Tihomir Orehovački

Pula, 19.rujna 2021.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Filip Barkijević kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

U Puli, 19.09.2021. godine

Student

IZJAVA
o korištenju autorskog djela

Ja, Filip Barkijević dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom životni ciklus programskog proizvoda koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 19.09.2021. godine

Potpis

Filip Barkijević

SAŽETAK

Životni ciklus razvoja softvera okvir je koji definira skup zadataka koje treba obaviti u svakom koraku procesa razvoja softvera. Životni ciklus pruža strukturu koju razvojni tim organizacija treba slijediti. Ova struktura se sastoji od detaljnog plana koji opisuje kako razviti, održavati i zamjeniti određeni softver. Životni ciklus također definira metodologiju za poboljšanje kvalitete softvera i cjelokupnog procesa razvoja.Životni ciklus softvera najčešće obuhvaća sljedeće faze: analiza i definiranje zahtjeva, dizajn sustava, implementacija programa, testiranje, isporuka i održavanje, a svako od ovih stanja može se promatrati kao nezavisan proces. Postoji veliki broj modela životnog ciklusa razvoja softvera (tradicionalno planski modeli i novije agilne razvojne metode).U industriji informacijskih tehnologija životni ciklus programskog proizvoda ima veliku ulogu. Nijedan proces razvoja softvera nikada neće biti dovršen učinkovito i zadovoljiti zahtjeve klijenata bez primjene životnog ciklusa softverskog proizvoda. To je zato što životni ciklus programskog proizvoda uključuje strukturirani okvir koji opisuje faze uključene u razvoj informacijskog sustava.

Ključne riječi: životni ciklus programskog proizvoda, faze životnog ciklusa programskog proizvoda, tradicionalni i agilni modeli razvoja softvera

ABSTRACT

A software development lifecycle is a framework that defines a set of tasks to be performed in each step of the software development process. The life cycle provides the structure that an organization's development team needs to follow. This structure consists of a detailed plan that describes how to develop, maintain, and modify specific software. The life cycle also defines the methodology for improving the quality of the software and the overall development process. The software life cycle usually includes the following phases: analysis and definition of requirements, system design, program implementation, testing, delivery and maintenance, and each of these phases can be viewed as an independent process. There are a large number of software development lifecycle models (traditional planning models and new agile development methods). In the information technology industry, the software product life cycle plays a major role. No software development process will ever be completed efficiently and meet customer requirements without applying a software product lifecycle. This is because the software product life cycle includes a structural framework that describes the stages involved in information system development.

Key words:software development lifecycle, phases of software development lifecycle, traditional and agile models of software development

SADRŽAJ

1. UVOD

Životni ciklus predstavlja evoluciju sustava, proizvoda, usluge, projekta ili drugog entiteta koji je kreirao čovjek od konceptualne ideje do povlačenja iz upotrebe. U praksi, životni ciklus programskog proizvoda počinje idejom ili identificiranom potrebom za određenim tipom programskog proizvoda, a završava se povlačenjem programskog proizvoda iz uporabe. cilj životnog ciklusa programskog proizvoda jeproizvesti softver s najvišom kvalitetom i najnižim troškovima u najkraćem mogućem roku.

Životni ciklus programskog proizvoda najčešće obuhvaća sljedeće faze: analiza i definiranje zahtjeva, oblikovanje (dizajn), implementacija programa, verifikacija i validacija i održavanje. Svako od ovih stanja može se promatrati kao nezavisan proces.

Definirani su i dizajnirani različiti modeli životnog ciklusa razvoja softvera koji se slijede tijekom procesa razvoja softvera. Ti se modeli nazivaju i modelima procesa razvoja softvera. Svaki model procesa slijedi niz koraka jedinstvenih za svoju vrstu kako bi se osigurao uspjeh u procesu razvoja softvera. Neki od najpopularnijih modela životnog ciklusa razvoja softvera su: model vodopada, iteracijski model, spiralni model, V-model, agilne metode poput ekstremnog programiranja (XP), kristalnih metoda, Scrum-a, dinamična metoda razvoja sustava, razvoj baziran na karakteristikama itd.

Cilj ovog rada je utvrditi u čemu se ogleda značaj životnog ciklusa za jedan softverski proizvod. Žele se proučiti i prikazati glavna obilježja životnog ciklusa. Cilj rada je istražiti koliko je životni ciklus i njegove faze bitne da bi jedan softverski proizvod ostvario uspjeh. Također se želi na jednom mjestu objediniti sva saznanja iz trenutno dostupne literature iz područja modeliranja razvojnog procesa softvera, te pružiti podlogu i poticaj za daljnje proučavanje i unapređivanje ovog područja.

Rad je strukturiran na sljedeći način: nakon uvodnog razmatranja, upoznajemo se s značenjem životnog ciklusa softverskog proizvoda i njegovim povijesnim razvojem. Sljedeći dio analizira faze životnog ciklusa softvera. Nadalje, rad se bavi modelima životnog ciklusa, a u okviru njega bit će analizirane tradicionalne (klasične) i agilne metode razvoja softvera. Posljednji dio rada odnosi se na upravljanje životnog ciklusa koje obuhvaća tri važna aspekta: upravljanje životnim ciklusom, razvoj softvera i održavanje softvera. Slijedi zaključak, gdje su iznijete završne riječi o odabranoj temi.

2. PROCES RAZVOJA SOFTVERA

2.1. Pojmovno određenje životnog ciklusa softvera

U općem slučaju, pod procesom Pfleeger i Atlee (2006, str. 45) podrazumijevaju uređeni skup zadataka koji se obavljaju da bi se pružila određena usluga ili izradio proizvod. Zadaci se obično izvršavaju prema istom redoslijedu (primjerice, zidovi se ne žbukaju dok se ne postave sve električne instalacije). Postojanje procesa je neophodno jer on osigurava konkretno i strukturirano realiziranje aktivnosti.Za svaki proces je, dakle, jako bitno da se definiraju aktivnosti koje treba izvršiti, ograničenja i resursi da bi se na kraju ostvario željeni rezultat, zaključuju Pfleeger i Atlee (2006, str. 45).

S obzirom na to da softver predstavlja jednu vrstu proizvoda, navedene osobine važe i za proces razvoja softvera. Softverski razvojni proces se naziva i životni ciklus proizvoda, jer „opisuje "život" softverskog proizvoda od početka njegove izrade do njegovog operativnog korištenja i održavanja" (Tomašević, 2017, str. 15). Sličnu definiciju životnog ciklusa softvera iznose Čubranić, Kaluža i Novak (2013). Ovi autori softverski razvojni proces vide kao„ciklus aktivnosti u razvoju, korištenju i održavanju softvera" (Čubranić i sur., 2013, str. 241). Somerville (2011) daje nešto jednostavniju definiciju koja softverski razvojni proces odnosno životni ciklus programskog proizvoda određuje kao skup povezanih aktivnosti koji dovodi do proizvodnje softverskog proizvoda.

Čubranić i sur. (2013) navode sljedeće aktivnosti koje su od značaja za životni ciklus razvoja softvera:

- Inicijalizacija sustava – aktivnost u kojoj se navodi podrijetlo softvera;
- Analiza i specificiranje zahtjeva – aktivnost u kojoj se identificiraju problemi koje je potrebno riješiti novim softverom;
- Specifikacija funkcija – aktivnost u kojoj se identificiraju i formaliziraju podaktivnosti definiranja predmeta obrade, identificiranje atributa i veza objekata i operacija;
- Strukturiranje i izbor dijelova – aktivnosti kojima se na osnovi identificiranih zahtjeva i specifikacije funkcija strukturira softver na takve dijelove kojima se može upravljati, a koji predstavljaju logičke cjeline;
- Specifikacija strukture – aktivnost u kojoj se definiraju međusobne veze između dijelova strukture i sučelje između modula sustava;

- Specifikacija detaljnih komponenti dizajna – aktivnost u kojoj se definiraju procedure putem kojih se izvori podataka svakog pojedinog modula transformiraju iz potrebnih ulaza u zahtijevane izlaze;
- Implementacija komponenti i otklanjanje nedostataka – aktivnost u kojoj se kodiraju dizajnirane procedure i procesi i pretvaraju u izvorni kod;
- Integracija i testiranje softvera – aktivnost koja potvrđuje i održava cjelokupnu integralnost komponenti softvera putem verifikacije konzistentnosti i kompletnosti uvedenih modula;
- Provjera dokumentacije i uvođenje softvera – aktivnost koja obuhvaća izradu sistemske dokumentacije i uputa za korisnika;
- Obuka i upotreba – aktivnost koja osigurava korisnicima softvera instrukcije i upute za razumijevanje mogućnosti i ograničenja u cilju uspješne upotrebe sustava;
- Održavanje softvera – aktivnost koja podržava operacije sustava u ciljnom okruženju kako bi se osigurala potrebna unapređenja, proširenja, popravke, zamjene i dr.
- Gašenje softvera (povlačenje iz primjene) – posljednja aktivnost u životnom ciklusu.

„Upotreba koncepta životnog ciklusa omogućava podjelusloženih zadataka na manje i jednostavnije, lakšu identifikacijuzadataka, osiguravanje zajedničke terminologije, lakšukontrolu financijskih resursa projekta, sistematičnu evaluacijuopcija i alternativa, integraciju aktivnosti, upravljanjeneizvjesnošću itd. Životni ciklus, u najvećoj mjeri,determinira i metodu i strategiju razvoja softverskog sistema“ (Stanišević i sur., 2013, str. 635).

2.2. Povijesni razvoj životnog ciklusa proizvoda

Proces razvoja softvera je vremenom evoluirao. Prašo, Junuz i Hamulić (2016, str. 18) napominju da je u početku, s pojavom informacijskih tehnologija, razvoj softvera bio fokusiran na tehnologiju, programiranje i tehničke vještine. Dakle, takav softver bio je jednostavan i programeri su ga mogli generirati direktnim pisanjem koda. Nije bilo nikakvog planiranja, već se na osnovu nekoliko odluka formirao sustav.

Međutim, kasnije, kada su zahtjevi postali brojniji, softver je mogao razvijati samo veći broj programera koji su sačinjavali razvojni tim. Članovi razvojnog tima morali su

međusobno surađivati, na isti način shvaćati problem i razmatrati njegovo rješenje, da bi proizveli željeni proizvod. Zbog povećanja složenosti softvera, dodavanje novih funkcionalnosti je postajalo sve teže, kao i pronalaženje i otklanjanje grešaka. Vremenom je došlo do ideje da bi se ovaj problem najlakše mogao riješiti uvođenjem metodologije razvoja kako bi se postigla bolja predvidljivost i veća učinkovitost razvoja softvera (Tomašević, 2017, str. 15).

3. FAZE ŽIVOTNOG CIKLUSA PROGRAMSKOG PROIZVODA

Svaka faza životnog ciklusa softvera različito se promatra i svaka od njih predstavlja proces (ili skup procesa) koji se sastoji od aktivnosti, a svaka aktivnost podrazumijeva određena ograničenja, izlaze i resurse (Pfleeger i Atlee, 2006, str. 47).

Manger (2016, str. 5) navodi sljedećih pet faza životnog ciklusa softvera:

- Utvrđivanje zahtjeva (specifikacija);
- Oblikovanje (design);
- Implementacija (programiranje);
- Verifikacija i validacija;
- Održavanje odnosno evolucija.

3.1. Utvrđivanje zahtjeva

Utvrđivanje zahtjeva je prva i jedna od najvažnijih faza životnog ciklusa softvera. Prema Osivčić (2019, str. 6), u ovoj fazi se utvrđuju zahtjevi koje sistem treba zadovoljiti, što se postiže intenzivnom suradnjom razvojnog tima i kupaca odnosno budućih korisnika sistema. Od uspješnosti ove faze ponajvišeovisi ishod cijelog projekta razvoja softvera.

Zahtjev opisuje što sustav treba raditi, odnosno predstavlja izraz željenog ponašanja softvera. Zahtjevi se mogu podijeliti na dvije vrste: funkcionalnei nefunkcionalne zahtjeve. „Funckionalni zahtjevi opisuju funkcije sustava, dakle usluge koje bi on trebao obavljati, izlazekoje on daje za zadane ulaze, te njegovo ponašanje u pojedinim situacijama“ (Manger, 2005, str. 9). Primjerice, u slučaju sustava za obračun plaća, funkcionalni zahtjevi definiraju: koliko često se izdaju platne liste, koji ulaz je neophodan da bi se isprintala platna lista, pod kojim uvjetima se može promijeniti iznos za isplatu, kao i koji su razlozi da se radnik ukloni s platnog spiska. Nefunkcionalni zahtjevi izražavaju ograničenja na funkcije sustava, na primjer traženo vrijeme odziva, traženu razinu pouzdanosti, dozvoljeno zauzimanje memorije, poštivanje odredenih standarda itd.

Dora i Dubey (2013, str. 22) objašnjavaju da je glavni cilj ove faze znati stvarne potrebe klijenta i pravilno ih dokumentirati. Popović (2019, str. 5) navodi da se kao rezultat ove faze dobiva potpun i sveobuhvatan skup dokumentiranih zahtjeva u obliku lista zahtjeva, korisničkih priča ili slučajeva korištenja koji softverski tim koristi kako bi počeo razvijati softver.

Tijekom ove faze trebaju se obaviti sljedeće aktivnosti:

- Izrada studije izvodljivosti (mogu li uočene potrebe korisnika biti zadovoljene pomoću dostupnih hardverskih i softverskih tehnologija, bi li predloženi sustav bio poslovno isplativ te može razviti prema raspoloživom budžetu);
- Otkrivanje i analiza zahtjeva (otkrivaju se zahtjevi, tako da se promatraju postojeći sustavi, analiziraju radni procesi, intervjuiraju budući korisnici i njihovi manageri čime se stvaraju modeli sustava, a ponekad i prototipovi ponašanja sustava);
- Utvrđivanje zahtjeva (nakon otkrivanja zahtjeva, prelazi se na izradu specifikacije u okviru koje se informacije skupljene analizom pretvaraju u tekstove koji definiraju zahtjeve);
- Validacija zahtjeva (provjerava se jesu li zahtjevi realistični, konzistentni i potpuni; drukčije rečeno, odgovaraju li onome što korisnik očekuje od programskog proizvoda) (Manger, 2005, str. 9).

3.2. Oblikovanje (dizajn)

Oblikovanje (dizajn) je najkreativnija faza u životnom ciklusu razvoja softvera.Sommerville(2011, str. 177) definira dizajn softvera kao kreativnu aktivnost u kojoj se na osnovu zahtjeva korisnika identifikuju softverske komponente i njihovi odnosi. Riječ je, dakle, o kreativnom procesu pretvaranja problema u rješenje.

Cilj ove faze je pretvaranje specifikacije zahtjeva u strukturu ili plan. Manger (2016, str. 5) naglašava da se u ovoj fazi oblikuje građa sustava, način rada njegovih dijelova te sučelje između dijelova; drugim riječima, projektira se rješenje koje određuje kako će softver raditi. Popović (2019) oblikovanje (dizajn) dijeli na dvije faze: inicijalni i detaljni dizajn. Tijekom inicijalnog dizajna se modelira sustav tako što se na osnovu zahtjeva pravi skup skica, prikaza korisničkog interfejsa i drugih dokumenata koji pomažu softverskom timu da razumije što i kako treba napraviti, ali i korisnicima da potvrde da softverski tim ispravno razumije problem koji treba riješiti, dok detaljni dizajn predstavlja razvoj tehničke dokumentacije s detaljima kao što su klasni dijagrami, dizajn šablone, opisi protokola komunikacije i sl. (Popović, 2019, str. 6).

Dizajn se može realizirati na jedan od sljedećih nekoliko načina:

- Modularno raščlanjivanje (ovaj dizajn zasniva se na tome da se komponentama dodijele funkcije);
- Raščlanjivanje na osnovu sustavnih podataka (ovaj dizajn zasniva se na vanjskim strukturama podataka);
- Raščlanjivanje na osnovu događaja (dizajn se zasniva na događajima koje sustav treba obraditi i koristi informacije o tome kako događaji mijenjaju stanje sustava);
- Dizajn iz vana ka unutra (zasniva se na onome što korisnik unosi u sustav);
- Objektno orijentirani dizajn (definira klase objekata i njihove međusobne veze) (Wasserman, 1995; prema Pfleeger i Atlee, 2006, str. 227).

Kako navodi Tomašević (2017, str. 71), razlikuju se tri nivoa dizajna:

- Arhitektura sustava (povezuje mogućnosti sustava navedenih u specifikaciji zahtjeva s komponentama sustava koje će biti implementirane. Komponente su najčešće moduli, a arhitektura također opisuje njihove međusobne veze, kao i načine generiranja složenih sustava od manjih podsistema);
- Dizajn koda (obuhvaća izbor programskog jezika, izbor struktura podataka u kojima će se čuvati relevantni podaci, izbor algoritama za obradu i manipulaciju podacima, a također se utvrđuje skup programskih modula, potrebne procedure i funkcije sustava) i
- Završni dizajn (još detaljnije opisuje implementaciju navođenjem informacija o formatima podataka, šablonama bitova, primjenjenim protokolima i sl.).

Premda je korisno raditi od gore nadolje, iskustva pokazuju da u praksi projektanti naizmjenično prelaze s jedne razine na drugu kako više upoznaju rješenja i njegove posljedice. Tako na primjer, dok projektanti istražuju neke aspekte sustava, mogu u komunikaciji s programerima ili onima koji testiraju sustav doći do zaključaka koji ukazuju na to da bi trebalo promijeniti dizajn u svrhu unaprjeđenja implementacije, mogućnosti testriranja ili održavanja. Stoga, projektanti moraju postupno raditi na arhitekturi, programima i završnom dizajnu u skladu s osobnim poznavanjem rješenja i svojom osobnom kreativnošću.

3.3. Implementacija

Nakon što se napravi dizajn, može se početi s programiranjem (implementacijom). Implementacija koda nije izolirana aktivnost i često je tijesno povezana s aktivnostima dizajna i testiranja tako da se ove tri aktivnosti često vrše usporedno. Dora i Dubey (2013, str. 23) napominju da se u ovoj fazi oblikovano softversko rješenje pretvara u kod uz pomoć raspoloživih programskih jezika. Šimunović (2017, str. 11) dodaje da se izvorni kod obično piše u programskim jezicima visoke razine kao što su PHP, Java, Delphi, C/C++ itd. Implementacija, pored pisanja programskog kodaprema napravljenom projektu, uključuje i aktivnosti pregleda programskog koda, spajanja verzija koda koje napišu različiti članovi tima, kao i isporuku koda korisniku (Popović, 2019, str. 6).

Implementacija, iako djeluje jednostavna jer je sve ranije definirano modelima, obično to nije. Tomašević (2017, str. 111) navodi da su razlozi za to četvorostuki: najprije, postoji vjerovatnost da projektanti nisu uzeli u obzir sve značajke platforme ili razvojnog okruženja koji će biti korišteni, drugo, strukture podataka je jednostavnije predstaviti tablicama ili grafičkim prikazom nego napisati ekvivalentan programski kod, treće, programi trebaju biti razumljivi svima, kako onima koji su ih pisali, tako i onima koji sudjeluju u njegovom razvoju i četvrto, treba težiti da se napisani kod može jednostavno ponovo iskoristiti ako se za to ukaže prilika.

Tijekom razvoja softverskog proizvoda moraju se poštovati standardi koji vrijede u konkretnom okruženju. Mnoge kompanije zahtijevaju da programski kod bude usklađen po pitanju stila, formata, sadržaja i drugih parametara, kako bi kod i sva prateća dokumentacija bili jasni onome ko ih čita. Brojne su prednosti ovakvog pristupa: postizanje sistematičnosti u radu, lakše i brže tumačenje koda, olakšano pronalaženje grešaka u radu, jednostavnije unošenje u sustav svih novonastalih izmjena, održava se usklađenost elemenata iz dizajna sa elementima iz implementacije, olakšanje nastavka rada na projektu ako bi iz nekog razloga projekat morao biti prekinut (Tomašević, 2017, str. 111).

3.4. Verifikacija i validacija

Verifikacija i validacija predstavlja fazu u kojoj se provjerava radi li softver prema specifikaciji odnosno prema postavljenim korisnikovim zahtjevima (Manger, 2016, str. 5). Obično se svodi na testiranje u kojem se otkrivaju i ispravljaju greške u sistemu. Imajući u vidu da se nakon faze testiranja softver isporučuje krajnjim korisnicima, značaj ove faze je

jako velik. Dora i Dubey (2013, str. 23) ističu da seprednosti učinkovitog testiranja ogledaju u pružanju visokokvalitetnih softverskih proizvoda, nižim troškovima održavanja te preciznijim i pouzdanijim rezultatima.

Proces testiranja ima više faza. Prva faza, jedinično testiranje, provjerava funkcioniraju li pojedinačne komponente ispravno sa svim očekivanim tipovima ulaza, u skladu s dizajnom komponente. Nakon završetka ove faze, pristupa se integracijskom testiranju, koje provjerava surađuju li sustavne komponente kao što je naznačeno u specifikacijama dizajna sustava i programa. Slijedi funkcionalno testiranje kojim se provjerava funkcionalnost sustava odnosno provjerava seizvršava li integrirani sustav zaista izvršava funkcije koje su navedene u specifikaciji zahtjeva. Nakon ove faze, u sklopu testiranja performansi, vrši se poređenje sustava s ostatkom hardverskih i softverskih zahtjeva. Naredna faza je završni test prihvaćanja u kojoj se provjerava usklađenost sustava s opisom zahtjeva kupca. Nakon obavljanja završnog testa, prihvaćeni sustav se instalira u okruženje u kojem se namjerava koristi, pa se tako fazom konačnoginstalacionog testa provjerava da li sustav i dalje ispravno funkcionira (Pfleeger i Atlee, 2006, str. 372).

3.5. Održavanje

Tomašević (2017, str. 16) vidi održavanje kao fazu u kojoj se popravljaju greške u sustavu koje se javljaju nakon isporuke korisnicima. Također se radi i na daljnjem unaprjeđenju dijelova sustava prema promjenljivim korisnikovim potrebama i zahtjevima ili promjenama koje se dešavaju u okruženju.

Održavanje sustava zahtjeva intenzivnu komunikaciju tima za održavanje s korisnicima, kupcima, operaterima i sl. U aktivnosti koje spadaju u djelokrug tima za održavanje ubrajaju se:

- Detaljno upoznavanje sa sustavom kako bi se shvatio način njegovog rada;
- Davanje informacija o radu sustava;
- Upoznavanje s dokumentacijom i njeno ažuriranje;
- Pronalaženje uzroka grešaka i njihovo otklanjanje;
- Prepoznavanje potencijalnih problema u sustavu i njihovo rješavanje;
- Izvođenje potrebnih izmjena u sustavu (u dizajnu, kodu, testovima);
- Oslobađanje sustava od komponenti koji se više ne koriste.

S obzirom na sadržaj i karkater promjena, Manger (2005, str. 54) razlikuje sljedeće vrste održavanja:

- Korekcijsko održavanje (bavi se otklanjanjem uočenih grešaka, bilo da se radi o greškama u kodiranju, greškama u oblikovanju ili greškama u specifikaciji;
- Adaptacijsko održavanje (primjenjuje se kada je potrebno provesti prilagodbu na novu platformu, na primjer drugi hardver,drugi operacijski sustav, ili drugu biblioteku potprograma) i
- Perfekcijsko održavanje (implementiraju se novi funkcionalni ili nefunkcionalni zahtjevi koji odražavaju povećane potrebe korisnika ili promjene u poslovnom okruženju).

4. MODELI ŽIVOTNOG CIKLUSA SOFTVERA

Modeliranje procesa razvoja softvera osigurava potpunu kontrolu i koordinaciju svih aktivnosti koje treba provesti da bi se proizveo željeni softver i ispunili ciljevi projekta. Pfleeger i Atlee (2006, str. 48) navode sljedeće razloge za modeliranje:

- Kada projektni tim opiše proces projektiranja, taj opis postaje zajedničko shvaćanje svih aktivnosti, resursa i ograničenja uključenih u razvoj softvera;
- Modelovanje pomaže u pronalaženju nedosljednosti, suvišnih i izostavljenih elemenata, čime proces postaje efikasniji i usmjereniji na gradnju finalnog proizvoda;
- Model odražava ciljeve razvojakao što je izrada softvera visokog nivoa kvaliteta, otkrivanje grešaka u ranim fazama razvoja itd. Nakon izrade modela, projektni tim ocjenjuje predviđene aktivnosti saspekta njihove usklađenosti s postavljenim zahtjevima;
- Modeli se kroje prema posebnoj situaciji u kojoj će se primjenjivati, ali se mogu koristiti i u drugim situacijama uz određena prilagođavanja.

Danas postoji veći broj modela životnog ciklusa razvojasoftvera. U zavisnosti od njihove fleksibilnosti, mogu se podijeliti na tradicionalno planski vođene modele i novije agilne razvojne modele. U nastavku rada će se pobliže objasniti osnovni modeli koji slijede propise i tradicionalne, ali i agilne metodologije.

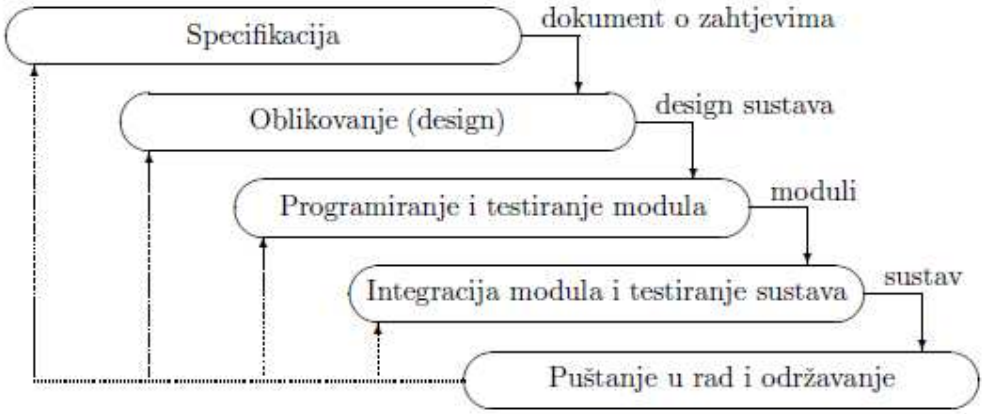
4.1. Tradicionalni modeli za razvoj softvera

Javanmard i Alian (2015, str. 1386) objašnjavaju da se tradicionalne metodologije temelje na planu u kojem rad započinje s pronalaženjem i dokumentiranjem cjelovitog skupa zahtjeva, nakon čega slijedi izgradnja rješenja, testiranje i implementacija. Zbog velikog spektra posla koje ove aktivnosti zahtijevaju, ova je metodologija postala poznata kao *heavyweight* metodologija. Iako postoji mnogo tradicionalnih metoda, u nastavku rada bit će objašnjeni vodopadni model, V-model, model prototipiranja, model inkrementalnog razvoja, evolucijski i spiralni model.

4.1.1. Vodopadni model

Model vodopada (engl. *waterfall model*) nastao je u ranim 70-im godinama 20. stoljeća što ga čini vjerojatno najstarijim objavljenim modelom razvoja softvera. Softverski

proces je građen kao niz vremenski odvojenih aktivnosti koje se odvijaju kao faze jedna iza druge (Manger, 2016, str. 6) (sl. 1).



Slika 1. Softverski proces prema modelu vodopada (Manger, 2016)

Dora i Dubey (2013, str. 24) ističu da u modelu vodopada, svaka faza mora biti dovršena u cijelosti prije nego što započne sljedeća faza. Na kraju svake faze odvija se pregled kako bi se utvrdilo je li projekt na dobrom putu i hoće li se projekt nastaviti ili odbaciti.Model vodopada služi kao osnova mnogim drugim modelima životnog ciklusa, zaključuju Dora i Dubey (2013, str. 24).

Model vodopada se koristio za propisivanje aktivnosti u procesu razvoja softvera u različitim kontekstima. Primjerice, ovaj model, opisan u sklopu standarda Defence standard 2167-A, je dugi niz godina bio osnova ugovora za razvoj i isporuku softvera u Ministarstvu obrane SAD. Uz svaku aktivnost u sklopu procesa definirane važne točke i međuproizvodi da bi rukovodioci projekta mogli na osnovu modela mjeriti završni stupanj projekta. Primjerice, testiranje dijelova i integriranosti u modelu vodopada završava se završnom točkom "napisani, testirani i integrirani moduli", dok je međuproizvod primjerak testiranog koda. Nakon toga, kod se predaje inžinjerima, koji obavljaju testiranje sustava (Pfleeger i Atlee, 2006, str. 49).

Primjena modela vodopada ima brojne prednosti. Prašo, Junuz i Hamulić (2016, str. 19) kao ključne prednosti ističu to što su softverski zahtjevi identificirani mnogo ranije nego što započne njegov razvoj, kao i to što su promjene u zahtjevima sve manje kako projekt implementacije softvera sve više napreduje. Manger (2016, str. 6) smatra da ovaj model

omogućava detaljno planiranje cijelog softverskog procesa i podjelu poslova na veliki broj suradnika te utvrđivanje s lakoćom stanja u kojem se proces trenutno nalazi. Veseli (2007, str. 9)dodaje da je prednost ovog modela u njegovom jednostavnom i discipliniranom pristupu, jer napreduje linearno kroz diskretne, lako razumljive ilako objašnjive faze.

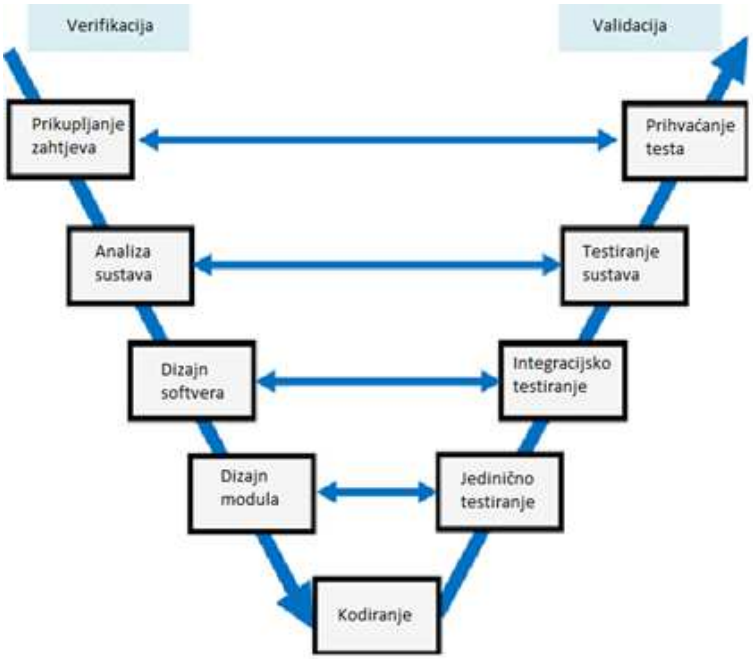
S druge strane, prema Mangeru (2016, str. 6-7), nedostaci vodopadnog modela su:

- Navedene faze softverskog procesa u modelu vodopada u praksi je teško razdvojiti, zbog čega dolazi do naknadnog otkrivanja pogrešaka i nepoželjnog vraćanja u prethodne faze;
- Zbog tendencije „zamrzavanja“ pojedine faze, događa se se da se sustav nastavlja razvijati u nezadovoljavajućem obliku i
- Proces je spor zbog čega se može dogoditi da u trenutku puštanja u rad sustav bude neažuriran i zastario.

Zbog svih ovih nedostataka, vodopadni model je vremenom doživio mnoga unapređenja. Na osnovu njega, predloženo je nekoliko novih modela procesa razvoja softvera u kojima su otklonjeni neki od nedostataka.

4.1.2. V model

Sljedeći model životnog ciklusa jest V-model, koji predstavlja modificiranu i unaprijeđenu verziju modela vodopada. Ovaj model nastao je 1992. godine u Njemačkoj za potrebe Ministarstva obrane Njemačke. Naziv modela potiče od činjenice da se proces razvoja softvera predstavlja u vidu dijagrama u obliku slova V (slika 2). Također, slovo V u nazivu modela označava važnost verifikacije i validacije u procesu. Kao što je prikazano na ovoj slici, kodiranje predstavlja ishodište V modela, pri čemu su analiza i dizajn na lijevoj, a testiranje i održavanje na desnoj strani. Sve faze kod V-modela su sekvencijalne, što znači da sljedeća faza može početi tek kad se prethodna završi (Tomašević, 2017, str. 20).



Slika 2. V model životnog ciklusa softvera (Tierno, Santos i Arruda, 2016)

Podreka (2018, str. 25) objašnjava osnovnu razliku u odnosu na model vodopada, navodeći da se, za razliku od vodopadnog modela gdje se tijek aktivnosti konstantno provodi prema dolje, kod V modela nakon implementacije slijedi povratak prema gore, što omogućava vraćanje iz procesa koji dolaze kasnije u procese koji su već izvršeni ranije. Kod vodopadnog modela, podsjećaju Pfleeger i Atlee (2006, str. 52), pozornost se posvećuje dokumentima i artefaktima, dok se V model usredotočuje na aktivnosti i ispravan rad sustava.

Da bi sustav ispravno radio, on mora proći kroz tri faze: testiranje pojedinačnih modula s integracijom, testiranje integriranog sustava i završno testiranje.Cilj testiranja pojedinačnih modula uz postepenu integraciju da se razvojni tim uvjeri da je svaki aspekt sustava ispravno implementiran. S druge strane, cilj testiranja integriranog sustava je taj da dokaže da sustav kao cjelina radi ono što se od njega očekuje. Dakle, može se zaključiti da ove dvije faze služe za provjeru je li sustav ispravno i u potpunosti implementiran prema napravljenom projektu. Završnim testiranjem, kojeg češće provode naručioci softvera nego projektni tim, provjeravajesu li svi zahtjevi korisnika u potpunosti implementirani (Pfleeger i Atlee, 2006, str. 52).

Tomašević (2017) objašnjava postupak razvoja softvera prema V modelu. Najprije se provodi analiza zahtjeva, zatim se projektira sustav i programi, a onda se prelazi na kodiranje. Nakon kodiranja slijede faze testiranja. Ukoliko bi se desilo da se u nekoj od faza testiranja

pojavi problem, prema zadanim povratnim uvjetima, sve aktivnosti iz faza u lijevom dijelu dijagrama se ponavljaju. Nakon unošenja svih potrebnih prepravki i dopuna, intervenira se u programskom kodu, a zatim se ponovo provodi testiranje. Sustav se može razvijati u više iteracija, sve dok se ne dođe do konačne verzije sustava koja će biti isporučena (Tomašević, 2017, str. 21).

V model je jedan od najčešće korištenih modela za razvoj softvera. Njegove prednosti, prema Tomašević (2017, str. 21), ogledaju se u sljedećem:

- Podržava povratne uvijete (dopušta povratak na prethodne faze razvoja što ga čini primjenjivim i za razvoj složenih softverskih sustava);
- Omogućava verifikaciju i validaciju sustava (može se provjeriti je li projekt dobro napravljen i implementiran te da li sustav ispunjava zahtjeve korisnika);
- Generira kvalitetan proizvod (zbog svog strogo kontroliranog procesa razvoja, može se garantirati dobra kvaliteta softverskog proizvoda);
- Vodi računa o testiranju u ranim fazama projekta (što kasnije dovodi do značajne uštede u vremenu potrebnom za testiranje).

S druge strane, nedostaci ovog modela su:

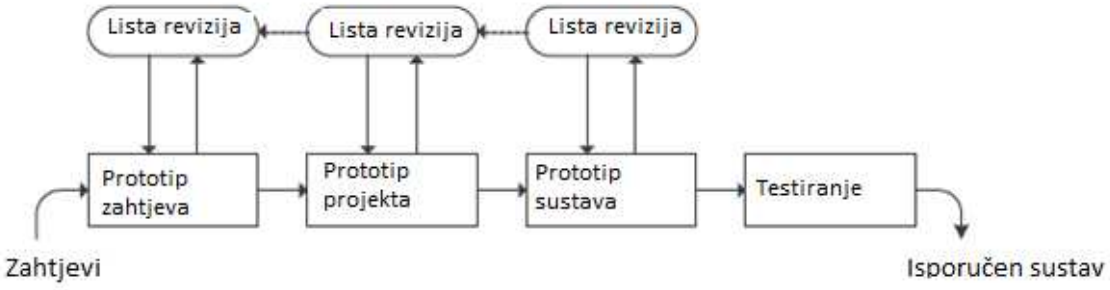
- Nedovoljna fleksibilnost (ako se jave neki problemi i dođe do povratka na ranije faze, nije dovoljno samo unijeti izmjene, već se moraju ažurirati i sve predhodne faze, uključujući i prateću dokumentaciju);
- Zahtjeva mnogo resursa (zbog zahtjeva za brojnim resursima i većim novčanim sredstvima mogu ga primjenjivati samo veće kompanije).

4.1.3. Prototipiranje

Prototipski model razvoja softvera se zasniva na izradi prototipova softverske aplikacije. Podreka (2018, str. 26) definira prototip kao „brzo i jednostavno razvijeni program, koji obavlja budući sustav“.Tomašević (2017, str. 25) pod prototipom podrazumijeva nekompletnu verziju programa koji se razvija. Pfleeger i Atlee (2006, str. 53) dodaju da se zahvaljujući prototipovima brzo konstruira kompletan sustav ili njegovi dijelovi u cilju razjašnjenja ili boljeg razumijevanja otvorenih pitanja.

Postoje prototipovi različitih vrsta, što ovisi od cilja koji se njima želi postići. Bez obzira na vrstu, svrha svih prototipova je smanjenje rizika i neodređenosti prilikom razvoja sustava.

Naredna slika prikazuje opći oblik prototipskog modela. Razvoj po protopskom modelu započinje prikupljanjem zahtjeva koji odgovaraju potrebama korisnika ili naručioca. Zatim se analiziraju prijedlozi različitih varijanti izgleda ekrana korisničkog interfejsa, načina unosa podataka, tablica s podacima, izlaznih izvještaja itd. Korisnici izlažu svoje želje i preciziraju zahtjeve za softver koji će se razviti. Nakon postizanja dogovora o zahtjevima, projektni tim prelazi na dizajn. Inicijalni dizajn se revidira sve dok projektni tim, naručilac i korisnici ne budu zadovoljni rezultatom. Razmatranje varijanti dizajna ponekad otkriva problem koji potječe od zahtjeva. U tom slučaju projektni tim se vraća na izrađene zahtjeve, ponovo ih analizira i mijenja specifikaciju zahtjeva. Sustav se zatim kodira uz razmatranje različitih varijanti, s mogućim ponovnim iteracijama kroz zahtjeve i dizajn (Pfleeger i Atlee, 2006, str. 54). Poslije više iteracija, kada se postigne konačni dogovor o zahtjevima, kao rezultat dobiva se prototip zahtjeva.



Slika 3. Model životnog ciklusa softvera baziran na prototipovima (Tomašević, 2017, str. 26.)

Prototipski model razvoja sustava odgovara potrebama za bržim, jeftinijim i jednostavnijim pristupima razvoja sustava (Podreka, 2018, str. 26). Općenito govoreći, njegove prednosti su:

- Reduciranje vremena i troškova (ranim utvrđivanjem što korisnik točno želi ubrzava se razvoj i smanjuju troškovi na projekti koji vremenom eksponencijalno rastu) i
- Intenzivna interakcija s korisnicima (uključivanjem korisnika u izradu prototipova značajno se smanjuje mogućnost pogreške čime se poboljšava kvaliteta finalnog proizvoda).

Nedostaci prototipskog modela razvoja softvera su:

- Nedovoljna analiza (ako se prototipovi ne analiziraju detaljno, može doći do previđanja boljih rješenja, izrade nekompletne specifikacije ili konverzije prototipova u konačna rješenja koja su teška za održavanje) i
- Konfuzija između prototipova i finalnih sustava (česte su pojave gdje korisnici očekuju od prototipova da precizno modeliraju performanse finalnog sustava, što nije namjera razvojnog tima. Također, korisnici ponekad prihvaćaju svojstva uključena u prototip kao finalna, iako će ona kasnije biti isključena iz sustava, pa može doći do konflikta s razvojnim timom).

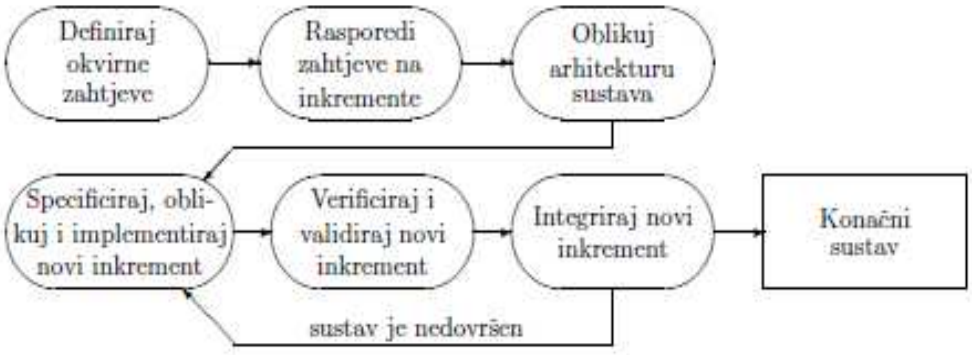
4.1.4. Model inkrementalnog razvoja

Inkrementalni model dijeli proizvod na verzije, gdje se dijelovi projekta kreiraju i zasebno testiraju (Dora i Dubey, 2013, str. 24). Tomašević (2017, str. 23) verzije definira kao male funkcionalne podsisteme koji sadrže različite skupove funkcija, svaki za sebe. „Svaka isporuka nove verzije znači dalju nadogradnju sustava do njegove pune funkcionalnosti“ (Tomašević, 2017, str. 23).

Kod inkrementalnog modela sustav se razvija u nizu iteracija (Manger, 2016, str. 8). Podreka (2018, str. 27) potvrđuje ovo gledište i dodaje da svaka iteracija procesa isporučuje skup samostalnih cjelina krajnjem korisniku.

Sam razvoj informacijskog sustava provodi se inkrementalno, odnosno po dijelovima. Primjenom svakog od tih dijelova nastoji se zapravo izbjeći konstantna promjena. Arhitektura ili okvir sustava utvrđuju se vrlo rano, u okviru istraživanja i razvoja, a služi kao osnova daljnjeg razvoja ili procesa (Galinac, 2018).

Razvoj jednog inkrementa unutar jedne iteracije odvija se po bilo kojem modelu – primjerice kao vodopad, a ideja je prikazana na sl. br. 4.



Slika 4. Inkrementalni razvoj softvera (Manger, 2016, str. 8.)

Kako Tomašević (2017, str. 23-24) navodi, prednosti koje korisnicima pruža inkrementalni model razvoja su:

- Brza isporuka operativnog skupa funkcija –isporukom prve faze, korisnici imaju na raspolaganju potpuno realizirani podskup funkcija koje će biti sustavni dio konačnog proizvoda;
- Vidljiv napredak na projektu – progres na projektu je uvijek transparentan, odnosno može se lako i jednostavno pratiti. Vidljivost napretka nije uočljiva samo u dokumentima, već i u praktičnom radu;
- Stalan odaziv korisnika – interakcija s korisnikom tijekom cijelog ciklusa razvoja vodi ka stabilnijim međurezultatima i sustavu uopće. Zahvaljujući interakciji korisnik može blagovremeno interveniratiako u prvim operativnim funkcijama s kojima se susreće uoči neke pogreške. Tako će troškovi biti manji, a razvoj učinkovitiji;
- Mali projektni tim – manji broj članova u razvojnom timu smanjuje troškove na projektu.

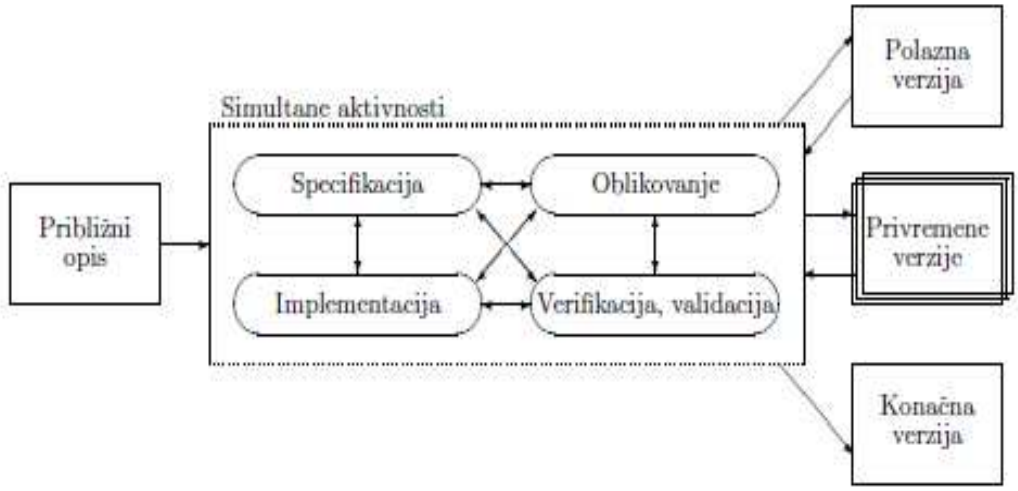
Manger (2016, str. 8) navodi nedostatke inkrementalnog modela:

- Ponekad je teško podijeliti korisničke zahtjeve u smislene inkremente i
- Teško je odrediti zajedničke module koji su potrebni raznim inkrementima i koji bi morali biti implementirani u prvom inkrementu.

Pa ipak, inkrementalni model razvoja softvera se vrlo intenzivno koristi u praksi. Svi generički softverski paketi (primjerice Microsoft Office i sl.) razvijani su u inkrementima, a svaka njihova nova verzija donosila je nove mogućnosti, zaključuje Manger (2016, str. 9).

4.1.5. Evolucijski model

Evolucijski model zasniva se na ideji razvoja ili evolucije polazne verzije sustava, koja se pokazuje korisniku, a on ga ocjenjuje i komentira (Podreka, 2018, str.28). Na osnovu korisnikovog mišljenja i njegovih primjedbi, polazna verzija sustava se usavršava sve dok ne dobije u potpunosti pozitivne ocjene korisnika. Tek tada riječ je o krajnjem proizvodu. Postupak evolucijskog modela prikazan je na sljedećoj slici.



Slika 5. Evolucijski razvoj softvera (Manger, 2016, str. 7.)

Prednosti modela evolucijskog razvoja su:

- Model je u stanju proizvesti brzi odgovor na zahtjeve korisnika;
- Razvoj je moguć i onda kad su zahtjevi ispočetka nejasni;
- Specifikacija se može postupno odvijati u skladu sa sve boljim korisnikovim razumijevanjem problema.

Evolucijski model ima sljedeće nedostatke:

- Proces nije transparentan za menadžere. Dakle, oni ne mogu ocijeniti koliki je dio posla napravljen i kad će sustav biti završen;
- Konačni sustav obično je loše strukturiran zbog stalnih promjena te je neprikladan za održavanje;

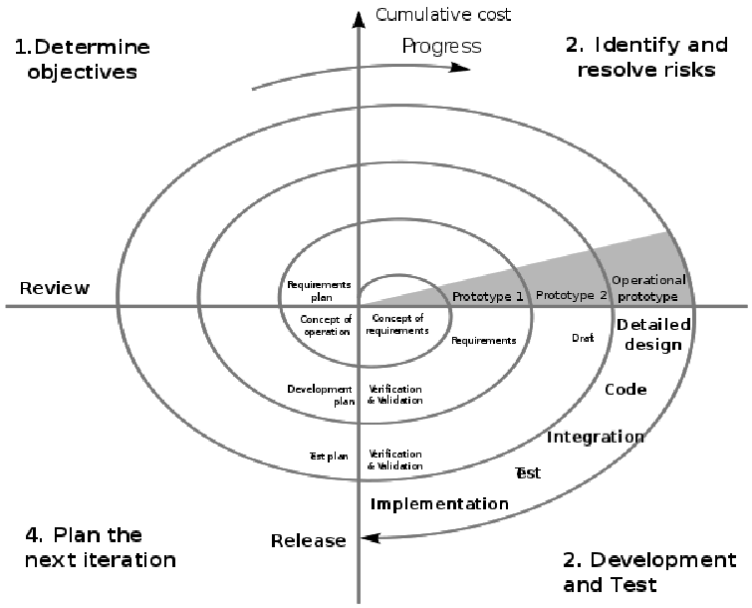
- Zahtijevaju se posebni alati i natprosječni softverski inženjeri.

Evolucijski model uspješno je pronašao svoju primjenu u razvoju web-sjedišta te za male sustave s kratkim životnim vijekom (Manger, 2016, str. 8).

4.1.6. Spiralni model

Dora i Dubey (2013, str. 24) smatraju da je spiralni model sličan inkrementalnom modelu, s tim da je veći naglasak kod ovog modela stavljen na analizu rizika. Spiralni model ima četiri faze: planiranje, analizu rizika, inženjering i evaluaciju.

Softverski projekt više puta prolazi kroz ove faze u iteracijama (u ovom modelu nazvane spirale). Osnovna spirala, počevši od faze planiranja, prikuplja zahtjeve i procjenjuje rizik. Svaka sljedeća spirala gradi se na osnovnoj spirali. Zahtjevi se prikupljaju tijekom faze planiranja. U fazi analize rizika poduzima se proces identifikacije rizika i alternativnih rješenja. Prototip se proizvodi na kraju faze analize rizika. Softver se proizvodi u fazi inženjeringa, zajedno s testiranjem na kraju faze. Faza evaluacije omogućuje korisniku da ocijeni izlaz projekta prema podacima prije nego se projekt nastavi do sljedeće spirale (Dora i Dubey, 2013, str. 24). Postupak spiralnog modela razvoja softvera prikazan je na sljedećoj slici.



Slika 6. Spiralni model (Şaykol, 2012)

Prednosti spiralnog modela su:

- reduciranje rizika (primjenom spiralnog modela, minimiziraju se rizici i greškepri izradi sustava);
- dobra kontrola troškova (procjena troškova se može lako izvršiti, zbog čega krajnji kupac ima dobru kontrolu administriranja novog sustava);
- aktivno sudjelovanje korisnika (interakcija s korisnikom omogućava ravnomjeran razvoj softverskog proizvoda koji zadovoljava sve potrebe korisnika).

Nedostaci mogu biti:

- ograničena primjena (pogodan je za velike i složene projekte);
- neophodno znanje o rizicima (model zahtjeva veliku vještinu u evaluaciji neizvjesnosti i rizika);
- složenost modela (spiralni model ima striktno definiran protokol razvoja, koga je ponekad teško ispoštovati).

4.2. Agilne metode

„Fokusiranost tradicionalnih modela u razvoju softvera na procese tijekom vremena postala je frustrirajuća za neke od sudionika koji su se služili tradicionalnim modelima“ (Šimunović, 2017, str. 16). Imajući u vidu tu činjenicu, grupa projektanata su formulirali vlastite metode i prakse kako bi bolje odgovorili promjenama na tržištu. „Ove metode i prakse bazirane su na iterativnim poboljšanjima, tehnici koja je nastala 1975. godine i koja je kasnije postala poznata kao agilne metode“ (Šimunović, 2017, str. 16).

Naziv „agilno“ nastao je u 2001. godini kada se sedamnaest istaknutih svjetskih softverskih inženjera sastalo kako bi raspravljali o budućim trendovima u razvoju softvera. Zaključili su da njihove metode imaju mnogo zajedničkih karakteristika, zbog čega su odlučili nazvati ove procese agilnima, jednostavnim i dovoljnim. Kao posljedica ovih sastanaka nastao je tzv. „Agilni savez“ (eng. *agile alliance*) čiji je cilj promoviranje agilnog razvoja i manifest za agilni razvoj softvera, koji propisuje vrijednosti i principe koje moraju prihvatiti sve druge metode da bi bile agilne.

Agilne metode su nastale kao protuteža metodama "teške kategorije" (tradicionalne, klasične metode), pa su prepoznatljive kao metode "lake kategorije" (Tadić, 2005, str. 238). Manger (2016, str. 12) objašnjava svojstva koja odlikuju agilne metode:

- Razvoj softvera promatra se kao kontinuirani niz iteracija čiji broj nije moguće predvidjeti. Svaka iteracija usklađuje sustav s trenutnim zahtjevima koji se stalno mijenjaju. Ne postoji cjelovita ili konačna specifikacija.
- Ne postoji upravljanje projektom u pravom smislu riječi. Sve aktivnosti se utvrđuju u dogovoru s korisnicima koji aktivno sudjeluju u procesu. Planiranje je kratkoročno (jedan ili dva tjedna unaprijed).
- Izbjegavaju se svi oblici dokumentacije osim onih koji se mogu automatski generirati. Smatra se da su agilne metode prije orijentirane ka izvornom kodu kao ključnom dijelu dokumentacije.

Pfleeger i Atlee (2006, str. 59) upućuju da je cilj agilnih metoda zadovoljavanje naručioca uz pravovremenu i stalnu isporuku softvera. Šimunović (2017, str. 16-17) dodaje da je cilj agilnih metoda stavljanje većeg naglaska na ljude, interakciju, funkcionalni softver, rad s klijentima i promjene. Stanišević i sur. (2013, str. 635) ističu da je primjena agilnih metoda jako korisna u okruženjima koja su stalno promjenljiva i gdje postoji potreba za parcijalnim rješenjima. Isti autori dodaju da agilne metode„pružaju podršku konkurentnom razvoju i permanentnojintegraciji u okruženje kupca" (Stanišević i sur., 2013, str. 636).„Pogodan teren za primjenu agilnih metoda je nizak stupanj kritičnosti, iskusni programeri, visok stupanj promjene zahtjeva, mali broj developera i kultura koja počiva na kaosu" (Tadić, 2005, str. 238)

Postoji veliki broj agilnih metoda razvoja softvera, ali najpopularnije agilne metode su: ekstremno programiranje (XP), kristalne metode, Scrum, dinamična metoda razvoja sustava, razvoj baziran na karakteristikama, adaptivni razvoj sistema, Lean razvoj softvera itd. U nastavku rada ćemo detaljnije objasniti neke od najčešće korištenih agilnih modela u razvoju softvera.

4.2.1. Ekstremno programiranje

Ekstremno programiranje je najpopularnija agilna metoda (Manger, 2016, str. 12). Nastala je 1999. godine, a formulirali su je Kent Beck,Ward Cunningham i Ron Jeffries

(Tadić, 2005, str. 239). Ova metoda, navodi Tomašević (2017, str. 36) orijentirana je ka malim i srednjim timovima koja imaju 6 do 20 članova.

Tadić (2005, str. 239) definira ekstremno programiranje kao „mehanizam za socijalne promjene, stil razvoja, put ka poboljšanju, produktivnost te disciplina razvoja softvera“. On omogućuje proizvodnju dugotrajnog softvera i sposobnost reagiranja na iznenadne trendove, promjene i zahtjeve. Vrlo je fleksibilan, a u tome se očituje njegova vodeća prednost. Pfleeger i Atlee (2006, str. 60) objašnjavaju da se ovaj model zasniva na vrijednostima poput komunikacije (podrazumijeva razmjenu informacija između naručioca i projektnog tima),jednostavnosti (koja ohrabruje projektni tim da odabere najjednostavniji dizajn ili implementaciju), odvažnosti (odnosi se na posvećenost blagovremenim i čestim isporukama funkcija), povratnih sprega (koje se ugrađuju u različite aktivnosti tijekom procesa razvoja).

Projektni tim ekstremnog programiranja se povezuje prateći 12 osnovnih praksi, koje Šimunović (2017, str. 17-18) sažima na:

- Planiranje – Programer procjenjuje rad potreban za implementaciju naručiteljevih zahtjeva te naručitelj odlučuje o rasponu i vremenu na temelju te procjene.
- Mali/kratki izlasci – Softverske aplikacije se razvijaju u serijama malih i često ažuriranih verzija. Nove verzije izlaze po dnevnoj, tjednoj ili mjesečnoj bazi.
- Metafore – sustav je definiran kroz set metafora između naručitelja i programera koje opisuju kako sustav radi.
- Jednostavan dizajn – naglasak je na dizajniranju najprostijeg mogućeg rješenja. Svaka nepotrebna kompleksnost uklanja se čim se otkrije.
- Refactoring– podrazumijeva restrukturiranje sustava kako bi se uklonila duplikacija, popravila komunikacija i jednostavnost i dodala fleksibilnost.
- Programiranje u parovima– sav produkcijski kod pišu dva programera koji rade na jednom računalu.
- Zajedničko vlasništvo – nitko nije odgovoran za pojedine dijelove koda. Svatko može promijeniti svaki dio koda bilo gdje u sistemu u bilo koje vrijeme.
- Stalna integracija – novi dijelovi koda se integriraju s trenutnim sustavom čim je spreman. Tijekom integracije, sustav se gradi nanovo i svi testovi moraju biti uspješno izvršeni kako bi se promjene u sustavu prihvatile.

- Radni tjedan od 40 sati – nitko ne smije raditi prekovremeno dva tjedna zaredom, a ako se to desi, to upućuje na jasne probleme u projektu i procesu.
- Dostupan klijent – klijent mora biti stalno dostupan razvojnom timu.
- Standardi kodiranja – postoje pravila kodiranja kojih se programeri drže kako bi se unijela konzistentnost i kako bi se povećala komunikacija u razvojnom timu.

Kako navodi Awad (2005, str. 9), životni ciklus ekstremnog programiranja podijeljen je u šest faza: istraživanje, planiranje, iteracije do isporuke, produkcija, održavanje i smrt. U fazi istraživanja kupac ispisuje kartice s pričama koje želi uključiti u svoj program. To dovodi do faze planiranja u kojoj je za svaku priču korisnika postavljen prioritet i razvijen raspored isporuke prvog izdanja programa. U fazi iteracije do isporuke, razvojni tim u prvoj iteraciji kreira sustav s arhitekturom cjelokupnog sustava, a zatim kontinuirano integrira i testira svoj kod. Dodatno testiranje i provjera performansi sustava prije nego što se sustav isporuči kupci vrši se u fazi produkcije. Sve odložene ideje i sugestije pronađene u ovoj fazi dokumentiraju se za kasniju primjenu u ažuriranim izdanjima koja će biti objavljena u fazi održavanja. Konačno, faza smrti je blizu kada kupac nema više kartica s pričama, a sva potrebna dokumentacija sustava je napisana, jer se više ne mijenjaju arhitektura, dizajn ili kod sustava (Awad, 2005, str. 9).

Ekstremno programiranje ima mnogostruke prednosti zbog čega je njegova primjena u razvoju softvera velika. Te prednosti ogledaju se u sljedećem:

- Fokus razvojnog tima je na izradi softvera, a ne na dokumentaciji i sastancima, što doprinosi ugodnijoj radnoj atmosferi, većim mogućnostima za napredovanje i učenje i ograničenom radnom vremenu do osam sati dnevno;
- Kraće vrijeme razvoja softvera što pogoduje kupcima;
- Ekstremno programiranje generira dobar softver po prihvatljivoj cijeni (Tomašević, 2017, str. 40).

Nedostaci mogu biti:

- Teško provođenje metoda;
- Nedovoljan broj programera koji bi prihvatili i izvodili obvezne prakse;
- Klijentima ne mora odgovarati ideja da budu uključeni u projekat, jer imaju drugih obveza;

- Teško je uklopiti različite karaktere članova razvojnog tima u funkcionalnu cjelinu.

4.2.2. Scrum model

Scrum je nastao 1994. godine od strane grupacije Object Tehnology, a kasnije su ga komercijalizirali Schwaber i Bedle. „Ovaj model temelji se na iterativnom razvoju, gdje se svaka 30-dnevna iteracija naziva „sprint“, a služi za implementiranje zaostalih zahtjeva najvišeg prioriteta“ (Pfleeger i Atlee, 2006, str. 59-60). Milovanović (2017, str. 699) ovu metodu razvoja softvera veže za složenije projekte sa većim timovima kako bi se omogućilo da se na pravi način raspodijelile timske uloge i kako bi se posao mogao uspješno delegirati.

Awad (2005, str. 10) navodi i objašnjava ključne prakse Scruma:

- Zaostatak proizvoda (engl. *product backlog*) – ovo je prioritetna lista svih funkcionalnosti i promjena koje sustav tek treba izvršiti po želji više aktera, poput kupaca, odjela marketinga i prodaje i projektnog tima. Vlasnik proizvoda (engl. *product owner*) odgovoran je za održavanje zaostalih proizvoda.
- Sprintovi – traju 30 dana, a riječ je o postupku prilagođavanja promjenjivim varijablama okoliša (zahtjevi, vrijeme, resursi, znanje, tehnologija itd.). Moraju rezultirati inkrementalnim softverom koji se može isporučiti. Radni alati tima su: sastanci za planiranje sprinta, zaostatak sprinta i dnevni Scrum sastanci.
- Sastanak o planiranju sprinta - skupu planiranja sprinta prvo prisustvuju kupci, korisnici, menadžment, vlasnik proizvoda i Scrum tim te se odlučuje o skupu ciljeva i funkcionalnosti. Zatim se Scrum Master (projektni manager) i Scrum tim usredotočuju na to kako se proizvod implementira tijekom sprinta.
- Zaostatak sprinta (engl. sprint backlog) – predstavlja listu funkcionalnosti koje su trenutno dodijeljene određenom sprintu. Kad su sve funkcionalnosti dovršene, isporučuje se nova iteracija sustava.
- Dnevni Scrum (engl. *daily Scrum*) – predstavlja dnevni sastanak od približno 15 minuta, koji je organiziran kako bi se pratio napredak Scrum tima i rješavao sve prepreke s kojima se tim suočava.

Dora i Dubey (2013, str. 27) objašnjavaju da Scrum model postaje sve popularniji u softverskoj zajednici zbog svoje jednostavnosti i dokazane produktivnosti. Milovanović (2017, str. 699) dodaje da je naročito pogodan za razvoj softvera u mobilnim tehnologijama.

Glavni nedostatak ovog modela predstavlja loše skaliranje u odnosu na kompleksnost projekta, pa tako ako projekt postane suviše kompleksan može doći do problema u koordinaciji, podjeli posla po iteracijama, strukturiranju tima itd. (Mlovanović, 2017, str. 699).

4.2.3. Dinamična metoda razvoja sustava

Dinamična metoda razvoja sustava razvijena je u Ujedinjenom Kraljevstvu sredinom devedesetih godina prošlog stoljeća. Awad (2005, str. 13) ističe da je ova metoda značajna po tome što ima veliki dio infrastrukture koja je karakteristična za zrele tradicionalne metode, slijedeći načela agilnih metoda. Poseban fokus postavlja na fiksiranje vremena i resursa.

Podreka (2018, str. 33-34) ističe da ovaj model razvoja softvera čine sljedećih pet faza:

- Studija izvedivosti – u ovoj fazi se odlučuje hoće li se koristiti ova metoda ili ne. Ovo se utvrđuje ispitivanjem tipa projekta, organizacijskih i ljudskih problema.
- Poslovna izvedivost – preporučeni pristup ovoj fazi je organiziranje radionica s ciljem olakšavanja razumijevanja poslovne domene projekta, a rezultati iste je definicija arhitekture sustava i plan nacrta prototipa.
- Iteracija funkcionalnog modela – ova faza uključuje analizu, pisanje koda i razvoj prototipa.
- Iteracija razvoja i dizajna – sustav je većinom izgrađen u ovoj fazi. Korisnici ispituju dizajn i funkcionalne prototipe.
- Implementacija –u ovoj završnoj fazi sustav se predaje krajnjim korisnicima.

Awad (2005) navodi da postoji devet praksi koje definiraju ideologiju i koje su temelj svih aktivnosti u ovom modelu razvoja softvera. Neki od temeljnih principa su: aktivna interakcija među korisnicima, česte isporuke, ovlašteni timovi i testiranje tijekom cijelog ciklusa (2005, str. 14). Naglasak je na visokom stupnju kvalitete i prilagodljivosti prema promjenjivim zahtjevima. Poput drugih agilnih metoda, dinamična metoda razvoja sustava pristupa iteracijama kao kratkim vremenski ograničenim ciklusima između dva i šest tjedana.

4.2.4. Razvoj baziran na karakteristikama

Prema Milovanoviću (2017, str. 699), razvoj baziran na karakteristikama (engl. *Feature-driven development* – FDD) predstavlja iterativni, inkrementalni proces razvoja

softvera koji u jednom pristupu objedinjuje najbolje prakse iz industrije. Milovanović (2017, str. 699) dalje nastavlja i navodi da se ovaj model razvoja softvera sastoji iz pet osnovnih aktivnosti:

- Razvoj modela;
- Kreiranje detaljne specifikacije karakteristika;
- Planiranje na osnovu specifikacije karakteristika;
- Dizajniranje na osnovu specifikacije karakteristika i
- Razvoj softvera na osnovu specifikacije karakteristika.

Dora i Dubey (2013, str. 27) napominju da se prve tri faze provode na početku projekta, a da su posljednje dvije faze iterativni dio procesa koji podržava agilni razvoj s brzim prilagodbama kasnim promjenama zahtjeva i poslovnih potreba.

4.2.5. Adaptivni razvoj sistema

Adaptivni razvoj sistema (engl. *Adaptive system development* – ASD) zasniva se na ideji da projekti uvijek trebaju biti u stanju kontinuiranog prilagođavanja promjenama. Zasniva se na ciklusu od tri serije koje se ponavljaju, a to su:

- Špekuliranje;
- Suradnja i
- Učenje.

Pfleeger i Atlee (2006, str. 60) navode šest osnovnih principa na kojima se zasniva adaptivni razvoj softvera, a to su:

- Misija koja usmjerava, postavlja cilj, ali ne propisuje način dolaska do cilja;
- Svojstva imaju suštinsku vrijednost za naručioca pa se projekt mora organizirati oko izgradnje komponenti koje ih implementiraju;
- Iteracije su jako bitne;
- Sve izmjene su dobrodošle i ne promatraju se kao korekcije pogrešaka nego kao prilagođavanje realnostima u procesu razvoja softvera;
- Vrijeme isporuke softvera je utvrđeno;
- Rizik je nešto što se prihvaća, a svi najteži problemi rješavaju se na početku projekta.

Podreka (2018, str. 36) otkriva da se ova metoda izvodi kroz nekoliko faza: pokretanje projekta, adaptivno i cikličko planiranje, konkurentni razvoj komponenti, pregled kvalitete i završni pregled te isporuka. „Pri tome, prve dvije faze obilježene su istraživanjem i promišljanjem, dok konkurentni razvoj biva zasnovan na suradnji. Posljednje dvije faze zasnivaju se na učenju“ (Podreka, 2018, str. 36).

4.2.6. Lean razvoj softvera

Lean razvoj softvera (engl. *Lean Software Development* – LSD) je metoda kod koje je moguće identificirati sedam principa, a to su: eliminacija nepotrebnog, pojačano učenje, odlučiti što je kasnije moguće, dostaviti što je brže moguće, osnažiti tim, graditi integritet i sagledati cjelinu (Podreka, 2018, str. 36). Osnovni nedostaci ove metode su: kašnjenje u izradi softvera, nejasni zahtjevi, nedovoljno testiranje, preopterećenje birokracijom, spora interna komunikacija. Kako bi se one izbjegle, važno je na vrijeme identificirati predmetne rizike te implementirati alat za njihovo reduciranje, izbjegavanje i otklanjanje. Greške se sprječavaju testiranjem, koje se treba provoditi odmah nakon pisanja koda (Podreka, 2018, str. 37).

5. UPRAVLJANJE ŽIVOTNIM CIKLUSOM SOFTVERA

Upravljanje životnim ciklusom softvera kombinacija je ljudi, alata i procesa koji koordiniraju softverom od konceptualizacije do njegove isporuke. Upravljanje životnim ciklusom softvera obuhvaća tri aspekta:

- Upravljanje životnim ciklusom;
- Razvoj softvera i
- Održavanje softvera (Microsoft Docs, 2020).

Životni ciklus programskog proizvoda započinje s idejom da je potrebno kreirati softver koji je potreban za rješavanje određene grupe problema prema prioritetu. Nakon toga slijedi faza intenzivnog razvoja softvera, a nakon nje slijedi faza isporuke softverskog proizvoda (engl. *deployment*) koji je spreman za uporabu, kao i njegovo instaliranje i podešavanje. Softver je tada u fazi upotrebe gdje se pojavljuju značajne aktivnosti održavanja softvera (engl. *software maintenance*), koji po potrebi uključuju cikluse ponovnog razvoja. Ponovni razvoj se javlja u funkciji održavanja onda kada je neophodno ugraditi neke nove funkcionalnosti u softverski proizvod ili kada je neophodno njegovo prilagođavanje novom okruženju. Onda kada softver više ne donosi korist u poslovanju, prestaje njegovo servisiranje, pa se shodno tome softver povlači iz uporabe, čime se završava njegov životni ciklus.

Faze razvoja i uporabe se odnose samo na pojedine etape u životnom ciklusu, dok se upravljanje životnim ciklusom javlja tijekom cijelog životnog ciklusa, jer je ono neophodno da bi se pratilo stanje softvera u svakom trenutku, njegova uporaba te da bi se donijele odgovarajuće odluke. Upravljanje životnim ciklusom je najčešće tijekom razvoja softvera planirano, a nakon isporuke je bazirano na reaktivnim aktivnostima koje se javljaju kao odgovori na određene događaje (npr. zahtjevi za modifikacijom). Uporaba softvera započinje neposredno prije zvanične isporuke, a odnosi se na fazu testiranja softvera od strane korisnika s ciljem sagledavanja softverovih nedostataka.

5.1. Aspekt upravljanja u životnom ciklusu

Upravljanje životnim ciklusom treba osigurati da softver uvijek zadovoljava poslovne potrebe korisnika. „Upravljanje obuhvaća upravljanje zahtjevima, upravljanje resursima, sigurnost podataka, korisnički pristup, praćenje promjena, pregled, reviziju, kontrolu primjene i vraćanje u prethodno stanje“ (Microsoft Docs, 2020).

Upravljanje počinje identificiranjem ideje za razvoj softvera, a zatim se nastavlja analizom slučaja upotrebe. Ovaj aspekt se javlja u fazi razvoja, ali prije nego što započnu konkretne aktivnosti koje imaju za rezultat elemente novog softverskog proizvoda. Kada počne stvarni razvoj softvera, upravljanje životnim ciklusom obuhvaća aktivnosti upravljanja projektom razvoja. Nakon isporuke softvera, upravljanje ciklusom se svodi na skup aktivnosti i metoda za upravljanje aplikacijama koje se koriste i za koje se pružaju usluge.

5.2. Aspekt razvoja životnog ciklusa softvera

Razvoj softvera osnovna je faza u životnom ciklusu softvera, kada se realizacija proizvoda izvodi na temelju poslovne ideje. Razvoj softvera je iterativni proces. Prije isporuke softvera nastupa klasična faza razvoja softvera, dok se nakon isporuke softvera, u sklopu održavanja postojećeg proizvoda, mogu dogoditi razvojne aktivnosti. Razvoj životnog ciklusa softvera obuhvaća utvrđivanje aktualnih problema te planiranje, dizajn, izradu i testiranje aplikacije. U ovoj etapi veoma važnu ulogu imaju razvojni inženjeri i autori softvera (Microsoft Docs, 2020).

5.3. Aspekt održavanja softvera u životnom ciklusu softvera

Održavanje softvera od strane korisnika počinje nakon isporuke, iako se početno održavanje događa i prije isporuke kako bi se vidjele mogućnosti i karakteristike novog proizvoda. Nakon toga započinje faza praćenja softverske aplikacije i njezino održavanje. Tijekom održavanja postoje aktivnosti koje uključuju razvoj i izmjenu postojećeg softvera, nakon čega se javno mijenja verzija i ažurira nova verzija za upotrebu. „Održavanje obuhvaća implementaciju aplikacije i održavanje neobaveznih i zavisnih tehnologija“ (Microsoft Docs, 2020).

5.4. Isporuka softvera

Isporuka softvera nije samo trenutak postavljanja softverskog sustava kako bi ga korisnici mogli koristiti, već uključuje i druge aktivnosti koje bi trebale osigurati učinkovito korištenje softvera. Tomašević (2017, str. 173) napominje da je glavni cilj isporuke osigurati učinkovit i produktivan rad korisnika nakon isporuke, kao i učiniti da se korisnici osjećaju ugodno u radu sa softverom. Osim instalacije, isporuka softvera može uključivati konfiguraciju i postavljanje sustava, dostavu dokumentacije i obuku korisnika za korištenje sustava.

Da bi krajnji korisnik mogao lako i učinkovito koristiti sustav, potrebno je ranije isplanirati i razviti sredstva koja bi mu u tome pomogla. Kako navodi Tomašević (2017, str. 173), sredstva koja najviše mogu doprinjeti boljem i lakšem radu korisnika su: treninzi odnosno obuka za korištenje softvera i dokumentacija.

5.4.1. Obuka korisnika

Softverske sustave koriste dva tipa korisnika: krajnji korisnik (koristi softverski sustav da bi riješio probleme opisane u projektnim zahtjevima) i operater (vrše dodatne aktivnosti koje osiguravaju učinkovito korištenje softvera od strane krajnjih korisnika).

Obuka krajnjih korisnika uključuje detaljnu obuku za upotrebu određenih segmenata softvera, a prema opisu posla koji obavlja korisnik. Bitno je da korisnik dobro razumije posao koji obavlja te da softver pruža podršku za svakodnevne aktivnosti. Primjer obuke je uporaba softvera za rad s određenom vrstom zapisa (entiteta) i uključuje sve osnovne operacije koje se mogu izvesti nad zapisom: dodavanje novog zapisa, izmjena postojećeg zapisa i uklanjanje postojećeg zapisa.

Obuka operatera provodi se s ciljem upoznavanja operatera kako softver radi odnosno njegovim načinom funkcioniranja kako bi mogli učinkovito održavati sustav i pružati podršku običnim korisnicima. Prema Tomašević (2017, str. 175) obuka operatera obuhvaća:

- Aktiviranje novog sustava i njegovo puštanje u rad;
- Konfiguraciju sustava;
- Upravljanje pravima pristupa sustavu;
- Raspodjeljivanje resursa korisnicima i
- Praćenje performansi sustava.

Tijekom obuke korisnicima je potrebno pružiti pomoć kako bi se postupno, ali detaljno upoznali sa sustavom. Metode koje pružaju pomoć tijekom obuke su:

- Korištenje formalne dokumentacije (organizirana je najčešće kao kolekcija dokumenata ili uputstava koji su dostupni korisnicima i koji sadrže sve informacije potrebne za ispravan i učinkovit rad sustava);
- Lekcije i demonstracije (veoma koristan i učinkovit način obuke jer su vrlo dinamične i fleksibilne. Ovakav pristup osigurava trenutne povratne informacije od korisnika što može pomoći i biti korisno za prevazilaženje nastalih problema);

- Napredni korisnici (korisnici koji su prethodno obučeni da softverski proizvod koriste na naprednom nivou. Oni najbolje razumiju potrebe korisnika tako da mogu dati najbolje izvještaje o tome koliko su korisnici zadovoljni sustavom, da li ima potrebe za dodatnom obukom, što je bilo najlakše a što najteže u savladavanju načina funkcioniranja sustava).

5.4.2. Dokumentacija

Dokumentacija se priprema kao pomoć pri korištenju softvera, za njegovo održavanje i kao pomoć pri obuci. Isti dokument obično nije prikladan za različite sudionike u životnom ciklusu softvera. Na primjer, dokumentacija koja sadrži tehničke pojedinosti o dizajnu softvera prikladna je za inženjere održavanja softvera, ali nije prikladna za krajnje korisnike koji ne razumiju razinu tehničkih pojedinosti. Stoga se dokumentacija obično priprema za određenu skupinu ljudi koji imaju posebnu ulogu u životnom ciklusu softvera. Tako se može razlikovati dokumentacija pripremljena za softverske inženjere, krajnje korisnike, operatore, menadžment ili tržište. Ovisno o publici, u životnom ciklusu softvera postoji nekoliko vrsta dokumentacije:

- Uputstvo za korisnika (veoma važan dokument, jer od njegove razumljivosti i sistematičnosti u velikoj mjeri ovisit će kako će se korisnik osjećati dok bude koristio sustav. Treba osigurati brzo i jednostavno pronalaženje svih potrebnih informacija.);
- Uputstvo za operatere (format ovog dokumenta veoma je sličan korisničkom uputstvu, ali s drukčijim sadržajem. Ovaj dokument sadrži informacije o performansama sustava, pristupu sustava, hardverskoj i softverskoj konfiguraciji sustava, postupcima arhiviranja podataka od značaja i sl.);
- Uputstvo za instalaciju (dokument u kojem je, korak po korak, objašnjen postupak instalacije softvera);
- Opći vodič kroz sustav (opisuje što sustav radi, ali bez detaljnog objašnjenja pojedinačnih funkcija);
- Uputstvo za programere (dokument u kome su detaljno objašnjene programske komponente i njihov odnos prema funkcijama sustava) (Tomašević, 2017, str. 178-180).

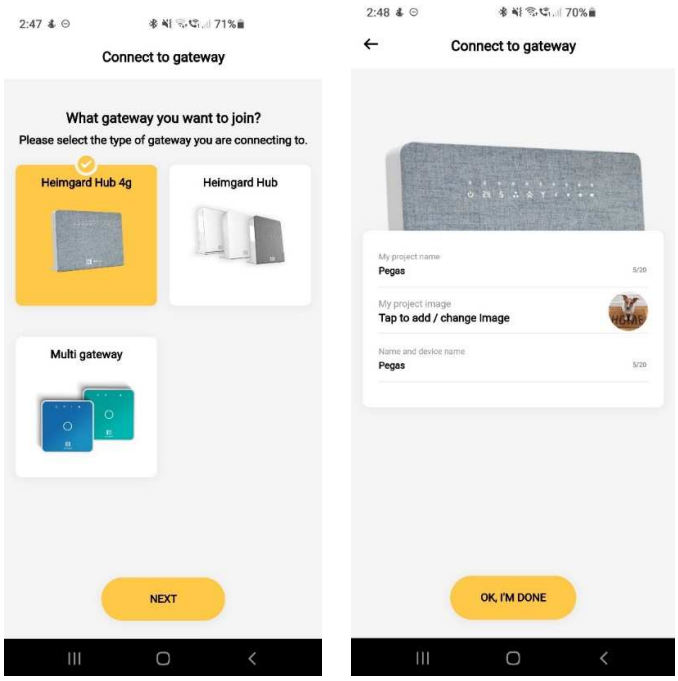
6. STUDIJA SLUČAJA: RAZVOJ PROGRAMSKOG PROIZVODA TVRTKE HOME CONTROL

U sljedećem primjeru se prikazuje stvaran softver kojem je zadaća zadovoljiti korisničke potrebe te postaviti nove standarde u korištenju pametnih električnih uređaja. Glavni cilj ove studije je provesti softver kroz razvojne faze te utvrditi aktivnosti kojima se bavi razvojni tim. Ovaj primjer će nam jasno prikazati konkretnu primjenu agilnih metoda razvoja softvera te razlog zašto se sve više razvojnih timova odlučuje za takve metode razvoja proizvoda. Primjer će obuhvaćati sljedeće faze: analizu i definiranje zahtjeva, oblikovanje (dizajn), implementaciju programa, validaciju te održavanje. Također biti će navedeo neliko verzija proizvoda kako bi uvidjeli potrebu za stalnim održavanjem i poboljšanjem proizvoda kao i glavne funkcionalnosti sustava.Cilj ovog primjera je konkretno prikazati koliko je bitna pojedina faza razvoja kako bi jedan softverski proizvod doživio uspijeh te kojim se sve trikovima služe iskusni programeri kako bi ispoštovali dogovorene rokove te zadovoljiti sve potrebe korisnika.

6.1. Tvrtka Home control

Tvrtka koja je započela razvoj kao mali start-up IoT (Internet of Thins) u Norveškoj 2003. godine te s razvojnim timom u Zagrebu prije nešto više od pet godina. godine s konceptom osiguranja domova. Danas tvrtka predstavlja svoju treću generacju IoT platforme koja obuhvaća kućnu atumatizaciju, glasovnu kontrolu i sigurnost video nadzora. Iako ne postoji mnogo korisnika takve ili slične usluge, smatra se da budućnost leži upravo u njihovom proizvodu te da će u budućnosti svaka stambena jedinica imati sličnu podršku.Tvrtka daje korisniku na rapolaganje prilagođenu platformu koja koja sigurno povezuje sve uređaje na isplativ i besprijekoran način. Poseban fokus stavljaju na privatnost korisničkih podataka, implementiranje sigurnosnih protokola komunikacije, fleksibilnost sustava te konstantna želja za poboljšanjem. Danas tvrtka broji oko četrdeset i pet zaposlenika. Heimgrad je glavni naziv primarnog proizvoda unutar tvrtke koji će nam poslužiti kao primjer. Proizvod se sastoji od nekoliko dijelova:

- Aplikacija (iOS I Android)
- Ruter za upravljanje pametnim uređajima (slika 7.)
- Cloud rješenje za spajanje

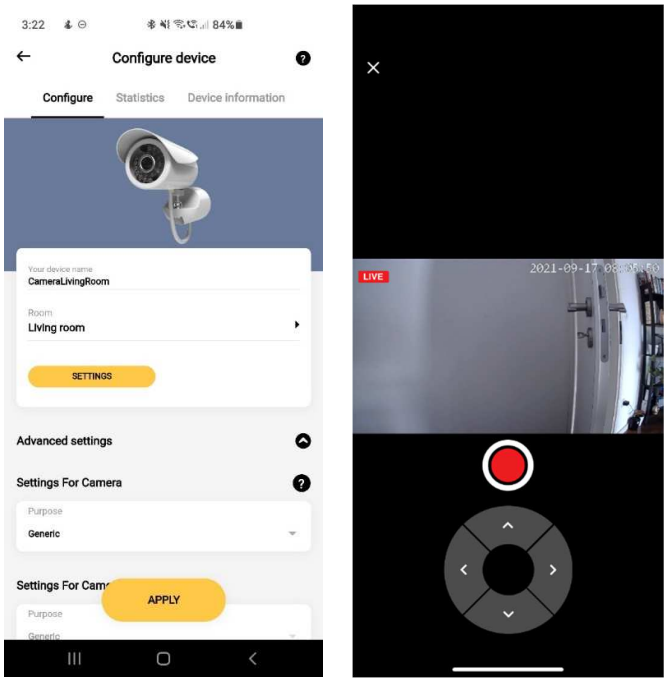


Slika 7. Selektirani gateway (lijevo), postavke selektiranog gateway-a (desno)

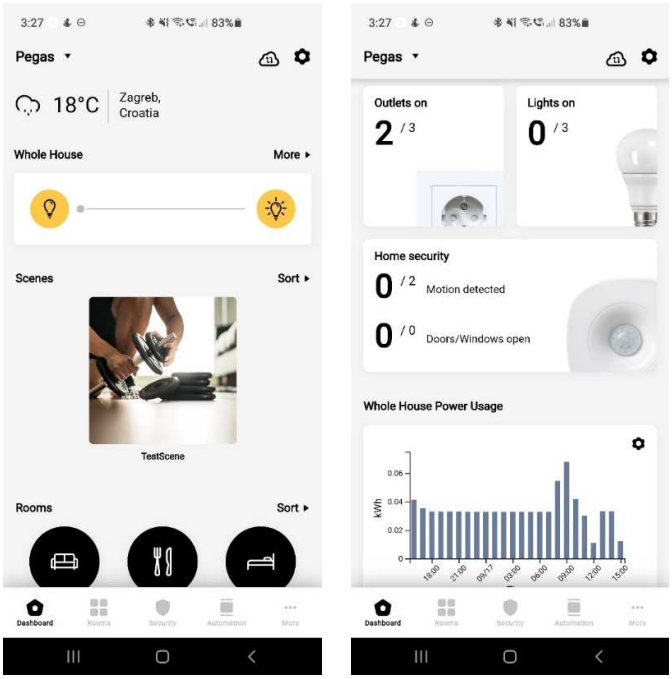
Prilikom opisa ćemo se primarno držati opisa aplikacija međutim većina funkcionalnosti se proteže kroz cijeli sustav, a razvojni procesi su unificirani na razini tvrtke.

Glavne funkcionalost sustava su:

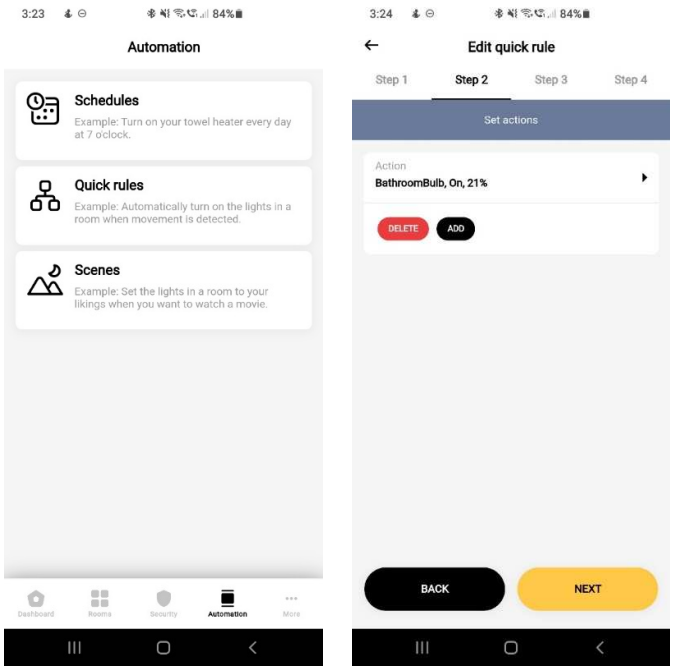
- Kreiranje i administriranje soba (slika 11.);
- Skeniranje pametnih uređaja;
- Kontroliranje pametnih uređaja;
- Prikaz statistike pametnih uređaja;
- Nadzor kamerama (slika 9.)
- Snimanje kamera, reprodukcija i skidanje snimki (slika 9.);
- Integracija s Google Voiceom (upravljanje uređajima glasovnim naredbama);
- Kreiranje i administriranje scena (slika 13.);
- Kreiranje i administriranje rasporeda (vremensko upravljanje uređaja);
- Kreiranje i administriranje “brzih pravila”



Slika 8. Nadzor kamerame (lijevo), snimanje kamere (desno)



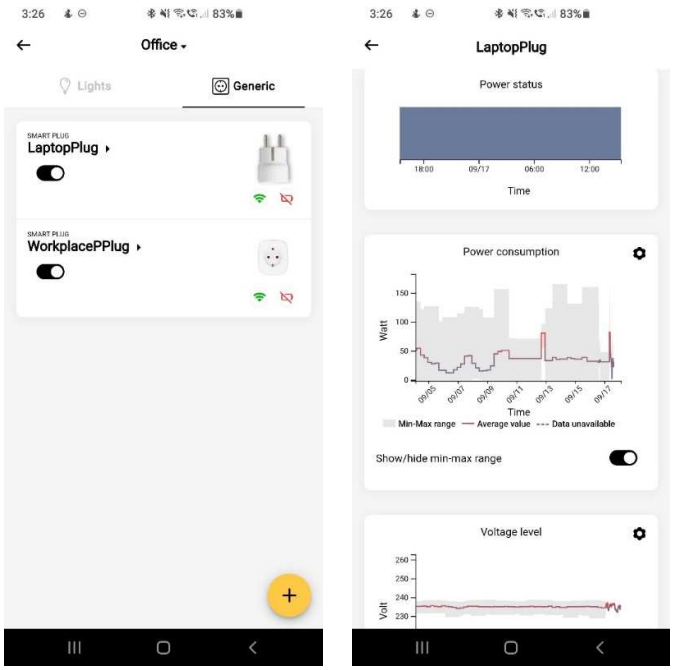
Slika 8. Kreiranje i administracija soba (lijevo), prikaz statistike pametnih uređaja (desno)



Slika 9. Kreiranje i administriranje scena (lijevo), administriranje "brzih pravila" (desno)

Primjeri pametnih uređaja koji su podržani u sustavu su:

- Utičnice (slika 15.);
- Kamere;
- Brave;
- Detektori pokreta;
- Magnetski senzori za vrata i prozore;
- Prekidači;
- Sirene;
- Statistika potrošnje struje;
- Upravljači infracrvenih signala (kontroliranje klima i TV-a)



Slika 10. Lista utičnica unutar sobe (lijevo), statistika potrošnje struje (desno)

6.2. Planiranje

Razvoj se planira u trodtjednim fazama, sprintovima. Sprintovi mogu biti organizirani od strane višeg managementa te se na takvim sastancima određuje prioritet izvršavanja pojedinog zadatka. Sastanci su strukturirani na način da zaključci moraju biti veoma precizni kako nebi došlo do nerazumijevanja zadataka. Obično je to kombinacija rješavanja zadataka sa dugotrajnije liste novih većih zadataka odnosno uvođenje novih funkcionalnosti u aplikaciju, međutim češći je slučaj da se rješavaju manji problemi, sitna poboljšanja te ispravke grešaka prijavljenih od strane testera ili korisnika. Zatim dolazimo do faze određivanja prioritizacije koja se održava sedam dana prije početka sprinta. To daje dovoljno vremena da se sprint isplanira te napišu jednoznačne instrukcije za kompleksnije zadatke. Kod takvih zadataka specifikacije uključuju priče korisnika, očekivana ponašanja potencijalno tehničke detalje te izgled pojedine funkcije(designekrane). Priče korisnika u razvojnoj fazi imaju veliku ulogu te moraju biti veoma konkretne, precizne, sa mnogo detalja kako bi razvojni tim što bolje razumio zahtijeve te samim time ubrzao razvojnu fazu, pisanje koda. Prilikom planiranja samih sprintova viši management u dogovoru sa voditeljima timova za svaki pojedini projekt određuju određeni broj bodova te se za svakog člana tima određuje gornja i donja granica bodova koji moraju biti ostvareni u pojedinom sprintu. Sakupljanje bodova se može organizirati na više načina, a jedan od načina je dodjela bodova članovima

tima prema dnevno ispunjenim zadacima. Navedeni bodovi služe voditeljima timova za procjenu stanja u kojem se pojedina faza sprinta trenutno nalazi, jesu li svi članovi tima jednako razumili korisničke priče odnosno napreduje li projekt po planu te na kraju planiranje završetka sprinta. Također bodovi služe kako bi se vodila statistika predhodnih spritova te pomoću tih podataka voditelji timova lakše mogu organizirati buduće sprintove i odrediti točniji rok isporuke.

Unutar procesa planiranja sljedećeg sprinta analizira se napredak na trenutnom sprintu te se po potrebi rade dodatne korekcije. Po završetku aktualnog sprinta zapisuju se metrike koje smo ranije naveli, a odnose se na bodove koje su dodjeljene timu te svakom članu tima za kasnije analize napredka tima.

6.3. Razvoj

Svaki sprint započinje s dužim sprint sastankom na kojem se diskutira focus sprinta te najbolji plan za rješavanje specifičnih zadataka. Programeri se potiču da sami odabiru taskove na kojima žele raditi unutar sprinta te se time pokušava trenirati proaktivnost i kritično razmišljanje.Svaki dan postoje scrum sastanci, deset do petnaest minuta gdje se prati napredak svakog člana tima. Po potrebi ti sastanci se mogu i produžiti međutim ukoliko neka specifikacija nije dovoljno jasna ili programer nije upoznat s tim dijelom sustava komunikacija se prebacuje na direktan poziv kako bi ostatak tima mogao nastaviti s poslom. Na direktnom pozivu programmer zajedno sa voditeljem tima raspravlja i donosi zaključke o korisničkim pričama u cilju boljeg razumijevanja zadatka kako bi priču mogao prenjeti u programski kod. Po završetku zadatka otvara se zahtijev za spajanje koda (Pull request). Prilikom otvaranja zahtjeva koristi se predložak koji potvrđuje da je programer odradio sve po dogovoru. Prije odobravanja zahtijeva iskusni programeri pregledaju kod te funkcionalno testiraju napravljeni zadatak. Ukoliko se radi o jednostavnijem zadatku dovoljna je potvrda od strane jednog programera međutim kod složenijih zadataka zahtijevaju se potvrde od nekoliko programera kako bi se smanjile šanse problema u produkciji.Zadaci koji se mogu pokriti s testovima implementiraju se na taj način međutim uvijek je potrebno provesti testiranje s fizičkim uređajima što značajno usporava razvoj.

6.4. Isporuka

Isporuka za produkciju se planira nekoliko mijeseci unaprijed. Verziju koja se isporučuje za korisničku upotrebu diktiraju planovi razvoja, uspiješno provedeni testovi te mnogi drugi faktori. Verzioniranje ima veliki faktor prilikom praćenja kontinuriranog razvoja proizvoda te korelacije prijavljenih problema s obzirom na verziju produkta. U ovom slučaju proizvoda uz mobilnu aplikaciju postoje i drugi faktori koji moraju biti aktualizirani, slučaju Heimgrad aplikacije to je (getway) pristupna stanica pomoću koje se upravlja svim pametnim uređajima koje su spojene na taj uređaj te riješenje za spajanje (Cloud). Stoga je bitno voditi računa da svi uređaji rade na aktualnoj verziji kako nebi došlo do možebetnih konfuzija prilikom korištenja aplikacije. Format kojim se tvrtka koristi prilikom verzioniranja jest 2021.7.2 gdje je 2021 godina verzije, 7 jest mjesec verzije te 2 broj inkrementa.Svaka planska isporuka za produkciju počinje internom isporukom timu za testiranje. Oni kroz dogovorene termine testiranja (obično četrnaest dana) prolaze kroz sustav te sistematski testiraju sve funkcionalnosti.Ukoliko dođe do nepredviđenih problema, proces se ponavlja sve dok testni tim ne potvrdi stabilnost sustava. Po uspješnom zaključivanju testa aplikacije se stavljaju u produkciju na korištenje korisnicima preko Google Play (Android) i AppStore (iOS) sučelja.

6.5. Primjeri verzija u produkciji

Verzija 2021.1.4

- Implementirana nova verzija dohvaćanja podataka o klimatskom vremenu;
- Promjena postupka kreiranja scena;
- Maknuta restrikcija broja kamera koje se mogu dodati u sustav;
- Ispravke prijevoda;
- Sitnije ispravke

Verzija 2021.5.5

- Rebranding u Heimgard aplikaciju;
- Novi dizajn;
- Implementiran sustav za brzo upravljanje uređajima;
- Animacija za aktivaciju alarma poboljšana;
- Dodane slike za gumbe;
- Ispravke na incijalnom tutorijalu;

- Popravljeno skidanje kamera snimaka;
- Standardiziranje fontova kroz aplikaciju;
- Ispravke prijevoda

Ukoliko dođe do grešaka u rada aplikacije tada korisnici prijavljuju poteškoće korisničkoj podršci (*costumer support*). Postoji više načina prijavljivanja poteškoća: prijavljivanje na web mijestu, kontaktirati direkto korisničku podršku, prijava problema putem same aplikacije ili ocijenjivanje aplikacije uz osvrt. Kada korisnička podrška procesuira prijavljeni problem kontaktira voditelje tima te ih obaviještava i upućuje na pogreške. Tada se rade modifikacije ili se formira promjena sprintova te se dogovaraju određena rješenja koja će se implementirati sljedećom verzijom sustava.

6.6. Zaključak studije

Cilj ovog istraživanja je prikazati stvaran softver te način i metode koje se koriste kako bi se došlo do gotovog proizvoda. HomeControl tvrtka već duži niz godina razmatra rješenja pomoću kojih bi ponudili što kvalitetniju uslugu svojim korisnicima. Uvođenjem agilnih metoda u razvoj softvera u počecima su se koristili Kanban agilnom metodom međutim poučeni iskustvom već duži niz godina se koriste Scrum modelom jer im taj način rada garantira maksimalnu fleksibilnost te brza isporuka proizvoda. Ukoliko dođe do pogreške u pojedinoj fazi razvoja, ova metoda razvoja im vrlo jednostavno omogućava povratak na predhodne zadatke ne ometajući danji razvoj sustava. Uz sve navedeno dolazi se do zaključka da se Agilne metodemogu primjeniti u manjim razvojnim timovima te iako precizno definirane u praksi i stvarnom životu su ponešto drugačije jer postoji mnogo čimbenika utječu na svaku od razvojnih faza.

7. ZAKLJUČAK

Za učinkovit rad putem računala i strojeva potreban je kvalitetan softver za izvršavanje namjenskih zadataka. Ako softver nije dizajniran prema standardima i postupcima koje treba slijediti u razvoju softvera, to može dovesti tvrtku ili organizaciju do ruba neuspjeha. Od mnogih standarda postoji metoda koja se naziva životni ciklus razvoja softvera. Kako bi se izbjegao kvar, bitno je da softver prati neke procese životnog ciklusa softvera. Zadovoljstvo korisnika ovisi o razvoju kvalitetnog softvera, pa je za zadovoljenje zahtjeva kupaca i kvalitetnog softvera potrebno znati korištenje procesa životnog ciklusa razvoja softvera.

Ovaj rad je imao za cilj prvenstveno razmotriti, prije svega, s teorijskog stajališta životni ciklus softverskog proizvoda kao metodologiju koja upravlja njegovim cijelim razvojnim procesom. Životni ciklus pokriva sve faze softvera od njegovog početka pa do isporuke korisnicima. Razlikuju se pet faza životnog ciklusa softvera: utvrđivanje zahtjeva (specifikacija), oblikovanje (design), implementacija (programiranje), verifikacija i validacija te održavanje odnosno evolucija. U sljedećoj fazi zahtjevi se pretvaraju u strukturi ili plan, oblikuje se dakle građa sustava odnosno projektira rješenje koje određuje kako će softver raditi. Implementacija/kodiranje počinje kada programer dobije dokument o dizajnu. Dizajn softvera preveden je u izvorni kod. Sve komponente softvera implementirane su u ovoj fazi. Verifikacija i validacija predstavlja fazu u kojoj se provjerava radi li softver prema specifikaciji odnosno prema postavljenim korisnikovim zahtjevima. U ovoj fazi, razvijeni softver se temeljito testira kako bi se otkrile i ispravile greške u sistemu. Nakon postavljanja proizvoda u proizvodno okruženje, programeri će se pobrinuti za održavanje proizvoda, tj. za rješavanje bilo kakvog problema ako se pojavi u budućnosti.

U ovom se radu proučavaju različiti modeli životnog ciklusa razvoja softvera kao što su vodopadni, spiralni, inkrementalni, iterativni, prototipi, V-model, agilni modeli.Modeli životnog ciklusa razvoja softvera omogućavaju učinkovito planiranje i praćenje procesa razvoja softvera korak po korak, čineći ga što je moguće predvidljivijim.

LITERATURA

1. Awad, M. A. (2005). *A comparison between agile and traditional software development methodologies*. Crawley: University of Western Australia.
2. Čubranić, D., Kaluža, M., Novak, J. (2013). Standardne metode u funkciji razvoja softvera u Republici Hrvatskoj. U: *Zbornik Veleučilišta u Rijeci*, 1 (1), 239-256.
3. Dora, S. K., Dubey, P. (2013). Software development life cycle (SDLC) analytical comparison and survey on traditional and agile methodology. *National monthly refereed journal of research in science & technology*, 2 (8), 22-30.
4. Javanmard, M., Alian, M. (2015). Comparison between Agile and Traditional software development methodologies. *Cumhuriyet Science Journal*, 36 (3), 1386-1394.
5. Manger, R. (2005). *Softversko inženjerstvo – skripta*. Zagreb: Prirodoslovno matematički fakultet Sveučilišta u Zagrebu.
6. Manger, R. (2016). *Softversko inženjerstvo*. Zagreb: Element.
7. Milovanović, S. (2017). Primena agilnih metoda razvoja softvera u mobilnim tehnologijama. *Infoteh-Jahorina*, 16, 698-702.
8. Osivčić, J. (2019). *Metode i metodologije u testiranju softvera*, Završni rad. Travnik: fakultet informacionih tehnologija Internacionalnog Univerziteta Travnik.
9. Pfleeger, S. L., Atlee, J. M. (2006). *Softversko inženjerstvo*, Treće izdanje, Beograd: Računarski fakultet.
10. Podreka, P. (2018). *Model životnog cirkusa softvera - sekvencijske i agilne metode*, Završni rad. Pula: Fakultet informatike Sveučilišta Jurja Dobrile u Puli.
11. Popović, J. (2019). *Osnove softverskog inženjerstva*. Beograd: CET Computer Equipment and Trade, Računarski fakultet.
12. Prašo, M., Junuz, E., Hamulić, I. (2016). *Upravljanje softverskim projektima*. Mostar: Univerzitet „Džemal Bijedić“ u Mostaru.
13. Sommerville, I. (2011). *Software engineering*, Ninth edition. Boston: Addison-Wesley.
14. Stanišević, I., Marković, V., Pavlović, V., Obradović, S. (2013). Niskobudžetni životni ciklus razvoja softvera, *Infoteh – Jahorina*, 12, 635-640.
15. Šimunović, D. (2017). *Analiza primjena agilnih metoda u razvoju softvera kod IT tvrtki s obzirom na globalne trendove*, Diplomski rad. Split: Ekonomski fakultet Sveučilišta u Splitu.

16. Şaykol, E. (2012). An Economic Analysis of Software Development Process based on Cost Models. International conference on eurasian economies, 2-9, Dostupno na: https://www.researchgate.net/publication/281270700_An_Economic_Analysis_of_Software_Development_Process_based_on_Cost_Models (13.08.2021.)
17. Tadić, B. (2005). Ekstremno programiranje i primjena na Balkanu. *4. Naučno-stručni skup sa međunarodnim učešćem*, Fojnica, 237-246.
18. Tierno, A., Santos, M., Arruda, B. (2016). Open Issues for the Automotive Software Testing, *12th IEEE International Conference on Industry Applications (INDUSCON),* Dostupno na: https://www.researchgate.net/figure/V-Model-life-cycle-for-the-automotive-software-testing_fig1_314665883 (pristupljeno 12.08.2021.)
19. Tomašević, V. (2017). *Razvoj aplikativnog softvera*. Beograd: Univerzitet Singidunum.
20. Veseli, I. (2007). *Sustavi za praćenje i vođenje procesa: Proces razvoja softvera*, Završni rad. Zagreb: Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu.
21. Microsoft Docs. (2020). *Pregled upravljanja životnim ciklusom aplikacija uz Microsoft Power Platform*. Dostupno na: https://docs.microsoft.com/hr-hr/power-platform/alm/overview-alm (pristupljeno 18.08.2021.)
22. Glavna stranica Home Controla, https://homecontrol.no/#1
23. Heimgard glavna stranica, Dostupno na: https://heimgard.com/ Android aplikacija
24. https://play.google.com/store/apps/details?id=no.homecontrol.wattle
25. IOS aplikacija, Dostupno na: https://apps.apple.com/no/app/wattle/id1335562102

POPIS SLIKA