VAŽNOST I NAČIN PRIMJENE SDLC OKVIRA U RAZVOJU SOFTVERSKIH RJEŠENJA

Šuško, Marija

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

VAŽNOST I NAČIN PRIMJENE SDLC OKVIRA U RAZVOJU SOFTVERSKIH RJEŠENJA

Marija Šuško

Zagreb, veljača 2020.

Student vlastoručno potpisuje Završni rad na prvoj stranici ispred Predgovora s datumom i oznakom mjesta završetka rada te naznakom:

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, datum.

Ime Prezime

Predgovor

Obrazovanje je kao ljestvica mogućnosti i ponekad se strmo i teško uspinjemo, ali uvijek se isplati uložiti sav napor. Studiranje na Visokom učilištu Algebra je zahtijevalo puno truda, ali svjesna sam da bez pomoći mnogih nikada ne bih uspjela biti tu gdje jesam. Prije svega, htjela bih izraziti zahvalnost svim profesorima i asistentima što su dijelili svoje znanje ove 3 godine, Danielu Beleu bez čijeg entuzijazma i motivacije nikad ne bih razvila takvu strast prema programiranju, svom mentoru i uzoru Renatu Barišiću zbog njegova sjajna vodstva i podrške, međunarodnom odjelu na Algebri za pomoć u stjecanju novih iskustava i znanja iz drugih dijelova svijeta, i konačno, mojim roditeljima što su bili uz mene te mi vjerovali i pružali podršku sve ove godine.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Životni ciklus razvoja softvera može se definirati kao vodič koji daje detaljan opis kako ispuniti korisnikove zahtjeve i ciljeve, planirati, izgraditi i održavati softversko rješenje te na taj način osigurati visoku kvalitetu i učinkovitost cijelog projekta. Konvencionalne aktivnosti SDLC okvira koje pomažu u ispunjavanju zahtjeva klijenta distribuiraju se kroz šest glavnih SDLC faza. U prvoj fazi prikupljanja i analize korisničkih zahtjeva, koju obično provode stariji članovi tima, moraju se prikupiti i razumjeti svi korisnikovi zahtjevi kako bi se analizom mogao kreirati dokument za daljnji dizajn i razvoj željenog softverskog rješenja te time pružiti veća mogućnost uspješnosti projekta. U fazi dizajna se, prema dokumentu o specifikacijama korisničkih zahtjeva, detaljno dizajniraju sistemski i softverski dokument kako bi se što lakše i uspješnije mogla izvršiti iduća i najduža faza u kojoj se konačno, načinom ovisnim o odabranoj platformi i programskog jeziku, razvija i gradi softversko rješenje. Iako tijekom razvoja proizvoda programeri provjeravaju i testiraju softver kako bi provjerili rad aplikacije, potrebni su timovi testera koji će, u SDLC fazi testiranja, kreirati scenarije za provjeru radi li cijela aplikacija bez grešaka u skladu sa zahtjevima kupca. Konačno, nakon uspješno provedene faze testiranja razvijenog softverskog rješenja koje je ispunilo sve korisničke zahtjeve, isto se pušta u rad, postaje dostupno publici i započinje faza održavanja softverskog rješenja, u kojoj se popravljaju prijavljeni *bugovi*, ažurira i nadograđuje softversko rješenje, a faza traje sve dok ono postoji. Navedene faze se pojavljuju i odvijaju na različite načine u brojnim SDLC modelima od kojih će neki biti detaljnije obrađeni u ovom završnom radu.

Software development life cycle can be defined as a guide that provides a detailed description of how to meet client's requirements and objectives, plan, build and maintain a software solution and thus ensure high quality and effectiveness of the entire project. The SDLC framework conventional activities, that help in meeting client requirements, are distributed through six main SDLC phases. In the first phase of collecting and analysing user requirements, usually performed by senior team members, all user requirements must be collected and understood to enable the analysis to create a document for the further design and development of the desired software solution and thus provide a greater chance of project success. In the design phase, a system and software document is designed in detail, according

to the user requirements specification document, so that the next and the longest phase can be completed as quickly and successfully as possible, in which the software solution is finally developed and built in a manner dependent on the chosen platform and programming language. While developers are validating and testing software during product development to test the application, test teams are required to, in the SDLC testing phase, create scenarios to verify that the entire application is bug-free according to customer requirements. Finally, after successfully completing the testing phase for the developed software solution, which eliminated all user requirements, the software is released, made available to the public and the software solution maintenance phase begins, in which the reported bugs are repaired, the software solution is updated and the phase is completed. while the software solution exists. Said phases appear and act in various ways in numerous SDLC models, some of which will be covered in detail in this bachelor thesis.

Ključne riječi: SDLC, faze, modeli, korisnički zahtjevi, softversko rješenje, planiranje, analiza, dizajn, razvoj, testiranje, puštanje u rad, održavanje.

Sadržaj

1. Uvod

U današnje vrijeme, zahvaljujući digitalnom svijetu u kojem živimo, web stranice više nisu mogućnost, nego nužnost za marketing i uspješno poslovanje. Web ima puno veći doseg nego bilo koji drugi oblik oglašavanja. „55% ljudi će prije kupovine pretraživati online recenzije i preporuke (https://wpforms.com/the-ultimate-list-of-online-business-statistics/, pristupano 25.10.2019.)“. Stoga, ne korištenjem web stranica, gubi se veliki broj mogućnosti u poslovanju, što posebice vrijedi za mala poduzeća: „19% vlasnika malih poduzeća, koji su bez web stranice, vjeruju da bi njihov posao porastao za 25% u tri ili manje godina kada bi imali prednost web stranica (https://wpforms.com/the-ultimate-list-of-online-business-statistics/#online-businesses, pristupano 25.10.2019.)“. Upravo su zato traženi web developeri kako bi, putem interneta, na profesionalan način predstavili kompanije i njihove proizvode potencijalnim korisnicima. Cilj takvih web stranica, koje će privući i zadržati korisnike, je omogućiti kompanijama konkuriranje na tržištu. No biti dobar web dizajner nije jedini preduvjeti za stvaranje kvalitetnih web stranica. Naime, važno je imati i sposobnost razumijevanja korisnikovog viđenja konačnog proizvoda i prikupljanja ključnih informacija za uspješnu implementaciju onoga što im je potrebno. Često korisnici nisu u poziciji davati zahtjeve jer nisu sigurni što točno žele ili pak ne razumiju u potpunosti način izrade traženih web stranica. U tom slučaju, potencijalne procjene i korisni prijedlozi developera su neiskoristivi, ako su utemeljeni na neusavršenim korisnikovim zahtjevima. Ignoriranje situacije, gdje su specifikacije web rješenja utemeljene na korisnikovim nerazjašnjenim ciljevima te kretanje u implementaciju takvog rješenja je upravo ono što će osigurati neuspjeh projekta. Također, napori web developera na redizajnu ili preuređivanju stranica su uzaludni ako ih korisnik nije zahtijevao, a opet treba biti spreman na promjene tijekom implementacije koje su gotovo neizbježne.

Sve su to mogući problemi i poteškoće kod planiranja i razvoja web rješenja za čije je razrješavanje potreban životni ciklus razvoja sustava (eng. Software development life cycle, skraćeno SDLC). Aktivnosti SDLC-a su usmjerene k određenom cilju te kao takve ne slijede metodologiju jednake veličine za sve nego se prilagođavaju potrebama korisnika. Prije kretanja u razvoj novog projekta potrebno je utvrditi koji su sve zahtjevi korisnika i identificirati kako će SDLC pokriti sve zahtjeve da se postigne najbolji rezultat. Zatim je potrebno odabrati najbolji SDLC model, ili kombinaciju modela, kako bi pristupili najboljem

načinu izvršavanja SDLC-a. Cilj SDLC procesa, pa tako i ovog završnog rada, je istaknuti najbolji način razvoja web rješenja, što će biti i prikazano kroz konkretan primjer izrade i implementacije web rješenja.

2. SDLC

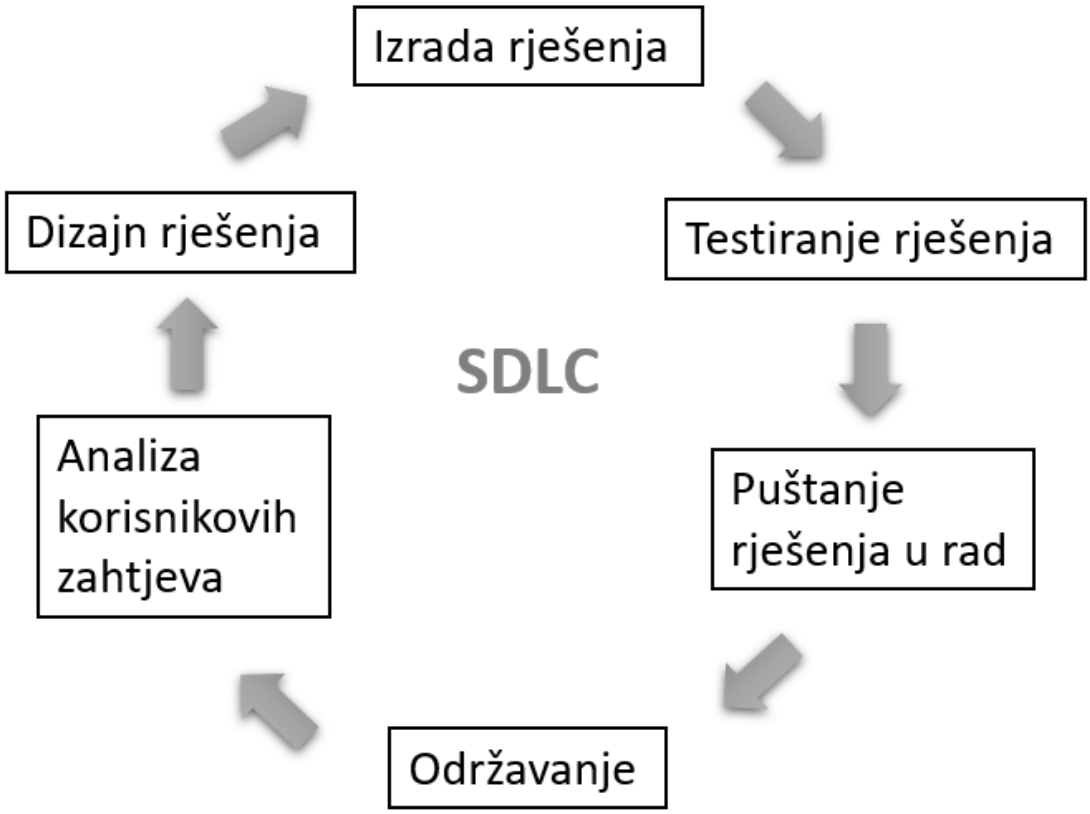
Životni ciklus razvoja softvera prvi put se „pojavio 1960-ih kao način za razvoj funkcionalnih poslovnih sustava velikih razmjera (https://online.husson.edu/software-development-cycle/, pristupano 03.11.2019.)“, a danas se razvio u okvir koji u svakoj svojoj fazi, od prikupljanja korisničkih zahtjeva do puštanja u rad i održavanja, osigurava sve veću uspješnost u razvoju softverskih rješenja.

2.1. Opis okvira

„SDLC je proces koji se sastoji od niza planiranih aktivnosti za razvoj ili promjenu softverskih proizvoda (https://www.tutorialspoint.com/sdlc/index.htm, pristupano 03.11.2019.)“. Tijekom godina, različitim definicijama se pokušavalo što preciznije opisati SDLC okvir, ali temelj svakoj je bio isti, a to je skup aktivnosti, politika i procedura, odnosno kontroliranih faza SDLC-a, usmjerenih k zadovoljenju kupca traženim rješenjima. Prva faza SDLC okvira, koju čine prikupljanje i analiza korisnikovih zahtjeva, a služe postavljanju obujma projekta te planiranju unaprijed i predviđanju budućeg upravljanja projektom, početni su ključ uspjehu čitavog projekta.

„*If you fail to plan, you are planning to fail* (Benjamin Franklin)“. Većina organizacija koje, od cjelokupnog vremena koliko traje neki projekt, ne potroše dovoljno vremena na prvu fazu imaju stopu neuspjeha projekta od 70% (https://4pm.com/2019/05/26/project-failure/, pristupano 03.11.2019.). Razlog tome je veliki obujam projekta, a nerijetko i veliki broj timova koji, ako nemaju mehanizam za upravljanje i predodređene faze za praćenje razvoja projekta, neće znati koliko su popraćeni korisnikovi zahtjevi i kako napreduju k izvršenju projekta. Stoga je, prije nego što se uopće krene u razvoj novog projekta, potrebno odrediti što sve obuhvaća SDLC i odrediti pristup web rješenju na temelju nekih od SLDC modela, najprikladnijih za praćenje i upravljanje projektom te za razvoj i održavanje željenog web rješenja.

U rijetkim slučajevima se puštanje prvog web rješenja u rad može smatrati završenim. Razvoj web rješenja neprekidnim SDLC ciklusom, kao što je predočeno ispod na *slici 1*, omogućuje pronalaženje i ispravljanje grešaka te implementiranje dodatnih funkcionalnosti.



Slika 1 Prikaz SDLC ciklusa

SDLC proces vizualno predočava projekt i određuje okvir unutar kojeg će se izvršavati te omogućuje praćenje i kontrolu rada kako bi osigurali isporuku web rješenja, one kvalitete koja je u skladu s korisnikovim očekivanjima. Za razvoj takvog rješenja web developeri koriste strukturirane faze SDLC-a. Način izvršavanja faza SDLC-a ovisi o SDLC modelu odabranom za projekt, ali neovisno o modelu, za uspješnost svake SDLC faze neophodna je uspješnost prethodne joj faze, što će na koncu omogućiti uspješnost cijelog projekta.

2.2. Modeli razvoja

Kako se tijekom godina razvijao SDLC, tako su stvoreni i različiti modeli za praćenje razvoja web rješenja. „Ovi se modeli nazivaju i Modeli Procesa za Razvoj Softvera (https://www.tutorialspoint.com/sdlc/sdlc_overview.htm, pristupano 05.11.2019.)“.

U svakodnevnom životu, ovisno o tome kakav je tko tip osobe, različitim problemima i različitim aktivnostima se pristupa na različiti način. Pa tako i kod razvoja web rješenja odabir SDLC modela će ovisiti o tipu klijenta, njegovim zahtjevima i očekivanjima te o traženom rješenju, odnosno postavljenim specifikacijama i razmjeru projekta. Svaki se

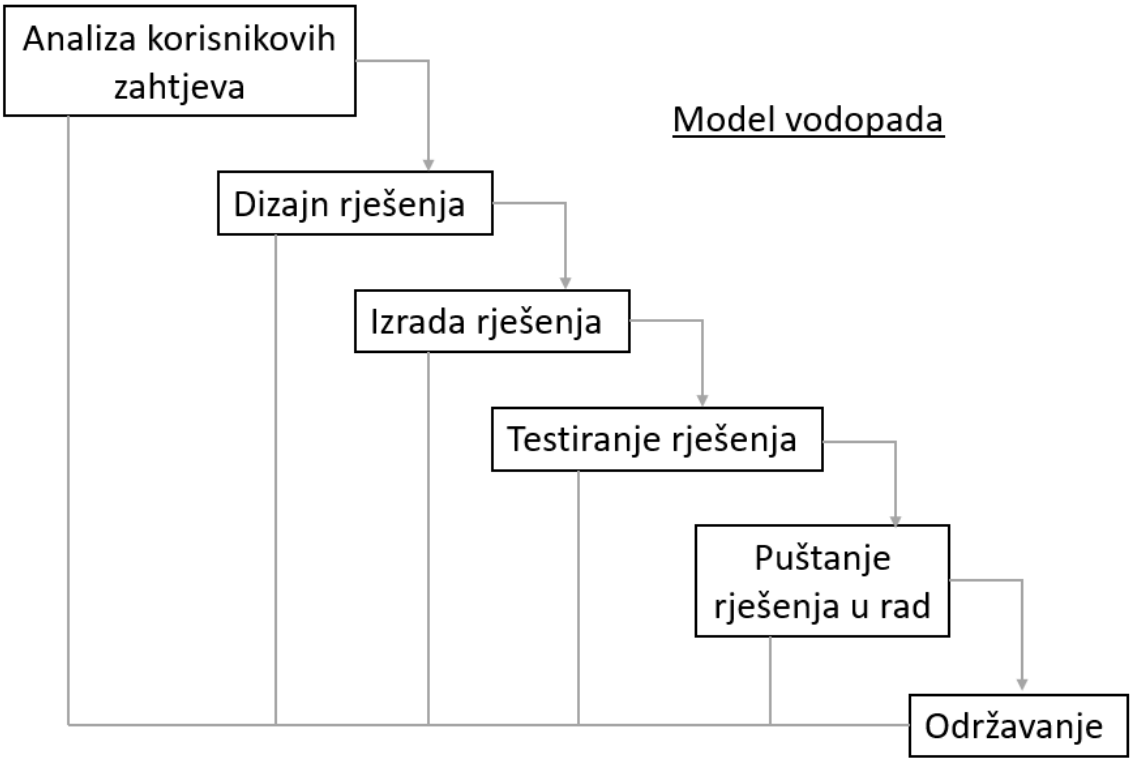
model sastoji od specifičnog izvršavanja SDLC faza i stoga svi imaju različite načine pristupa izvođenju projekta. Naravno, kako svaki model ima svoje prednosti, ima i svoje mane stoga se nerijetko za uspješnu izradu web rješenja poseže za kombiniranjem različitih SDLC modela. Ali sigurno je da će se ponavljajućim izvršavanjem svakog od modela, ili kombinacije modela, osigurati veća uspješnost i usavršavanje web rješenja.

Od pojave SDLC-a 1960-ih do danas je stvoren veliki raspon različitih SDLC modela, a među najčešće korištene spadaju:

- Model vodopada;
- V-model;
- Inkrementalni model;
- Spiralni model;
- Agilni model.

2.2.1. Model vodopada

„Model vodopada se također spominje i kao model linearno – sekvencijalnog životnog ciklusa (https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm, pristupano 06.11.2019.)“. Prvi je od Modela Procesa kojeg je uveo 1970. godine Winston Royce (https://www.guru99.com/what-is-sdlc-or-waterfall-model.html, pristupano 06.11.2019.). „To je životni ciklus koji se temelji na fazama ovisnim koracima za dovršetak implementacije sustava. Svaki korak ovisi o završetku prethodnog koraka (*Guide to Software development*)“. Iako prethodno navedena činjenica, da uspješnost pojedine faze SDLC ciklusa ovisi o uspješnosti prethodne, vrijedi za svaki od modela, za model vodopada je to važno dodatno naglasiti jer se, kako je vidljivo ispod na *slici 2*, izvršavanje njegovih faza odvija sekvencijalno kroz eksplicitno predefiniran niz vremenski odvojenih aktivnosti.



Slika 2 Model Vodopada

Preduvjet za odabir korištenja modela vodopada, za razvoj web rješenja, je dobro razrađena dokumentacija, kako bi se sa što većom sigurnošću izbjeglo kašnjenje projekta. S obzirom da su sve aktivnosti i faze vremenski odvojene i ograničene, potrebno je unaprijed dokumentirati što će se obavljati u pojedinim fazama, s prostorom za doradu svake iduće faze nakon izvršenja prethodne. Iako ne postoji eksplicitno definiran tip web rješenja za koje se može koristiti ovaj model, ipak se mogu izdvojiti određene karakteristike, prednosti i mane koje budući projekt mora zadovoljavati:

- Mora imati dokument s dobro definiranim zahtjevima i ciljevima;
- Ne smije biti preveliki opseg projekta;
- Ne smije biti funkcionalno prezahtjevan;
- Potrebno je koristiti stabilnu tehnologiju, koja se ne bi trebala promijeniti za vrijeme trajanja projekta;
- Mora koristiti resurse koji će biti dostupni tijekom cijelog projekta.

Prednosti:

- Olakšano praćenje, upravljanje i razumijevanje projekta zahvaljujući dokumentaciji;
- Olakšano određivanje potrebnih aktivnosti i njihovo izvršenje;

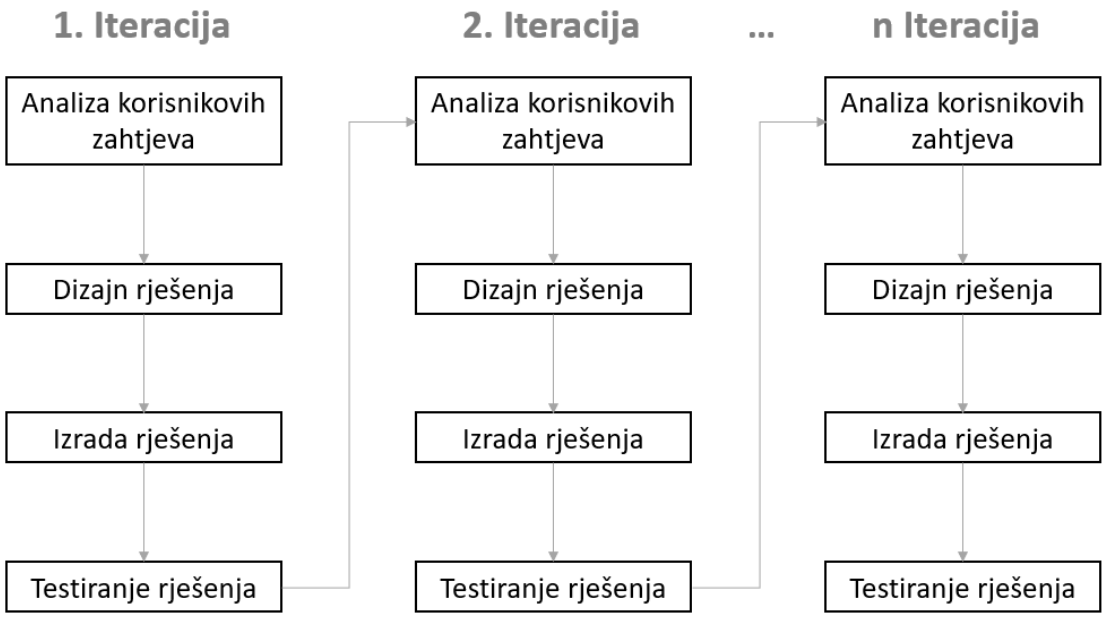
- Osigurana manja vjerojatnost kašnjenja zbog dobro određenih milestoneova.

Mane:

- Neiskoristiv je za velike i kompleksne projekte;
- Ne može se koristiti za razvoj web rješenja kod kojeg postoji mogućnost mijenjanja zahtjeva jer će to prouzrokovati promjenu svih faza te kašnjenje i povećanje troškova cijelog projekta;
- Faza puštanja web rješenja u rad se izvršava na samom kraju projekta što onemogućava feedback korisnika te moguće poboljšanje rješenja.

2.2.2. Inkrementalni model

„Inkrementalni model nije zasebni model. To je u osnovi niz ciklusa vodopada (https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano 12.11.2019.)“. Kao i kod svakog modela, korisnički zahtjevi se prikupljaju na početku projekta, ali ono po čemu se inkrementalni model ističe je činjenica da nije nužno provesti detaljno planiranje čitavog projekta, kao kod modela vodopada gdje se morala dobro razraditi dokumentacija koja opisuje kako će se odvijati projekt. Razlog tome je što se nakon prikupljenih korisničkih zahtjeva, a prije kretanja u sam razvoj web rješenja, vrši podjela i grupiranje istih po iteracijama. Svaku iteraciju čine iste faze, kao što je vidljivo ispod na *slici 3*, što znači da se svako web rješenje, koje proizađe iz određene iteracije, može smatrati završenim i potpuno funkcionalnim proizvodom. Tijekom analize korisničkih zahtjeva je također nužno postaviti opseg i granice projekta te budžet i vremenski okvir kako ne bi bilo velikih odstupanja.



Slika 3 Inkrementalni model

Ponavljanje iteracija i razvoj web rješenja inkrementalnim modelom traje sve dok se ne ispune svi korisnički zahtjevi. Isto tako, „poboljšanje proizvoda traje planiranim redoslijedom sve dok proizvod postoji (https://u-tor.com/topic/software-development-life-cycle-definitions-phases-models-and-simple-examples, pristupano 12.11.2019.)“.

Prednosti:

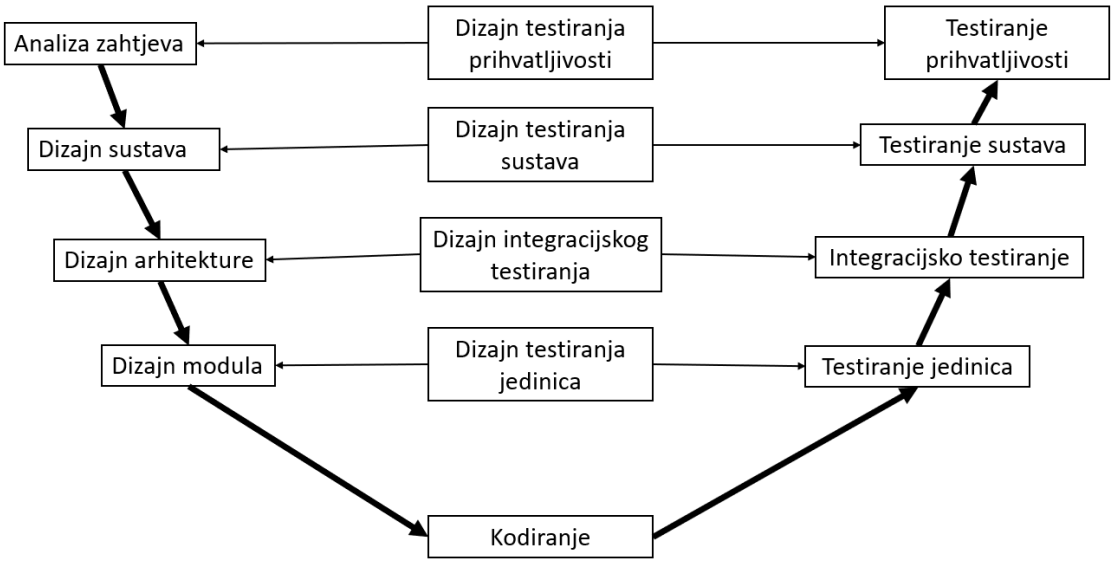
- Svaka iteracija započinje analizom korisničkih zahtjeva što omogućuje korisnikovu evaluaciju i feedback do sada razvijenog web rješenja i time sigurniji daljnji razvoj te, ukoliko se pojave, lakšu implementaciju novih zahtjeva;
- Manja je vjerojatnost kašnjenja projekta i većih prekoračenja budžeta jer je zahvaljujući iteracijama lakša implementacija novih zahtjeva i izbjegavanje rizika.

Mane:

- Potrebno je sakupiti i razumjeti sve korisničke zahtjeve te funkcionalnosti web rješenja kako bi se iste mogle podijeliti po iteracijama na samom početku projekta;
- Ukoliko se učestalo pojavljuju novi zahtjevi to može narušiti cijelu strukturu projekta te dovesti do velikog kašnjenja i prekoračenja budžeta.

2.2.3. V – model

„V-model je SDLC model gdje se izvršenje procesa odvija u sukcesivnom obliku, u obliku slova V. Poznat je i kao model verifikacije i provjere valjanosti (https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm, pristupano 15.11.2019.)“. Kao što je vidljivo ispod na *slici 4*, s jedne strane „V-modela“ se nalaze faze kojima se projekt definira, a s druge faze integracije, što je zapravo vizualna reprezentacija načina razvoja web rješenja ovim modelom, a to je paralelno izvršavanje i odvijanje testiranja i razvoja web rješenja te faza kodiranja koja ih spaja.



Slika 4 V-model

V-model je ekvivalent Modelu vodopada. Svaka sljedeća faza V-modela može započeti tek završetkom prethodne te ista omogućava i uklanjanje problema otkrivenih u prethodnoj fazi. Ono što razlikuje V-model od Modela vodopada je početak planiranja testiranja u ranoj fazi pisanja zahtjeva.

Prednosti:

- Dobar je za projekte manjeg opsega te gdje neće biti velikih i učestalih promjena korisničkih zahtjeva;
- Zahvaljujući početku testiranja u ranim fazama projekta olakšano je upravljanje rizicima te sprječavanje istih;

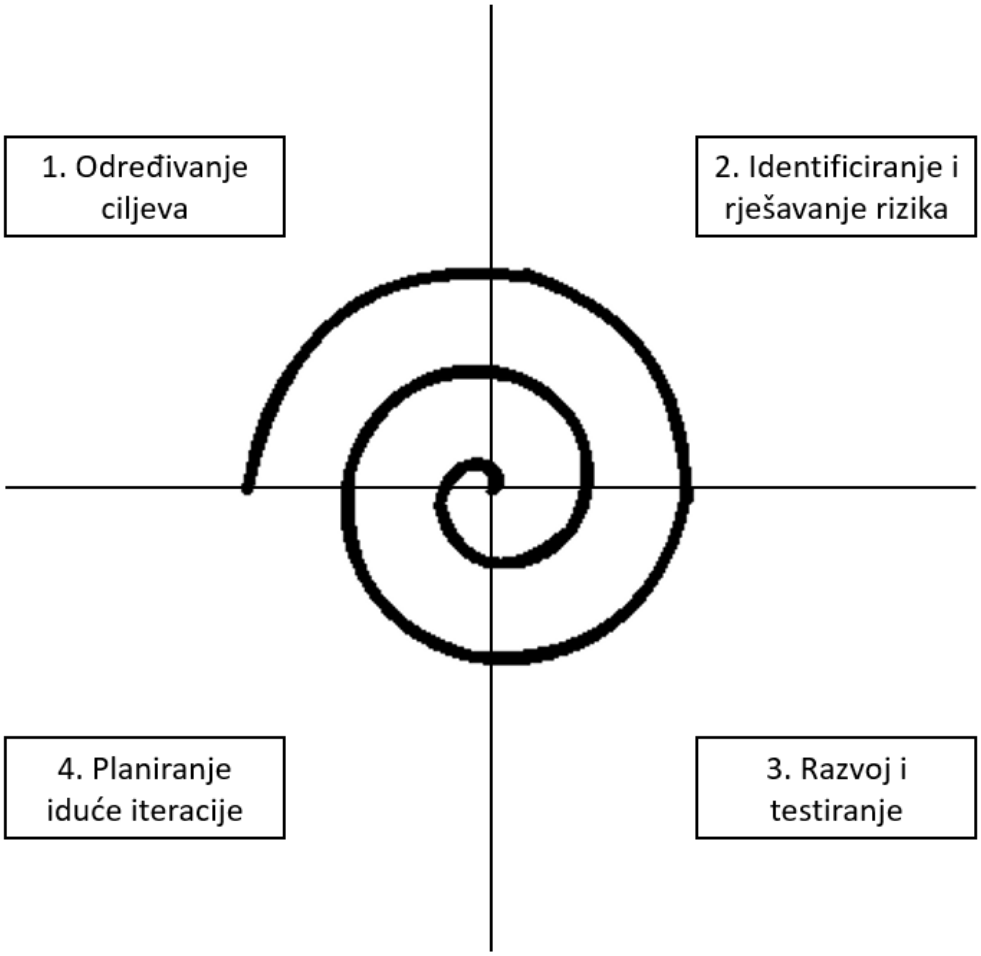
- Lakše praćenje projekta i bolja organizacija vremena zbog načina izvršavanja svih fazi.

Mane:

- Kao i Model vodopada, nije ga dobro koristiti za velike i kompleksne projekte;
- Promjene korisničkih zahtjeva će uzrokovati veliko kašnjenje projekta, posebice ako se pojave u kasnijim fazama projekta;
- Način na koji se projekt razvija ovim modelom može biti toliko dugotrajan da konačan proizvod ne bude više potreban korisniku.

2.2.4. Spiralni model

„Spiralni model je procesni model vođen rizikom (https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano 24.11.2019.)“. „… Kombinira ideju iterativnog razvoja sa sustavnim, kontroliranim aspektima Modela vodopada (https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm, pristupano 24.11.2019.)“. Ono što pruža ovakav, spiralni način, razvoja web rješenja, kombinacijom Iterativnog i Modela vodopada, je ubrzana izrada prototipa, a razlog zašto to uopće uspijeva je naglasak koji ovaj model stavlja na analizu rizika. Svaka spirala, kao što se vidi na *slici 5*, predstavlja jednu od faza SDLC-a, koje se razvijaju kao iteracije, stoga na kraju svake dobijemo testirani prototip softverskog rješenja koje se daljnjim spiralnim razvojem inkrementalno usavršava i pušta u rad.



Slika 5 Spiralni model

Svaka spirala, odnosno faza SDLC-a, prolazi kroz 4 osnovna zadatka:

- Određivanje ciljeva – ili fazom planiranja u kojoj se u prvoj spirali prikupljaju i dokumentiraju zahtjevi od kupca, a u svakoj idućoj se identificiraju zahtjevi za sustavom;
- Prepoznavanje i rješavanje rizika – prepoznavanjem, procjenom i nadzorom tehničke izvedivosti pronalazi se potencijalni rizik te se provodi analiza istog izradom, već navedenih, prototipa i pronalazi se najbolje rješenje za upravljanje pronađenim rizikom. Mogući i česti rizici su prekoračenje budžeta i/ili vremenskog roka;
- Razvoj i testiranje – nakon analize rizika, slijedi faza razvoja, tj. kodiranja i testiranja softverskog rješenja u svakoj od spirala. U prvoj spirali se kreira prvi prototip samo kao koncept kako bi dobili feedback od korisnika, a zatim se u idućim spiralama, kada su puno jasnije značajke (eng. features) i zahtjevi od korisnika, proizvodi prototip, odnosno *build* softverskog rješenja s određenim rednim brojem. „Izraz *build*

se odnosi na proces kojim se izvorni kôd pretvara u samostalni oblik koji se može pokrenuti na računalu ili u sam svoj oblik (https://www.techopedia.com/definition/3759/build, pristupano 24.11.2019.)". Naravno, svaka spirala uključuje konzultacije s korisnikom radi njegovih povratnih informacija;

- Planiranje sljedeće iteracije – korisnik bi na sastanku procijenio do tog trenutka razvijeno rješenje i dao feedback te kako bi se moglo krenuti u planiranje razvoja prototipa iduće iteracije.

Prednosti:

- Podjela razvoja na manje dijelove i izrada prototipa omogućuju bolju i uspješniju analizu i upravljanje rizicima projekta te fleksibilniji razvoj softverskog rješenja;
- Korisnici u ranoj fazi vide proizvod i više su uključeni u njegov razvoj;
- Mogućnost boljeg upravljanja zahtjevima i dodavanjima ili promjenom značajki, pa čak i u kasnijim fazama projekta;
- Prikladan za velike projekte.

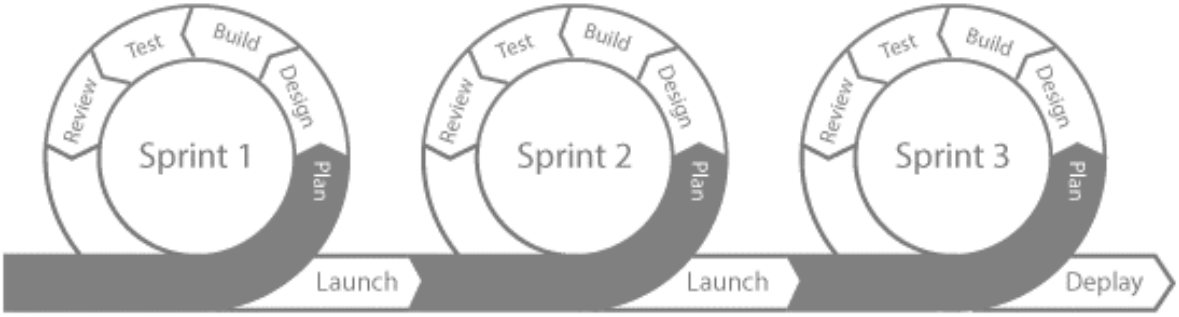
Mane:

- Zbog kompleksnosti procesa razvoja neprikladan je za manje projekte;
- Zbog čestih sastanaka s korisnikom te neprestanog pojavljivanja novih iteracija i zahtjeva može doći do znatnog povećanja troškova, kasnog određivanja krajnjeg datuma izdavanja proizvoda ili do pomicanja istog.

2.2.5. Agilni model

Agilni model je kombinacija različitih pristupa razvoju projekta, konkretno inkrementalnog i iterativnog modela. „Agilna metodologija promiče kontinuiranu interakciju razvoja i testiranja tijekom SDLC procesa bilo kojeg projekta (https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano 05.12.2019.)". Agilni model vjeruje da se postojeće metode i konvencionalne aktivnosti, koje se koriste prilikom razvoja softverskih rješenja, moraju prilagoditi projektnim zahtjevima jer se sa svakim projektom treba postupati drukčije. Cijeli projekt i njegovi zadaci se dijele na vremenske okvire, tj. manje inkrementalne *buildove*, koji se provode u već spomenutim iteracijama. Ovakav pristup razvoju softvera pruža mogućnost fokusa na fleksibilnost u razvoju, a ne na ispunjavanje zahtjeva u jednom potezu. Kao što je već objašnjeno u inkrementalnom modelu, svako

softversko rješenje, koje proizađe iz određene iteracije se može smatrati završenim i potpuno funkcionalnim proizvodom. Ista činjenica vrijedi i za softverska rješenja agilnog modela, ali se stavlja veći naglasak na ispunjenje značajki, tj. korisničkih zahtjeva predviđenih samo za tu iteraciju. Posljednja iteracija i konačan *build* traženog softverskog rješenja sadrži sve potrebne značajke i ispunilo je sve korisničke zahtjeve. Podjelu agilnog modela na iteracije možemo promatrati kao sprintove, kako je prikazano na *slici 6*, a od kojih je svaki u trajanju od 2 do 4 tjedna.



Slika 6 Agilni model

Na kraju svakog sprinta, do tada postignuti mini projekt se daje korisniku, kao vlasniku proizvoda, na uvid te se nakon njegovog odobrenja isti isporučuje kupcu. Jedan od bitnih dijelova Agilnog modela je upravo sastanak s kupcem te uzimanje u obzir njegovih povratnih informacija i prijedloga u cilju poboljšanja softverskog proizvoda u sljedećem sprintu, a time i veća mogućnost uspješnosti cijelog projekta.

Prednosti:

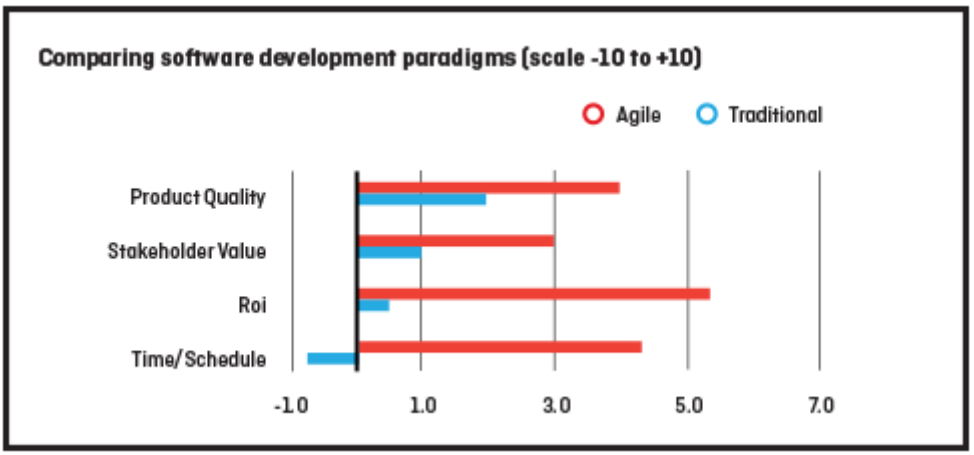
- Zahvaljujući iteracijama stvorena je fleksibilnost u razvoju te je olakšana mogućnost dodavanja novih značajki i prilagodba mogućim promjenama;
- Veća mogućnost uspjeha čitavog projekta i postizanju zadovoljstva korisnika i kupca zahvaljujući učestalim povratnim informacijama istih;
- Olakšano stvaranje i upravljanje dokumentacijom projekta.

Mane:

- Korisnik često zahtijeva previše promjena po pitanju značajki softverskog rješenja, što otežava planiranje iteracija te na kraju može dovesti do kašnjenja projekta;
- Ne preporuča se za projekte koji zahtijevaju procese velikih razmjera.

Članak https://clearcode.cc/blog/agile-vs-waterfall-method/ navodi istraživanje, provedeno od strane Ambysofta, prema kojem Agilni model ima stopu uspjeha od 64%, naspram

Modela vodopada, i njegovog tradicionalnog pristupa, kojemu je stopa uspješnosti svega 49%. Navedeno istraživanje je provedeno na temelju glavnih utjecajnih faktora projekta pod koje spadaju kvaliteta proizvodnje, vrijednost dionika, povratak investicije (eng. Return of Investment, skraćeno RoI) i vrijeme/raspored. U svakom od navedenih faktora neupitno prednjači Agilni model, kao što je i vidljivo na *slici 7*.



Slika 7 Usporedba paradigmi razvoja softvera

3. Faze i ciljevi

Kod razvoja web rješenja razlozima neuspjeha se mogu pripisati 2 glavna problema:

1. Kretanje u razvoj projekta nakon nedovoljno jasne specifikacije korisničkih zahtjeva i granica projekta;
2. Neuspješno upravljanje promjenama koje se pojavljuju tijekom odvijanja projekta.

Faze SDLC-a, predočene različitim SDLC modelima u prethodnom poglavlju, namijenjene su praćenju i upravljanju projektom. Kao takve omogućuju predviđanje i upravljanje projektnim rizicima, povećavaju učinkovitost izrade web rješenja na temelju korisnikovih zahtjeva, te osiguravaju kvalitetno upravljanje i veću vjerojatnost uspješnog izvršenja projekta, a kasnije i njegova održavanja. S tim na umu se prije kretanja u razvoj web rješenja provelo temeljito prikupljanje korisnikovih zahtjeva kako bi se razumjelo njegovo viđenje gotovog proizvoda. Kvalitetno se razvijalo web rješenje te se vodilo računa o upravljanju promjenama što je pružilo mogućnost uvođenja dodatnih funkcionalnosti koje su korisniku bile potrebne zahvaljujući sljedećim fazama:

- Analiza korisnikovih zahtjeva;
- Dizajn rješenja;
- Izrada rješenja;
- Testiranje rješenja;
- Puštanje rješenja u rad;
- Održavanje.

Nabrojene faze i konvencionalne aktivnosti se poduzimaju za provedbu različitih akcija kako bi se postigao konačan cilj, a to je zadovoljenje korisničkih zahtjeva. Svaka faza, sama po sebi, iako ključan, smislen i nezaobilazan dio u razvoju softverskih rješenja i zadovoljenju korisnika, je samo jedan mali dio velike priče koju čini SDLC. Jedino se skupom svih faza, te pridodavanjem dovoljno vremena i pažnje razvoju svake od njih, može ostvariti uspješnost i postići konačan rezultat.

4. Faza prikupljanja i analize korisničkih zahtjeva

Prvu i temeljnu fazu SDLC-a čine 2 dijela: analiza i prikupljanje korisničkih zahtjeva. Ova faza čini početak ciklusa, a započinje raspravama s korisnikom koje zakažu poslovni analitičar i projektni menadžer. Na njima sudjeluju stariji članovi tima, stručnjaci za domenu u industriji i svi dionici na projektu kako bi prikupili i zatim analizirali sve relevantne informacije koje će dati jasniju sliku opsega projekta te predviđenih pitanja, prilika i smjernica koje su uopće potaknule njegovo začeće. Već je spomenuto kako većina organizacija koje, od cjelokupnog vremena koliko traje neki projekt, ne potroše dovoljno vremena na ove dvije faze i ne razjasne s korisnikom sve detalje oko projektnih zahtjeva, imaju stopu neuspjeha od 70% (https://4pm.com/2019/05/26/project-failure/, pristupano 07.12.2019.). Stoga, kretanje u razvoj softverskog rješenja, za koji nije postignuta potpuna suglasnost s korisnikom o njegovim zahtjevima i ciljevima, neće imati učinka i u većini slučajeva će rezultirati neuspjehom.

4.1. Prikupljanje korisničkih zahtjeva

Tri su osnovna pitanja koja se moraju razjasniti sa svakim korisnikom u fazi prikupljanja zahtjeva:

- Što kupac točno želi?
- Tko će biti krajnji korisnik?
- Koja je svrha proizvoda?

Iznimno je važno razumjeti kakvo rješenje korisnik želi i steći znanja o proizvodu prije kretanja u razvoj jer se te informacije koriste u analizi korisničkih zahtjeva za planiranje osnovnog projektnog pristupa, osiguranja kvalitete proizvoda i prepoznavanja potencijalnih rizika. Naime, potencijalne procjene i korisni prijedlozi developera bili bi neiskoristivi, ako su utemeljeni na neusavršenim korisnikovim zahtjevima. Iz tog razloga je prijeko potrebno biti upoznat s vrstama klijenata i spreman na načine postupanja sa svakom od njih, kako bi se u ovoj fazi riješile sve nejasnoće. Postoje klijenti koji će zaboraviti na projekt netom nakon prvog sastanka, a postoje i oni koji će neprestano dolaziti s novim idejama i informacijama te time otežavati kontinuirani rad i razvoj. Klijent za kojeg se razvijalo višejezično web rješenje, spada u tip koji je došao na prvi sastanak s nekoliko konkretnih informacija, ali se oslanjao na to da će mu programer moći predložiti puno bolje rješenje od nečega što bi on

sam smislio i time nije bio voljan odgovarati na mnoštvo pitanja. U slučaju susreta s još jednim tipom korisnika, koji ne zna što točno želi jer ne razumije način izrade željenog softverskog rješenja, potrebno je pružiti relevantne informacije i setom konkretnih pitanja voditi ga kroz bitne točke svakog projekta, kako bi on naposlijetku uspio opisati precizne i detaljne zahtjeve.

4.2. Analiza korisničkih zahtjeva

Za kvalitetnu i uspješnu analizu korisničkih zahtjeva potrebna je suradnja i komunikacija poslovnog dijela firme i IT timova, uključujući i developere i testere. Prikupljenim informacijama od korisnika provodi se studija izvodljivosti proizvoda na različitim područjima (eng. feasibility study):

- Ekonomskim;
- Pravnim;
- Operativnim;
- Tehničkim;
- Rasporednim.

Rezultat studije tehničke izvodljivosti je mehanizam za upravljanje koji će olakšati razvojnom timu praćenje napretka projekta i izvršavanja korisničkih zahtjeva. Kako bi se, prilikom određivanja razvojnog pristupa, mogle uspostaviti razumne prekretnice (eng. milestones), poželjno je svim korisničkim zahtjevima pridružiti prioritete koji će odrediti redoslijed nadogradnje pojedinih značajki softverskog rješenja. Korisnik ima glavnu ulogu u određivanju prioriteta zahtjevima, ali je moguća preraspodjela istih od strane developera s obzirom na kompatibilnost značajki u razvoju.

Ovu fazu se može nazvati i „Fazom planiranja“ jer svaki uspješan projekt započinje planiranjem razvoja softverskog rješenja, koristeći se zahtjevima prikupljenim od korisnika, kako bi se postavio jasan opseg projekta i predočilo korisniku što ulazi u njegove granice, a čime se ista prekoračuje. Za to je ključan i nezaobilazan korak, koji čini osnovu cijelog projekta, izrada SRS (eng. Software Requirements Specification) dokumenta. Ključne informacije koje SRS mora sadržavati su:

- Koja je svrha softverskog rješenja;
- Opći opis softverskog rješenja;
- Specifični zahtjevi i značajke koje softversko rješenje treba ispuniti;

- Ako ih ima, pitanja nerazjašnjena tijekom prikupljanja korisničkih zahtjeva.

Dokument mora biti spreman za pregled na 2. sastanku s korisnikom kako bi se mogle raspraviti i razjasniti sve moguće nejasnoće te odgovoriti na pitanja od strane developera i/ili korisnika. Ovim načinom se treba osigurati da svaki sudionik projekta, uključujući neizbježno i kupca, razumije sve zahtjeve i zadatke projekta, način na koji će isti biti razvijan i održavan tijekom cijelog životnog ciklusa te služiti kao referenca koja će spriječiti moguće buduće sukobe.

5. Faza dizajna

Zahtjevi prikupljeni u prethodnoj fazi i analizirani SRS dokumentom su ulazne (eng. input) informacije za formuliranje najbolje arhitekture sustava koja će omogućiti prijelaz u sljedeću fazu gdje će se razvijati i implementirati softversko rješenje. Dokument o specifikaciji softverskog razvoja služi kao referenca za koncipiranje DDS (eng. Design Document Specification), dokumenta specifikacije dizajna softvera. Za kompletiranje DDS dokumenta razmatra se jedan ili više dizajnerskih pristupa izradi arhitekture softverskog rješenja. Posredstvom svih dionika i evidencijom utjecajnih parametara izabire se najbolji pristup. Parametri koji se uzimaju u obzir:

- kompleksnost i veličina proizvoda;
- dizajn kompatibilan s korisnikovim zahtjevima i uobičajenim standardima;
- potencijalni rješivi i nerješivi rizici;
- vremenska i financijska ograničenja.

DDS dokument jasno definira detaljan opis dizajna softverskog rješenja i svih modula predložene arhitekture. Takav dizajn se stvara postupkom odozgo prema dolje, tj. ispunjavanje zahtjeva i realiziranje značajki se kreće izvršavati na najvišoj razini te se zatim postepeno ulazi u detalje. Međutim ovakav dokument možemo raščlaniti na 2 vrste projektnih dokumenata:

- HLD (eng. High level design) – visoka razina dizajna;
- LLD (eng. Low level design) – niska razina dizajna.

HLD sadrži popis svih modula s nazivima, kratkim opisom i funkcionalnostima te opisuje njihovu međusobnu ovisnost i odnos sa sučeljem, tablice i dizajn baze podataka kategorizirane ključnim elementima softverskog rješenja te arhitektonski dijagrami protoka i strukture podataka s pojedinostima o tehnologiji.

LLD definira funkcionalnosti sustava te logiku rada, ulaze i izlaze (eng. outpute) za svaki od njegovih modula. Sadrži sve detalje sučelja te dijagrame klasa sa svim metodama i međuodnosima istih, tablicu baza podataka s njihovom vrstom i veličinom te pristupa svim pogreškama (eng. errors) i problemima ovisnosti (eng. dependency issues) koji se mogu pojaviti

5.1. Dizajn web rješenja

Fazu dizajna, kada govorimo o izradi web rješenja, možemo nazvati i fazom izrade prototipova u kojoj, nakon što se shvate svi zahtjevi, okupljaju se programeri i softverski arhitekti kako bi mogli početi dizajnirati softversko rješenje na visokoj razini. Prije kretanja u proces dizajna web rješenja na više razina te odabira najboljeg pristupa i stvaranja arhitektonskog dizajna potrebno je razmotriti:

- Koje tehnologije je najpogodnije koristiti s obzirom na zahtjeve, značajke i komponente traženog web rješenja;
- Kakva je sposobnost tima i njihov tijek rada;
- Koji su potencijalni rizici i tehnički problemi koji se mogu pojaviti te načini njihova sprječavanja ili rješavanja;
- Uzeti u obzir komunikaciju s trećim stranama (eng. third party);
- Kakav je protok korisnika (eng. user flow);
- Rad s bazom podataka;
- Kakva su vremenska i financijska ograničenja proizvoda.

Nakon razmatranja svih faktora koji će utjecati na željeno web rješenje, isti se zapisuju konvencionalnim načinom u već spomenuti DDS dokument, a veliku ulogu u dokumentaciji ima poslovni analitičar. Platforma odabrana za izradu višejezične web trgovine je WordPress. Isti je odabran iz razloga što trenutno čini više od 34% cijelog weba (https://kinsta.com/knowledgebase/what-is-wordpress/, pristupano 09.12.2019.), ima opsežnu bazu besplatnih pluginova, jednostavan je za korištenje, a isto tako pruža i mogućnost prilagođavanja web stranice potrebama korisnika pomoću HTML-a, CSS-a, Javascripta i PHP-a. Prije kretanja u dizajn također je potrebno provesti analizu rizika arhitekture kojom se provodi revizija sigurnih praksa dizajniranja i sigurnih platformi koje neće biti podložne promjenama tijekom cijelog trajanja projekta. WordPress ispunja i tu stavku jer ne spada u nove tehnologije za koje je upitno hoće li opstati za vrijeme izrade cijelog projekta, a s druge strane je popularna platforma čije web stranice prednjače na internetu.

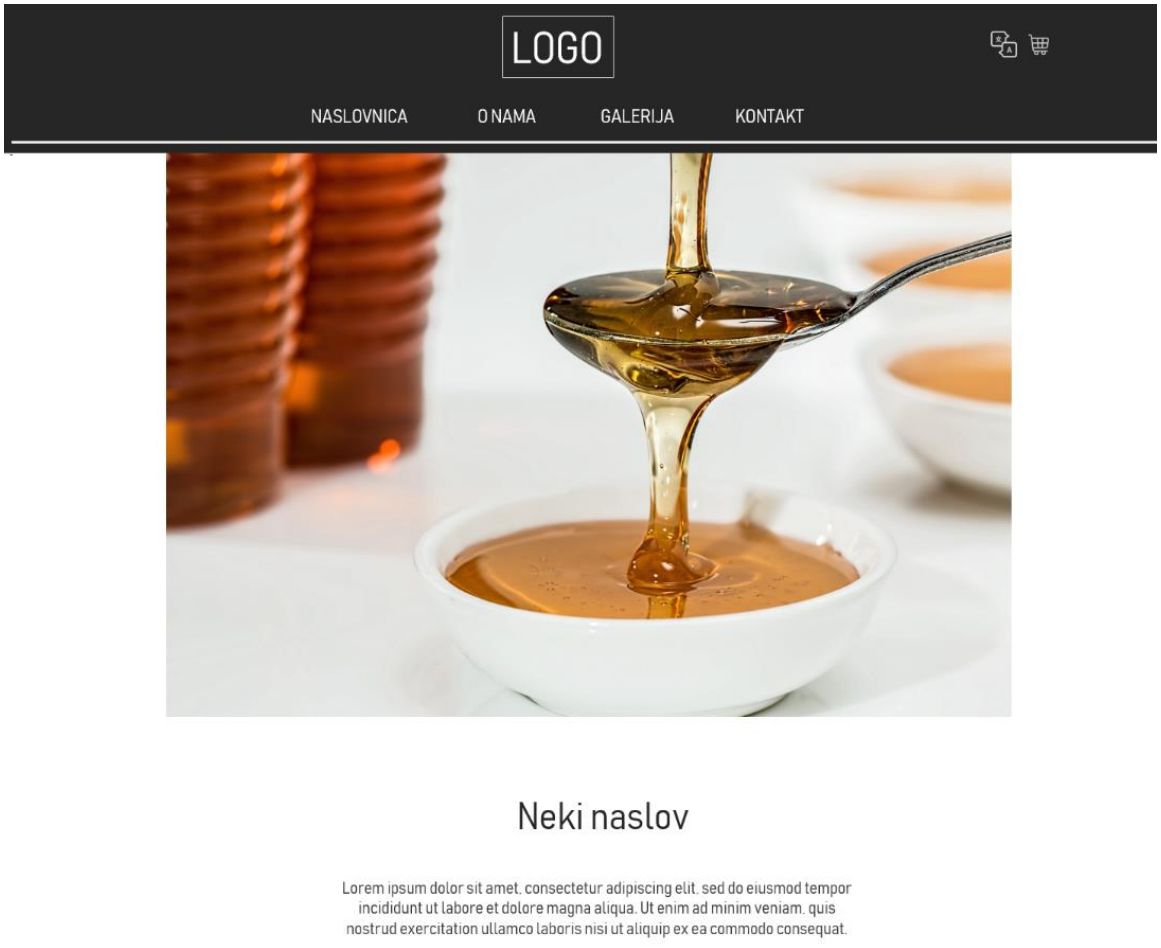
Kako bi se zatim krenulo u proces dizajna provodi se evaluacija šablona za razvoj arhitekture aplikacija, razvoja softvera i rješavanje algoritamskih problema na dosljedan način. Ova faza uključuje izradu prototipa za usporedbu rješenja kako bi se pronašlo ono najbolje koje ispunjava tražene korisničke zahtjeve i uobičajene standarde.

Iz ove faze proizlaze:

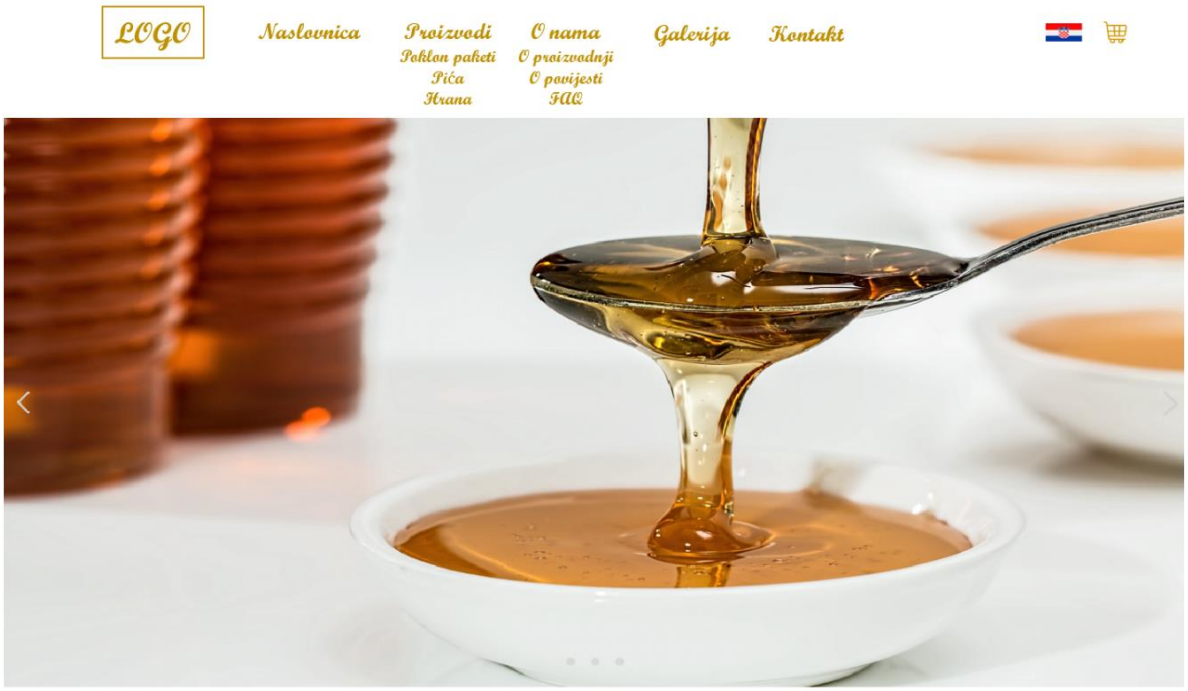
- Dizajnerski dokumenti s popisom šablona i komponenata određenih za projekt te prototip koje će se koristiti za daljnji *frontend* i *backend* razvoj;
- Počinje se s izradom strategije kojom će se testirati proizvod.

5.2. Prijedlozi dizajna

U fazi dizajna su se za korisnika, kako je i tražio, izradila 3 prototipa izgleda buduće web trgovine prema dogovorenim zahtjevima i značajkama koje bi se trebale u budućnosti implementirati. Glavni korisnikov zahtjev za izradu željene web trgovinu jest da ista na prvu ostavlja senzacionalan i luksuzan vizualni dojam. Međutim, korisnik nije imao konkretne zahtjeve za bojama i fontovima nego je prepustio iste na odabir programeru i izradu prototipa po želji smatrajući da će programer moći, u skladu s njegovim zahtjevima i standardima na tržištu, biti u mogućnosti ponuditi mu rješenje bolje nego što bi on sam smislio. Na temelju dogovora s korisnikom kreirala su se sljedeća 3 prototipa, prikazani na sljedeće 3 slike.



Slika 8 Izgled prvog prototipa



Slika 9 Izgled drugog prototipa

Slika 10 Izgled trećeg prototipa

Korisnik se odlučio na prototip broj 3 kao temeljnu vodilju za daljnji razvoj web shopa.

Kako je već navedeno i objašnjeno, odlučena je izrada web stranice u WordPressu, a s obzirom da se radi o izradi web trgovine potrebno je bilo implementirati metode plaćanja što se odlučilo riješiti pomoću Woocommercea jer je danas najpopularniji i najzastupljeniji e-commerce sustav te ga koristi preko 28% svih web trgovina (https://woocommerce.com/,

pristupano 12.12.2019.), omogućuje prodaju fizičkih i digitalnih proizvoda, raznolike opcije dostave, koje se mogu prilagoditi na različite načine za cijeli svijet i pojedine zemlje te, jedna od najvažnijih stavki, veliku pozornost posvećuje sigurnosti.

S korisnikom je također dogovoreno implementiranje SEO (eng. Search engine optimization) dodatka (eng. plugin) koji će omogućiti korisniku, odnosno web stranici korisnika, optimizaciju kako bi ista dosegla viši rang na rezultatima tražilica.

Korisnik je također zahtijevao web trgovinu na hrvatskom, engleskom i njemačkom jeziku te se višejezičnost web trgovine odlučila riješiti koristeći WPML (eng. WordPress Multilingual) plugin iz razloga što ga koriste milijuni drugih web stranica, osigurava stabilnost, sigurnost i brzinu te optimizira stranicu za SEO na različitim jezicima.

6. Faza razvoja

Nakon finaliziranja faze dizajna kreće se u iduću i najdužu fazu u kojoj započinje izravni razvoj softverskog rješenja. Nakon što developer dobije DDS dokument, s obzirom na odabranu tehnologiju za razvoj softverskog rješenja, priprema se potrebna platforma za razvoj, zadaci se razvrstavaju po jedinicama (eng. units) ili modulima te dodjeljuju pojedinim developerima, dizajn softvera se prevodi i generira u izvorni kôd (eng. source code), i to vrlo jednostavno ukoliko je dizajn realiziran detaljno i temeljito, a developeri slijede smjernice za kodiranje definirane njihovim organizacijama i programskim alatima. Developeri za izgradnju cijelog sustava odabiru, s obzirom na vrstu softvera koji se razvija, jedan od programskih jezika visoke razine, koji mogu uključivati:

- C;
- C++;
- Java;
- JavaScript;
- HTML;
- CSS;
- PHP.

Nakon odabira prikladnog programskog jezika, developeri započinju s pisanjem kôda, razvojem grafičkog sučelja i logikom interakcije sa serverom, a sistem administratori započinju postavljati softversko okruženje.

Ovisno o odabiru SDLC modela napredak ove faze, ispunjavanje korisničkih zahtjeva i značajki te usavršavanje softverskog rješenja se može odvijati postepeno u određenim vremenskim rokovima, tj. iteracijama kao što je praksa u Inkrementalnom i Agilnom modelu ili se cijelo softversko rješenje može implementirati odjednom, u ne ponavljajućim fazama razvoja, kao što je slučaj u Vodopad modelu, ali bez obzira no odabir SDLC metode, timovi developera moraju implementirati sve komponente i proizvesti funkcionalan softver u što kraćem roku.

U ovoj fazi je neizbježno biti u kontaktu s korisnikom i svim dionicima uključenim u projekt kako bili u toku s razvojem željenog softverskog rješenja te pružali povratne informacije o tome razvija li se ono u skladu s njihovim očekivanjima.

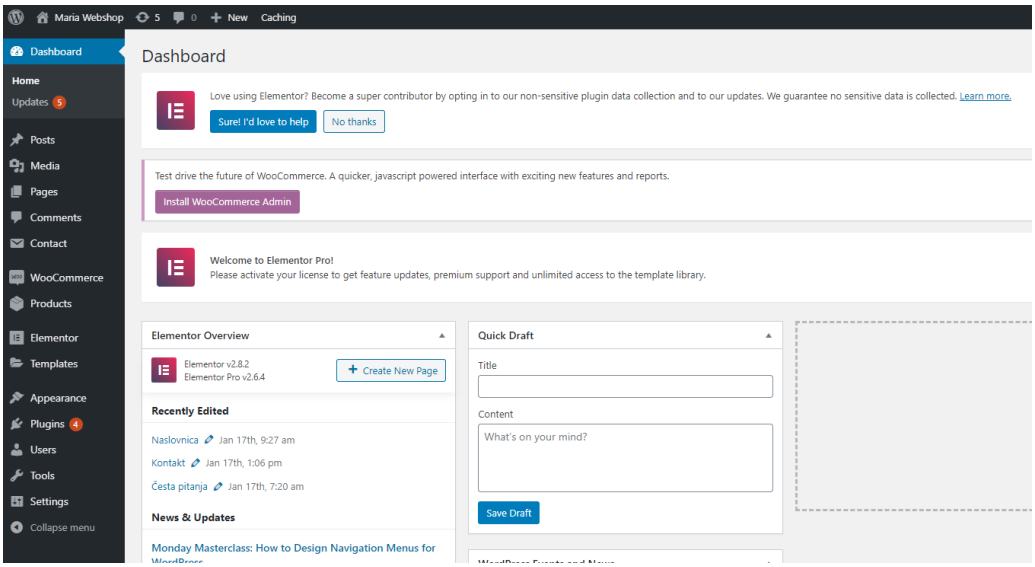
Izlaz ove faze mora biti potpuno funkcionalan proizvod, spreman za fazu testiranja, neovisno o tome jesu li ispunjene sve značajke i zahtjevi ili će se isti dodavati u sljedećim iteracijama. Kako bi se omogućila provjera rada proizvoda, testerima zaduženima za pisanje testnih slučajeva, za pregled funkcionalnosti svih komponenata razvijenog softverskog rješenja u samoj fazi testiranja, iste sastavljaju te izvršavaju također i programeri kako bi provjerili rad i ispravnost novog kôda te uopće mogli pustiti (eng. deploy) proizvedeni softver u testno okruženje.

Odabir platforme za izradu višejezične web trgovine, kao što je već spomenuto, je WordPress iz razloga što je svjetski najpoznatiji sustav za upravljanje sadržajem, skraćeno CMS (eng. Content Management System) i najpoznatiji alat za stvaranje web stranica. Izgrađen na MySQL-u i PHP-u, započeo je kao alat za samostalno vođenje blogova 2003. godine, a danas je prerastao u softver za stvaranje potpuno opremljenih poslovnih web stranica. Jedan od razloga koji čine WordPress toliko poželjnim je činjenica da je otvorenog kôda, licenciran pod GPLv2 (eng. Generic Public Licence, version 2).

Za postavljanje WordPressa potrebni su:

- Naziv domene odnosno web adresa;
- Usluga koja povezuje web lokaciju s internetom – web hosting.

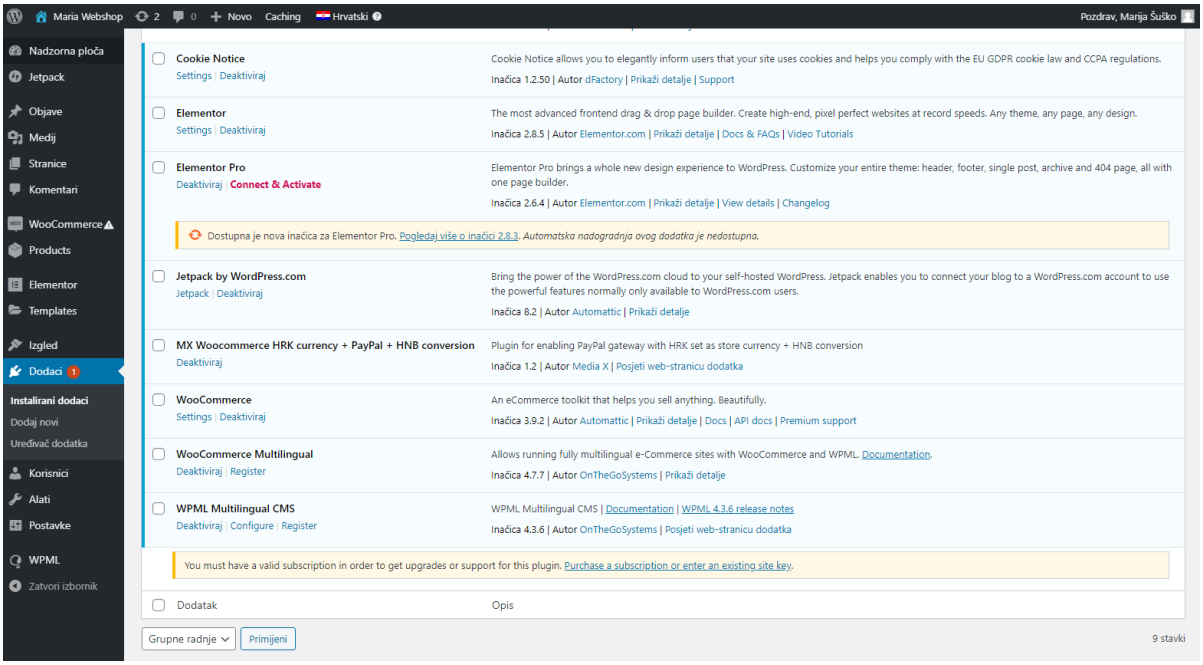
Platforma svojom fleksibilnošću omogućava developerima stvaranje prikladnih web stranica s raznim dodacima (eng. plugins) i temama, a ipak je s druge strane dovoljno jednostavna za korištenje. WordPress dolazi s intuitivnim i korisničkim sučeljem koje se lako koristi (eng. user-friendly interface), a prikazano je na slici 11.



Slika 11 Izgled kontrolne ploče WordPressa

„WordPress dodaci su male softverske aplikacije koje se integriraju i pokreću povrh WordPress softvera“ (https://www.wpbeginner.com/beginners-guide/what-are-wordpress-plugins-how-do-they-work/, pristupano 14.12.2019.). Dodatke čine funkcije koje se mogu upotrijebiti na WordPress web rješenjima, a mogu služiti za proširenje postojećih funkcionalnosti ili za umetanje novih značajki.

Postoje tisuće besplatnih dodataka dostupnih na WordPressu, kao što su WooCommerce i WPML koji su neophodni za izradu web rješenja zahtijevanog od strane korisnika, te omogućuju brže i efikasnije dobivanje izgleda konačnog rješenja uz značajne mogućnosti ugrađenih konfiguriranja funkcionalnosti. Navedene funkcionalnosti uključuju: izbor načina plaćanja, dostave, upravljanja cijenama, upravljanja akcijama, prevođenja sadržaja stranica, pozicija izbornika promjene jezika, pozicija zastavica i drugih mogućnosti. *Slika 12* prikazuje sučelje za upravljanje instaliranim dodacima.

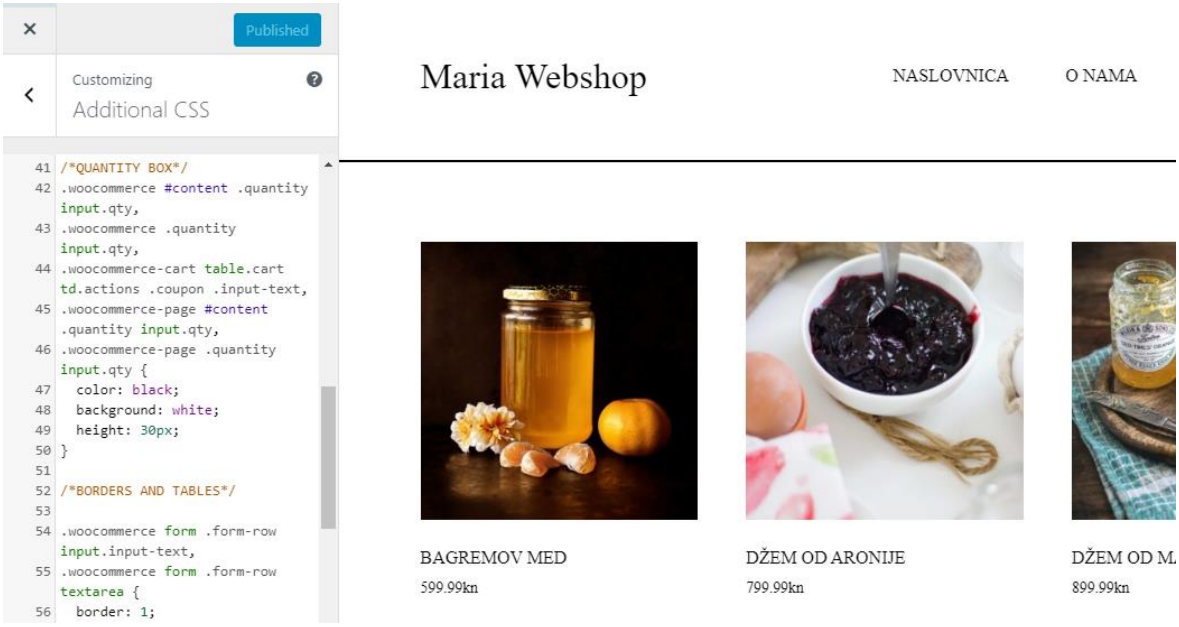


Slika 12 Dodaci korišteni za razvoj

WordPress dodaci su napisani PHP programskim jezikom te se integriraju s WordPress web rješenjima, ali se osim dodataka proizvoljni PHP kôd može dodavati i na teme te izrađene stranice, što olakšava kontrolu i pristup bilo kojem od elemenata.

Osim PHP programskog jezika, od velikog značaja je poznavanje CSS programskog jezika koji je nedvojbeno jedan od spornih načina unošenja promjena u WordPress web rješenja. Omogućuje prilagodbu izgleda web stranica, njenih fontova, boja i drugih značajki te nudi opsežniju kontrolu nad izgledom web stranica nego što to mogu ponuditi WordPress teme,

a ispod je na *slici 13* prikazan primjer korištenja CSS programskog koda na web rješenju zahtijevanom od strane korisnika.



Slika 13 Izgled CSS kôda

7. Faza testiranja

Fazu testiranja se može smatrati podskupom drugih faza SDLC modela danas jer se testiranje i konvencionalne testne aktivnosti provode u svim SDLC fazama. Nakon što programeri završe testiranje jedinica (eng. unit testing) u fazi razvoja i implementacije te proizvedu operativno te djelomično ili potpuno dovršeno softversko rješenje, s obzirom na odabrani SDLC model, ono se zatim može, zajedno sa svim svojim modulima, pustiti u testno okruženje gdje tim za testiranje započinje s provjeravanjem svih funkcionalnosti sustava i uspoređuje iste s propisanim projektnim dokumentima. Testeri se referiraju na SRS i DDS dokumente kako bi mogli provesti testiranje i pronaći nedostatke u softveru kojim će se uvjeriti da je softversko rješenje proizvedeno i radi u skladu s dokumentiranim standardima, značajkama i zahtjevima kupca. Iako je faza razvoja i implementacije softverskog rješenja najdulja, fazom testiranja se provjerava kvaliteta implementiranog proizvoda, je li isti ispunio korisnikova očekivanja, ima li nedostataka i *bugova*, „softverskih problema koji uzrokuju neispravan rad rješenja, a posljedica su pogrešne logike“ (https://www.techopedia.com/definition/24864/software-bug-, pristupano 16.01.2020.) te je li korisničko sučelje prijateljski nastrojeno (eng. user-friendly) te je stoga ova faza ključna kako bi se dobilo osiguranje da je softversko rješenje valjano i spremno za puštanje u rad.

Timovi testera su u neprestanoj komunikaciji s developerima tijekom provođenja temeljitog testiranje softvera i kada pronađu *bug* ili nedostatak softverskom rješenju, kreiraju izvješće te pronađeni problem prijavljuju i dodjeljuju nekome od programera. Programeri provjeravaju je li prijavljeni problem važeći te mu pristupaju i rješavaju ga najoptimalnijim načinom, a zatim implementiraju novu verziju softvera koju vraćaju timu testera zajedno s ažuriranom dokumentacijom na ponovno testiranje. Svaka nova verzija softvera iziskuje i nova testiranja kojima se mora utvrditi kako su ispravljeni svi pronađeni *bugovi*, a da ispravljanje pri tome nije utjecalo na druge značajke i funkcionalnosti.

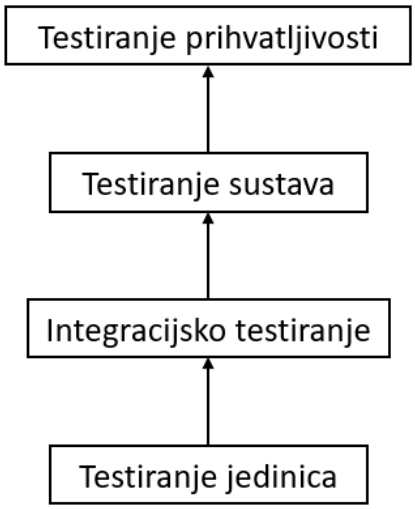
Testiranje i regresijsko testiranje softverskog rješenja se može smatrati manjim internim životnim ciklusom testiranja softverskog rješenja jer se ono ponavlja sve dok:

- tim testera ne pronađe što je moguć veći broj *bugova* i nedostatka;
- razvojni tim ne riješi sve prijavljene probleme;
- ne postigne se stabilno softversko rješenje;

- ne ispune se svi korisnikovi zahtjevi i značajke te proizvod bude u skladu s dokumentiranim potrebama softverskog rješenja.

Konačan izlaz ove faze je u potpunosti funkcionalan proizvod spreman za puštanje i uporabu u proizvodnom okruženju.

Niz testova se primjenjuje na implementiranom proizvodu kako bi se prijavili i popravili *bugovi*, a zatim isti testovi i ponovili, čime se stvara ciklus koji se provodi sve dok se ne postigne konačan proizvod koji je bez nedostataka, stabilan i radi kako je definirano projektnom dokumentacijom. Od širokog spektra testova koji se moraju primijeniti kako bi se postiglo softversko rješenje za koje se sa sigurnošću može reći da je spremno za iduću fazu, puštanja proizvoda u rad, oni osnovni i neminovni su prikazani ispod *slikom 13* te će biti pojašnjeni u idućim poglavljima.



Slika 14 Osnovna testiranja

Kako bi se moglo provesti svako od ovih testiranja prije svega se izrađuje plan testiranja, radi se osvrt na njega i po potrebi se prerađuje kako bi se stvorila osnova za primjenu testiranja. Isti postupak se ponavlja za stvaranje testnih slučajeva softverskog rješenja, a tek onda se može započeti s testiranjem koje će konačno, ovisno o softverskom rješenju za koje se izvodi, biti jednostavno ili složeno.

7.1. Testiranje jedinica

Jedinice podrazumijevaju najmanje dijelove softverskog rješenja koji se mogu odvojeno testirati, a imaju jedan ili više ulaza te nerijetko samo jedan izlaz, primjerice funkcije, metode i klase. Čini prvu razinu testiranja, a cilj je ispitati radi li svaki dio softvera i postiže li se

napisanim kôdom funkcionalnost kakvu je programer zamislio te podudara li se ista s korisnikovim željama i zahtjevima. Testiranje jedinica je temeljna i nezaobilazna razina testiranja koja, ukoliko se pravilno i uspješno provede, može kasnije značiti uštedu i vremena i novca.

Metoda, koja se koristi za provođenje testiranja jedinica, je testiranje bijele kutije (eng. White box testing). „Testiranje bijele kutije softverski je postupak testiranja, u kojem je unutarnja struktura / dizajn / implementacija predmeta koji se testira, poznata testeru… Neophodno je znanje programiranja i znanje o implementaciji. Testiranje bijele kutije testira se izvan korisničkog sučelja i ulazi u nečistoće sustava (http://softwaretestingfundamentals.com/white-box-testing/, pristupano 16.01.2020.)“. Testovi se mogu provoditi ručno ili automatizirano od strane programera koji su razvijali softversko rješenje ili programera testera. Kako bi se moglo uspješno provesti sljedeće integracijsko testiranje, nužan preduvjet je ispravljanje i utvrđivanje pouzdanog rada svakog od modula, odnosno jedinica razvijenog softvera.

7.2. Integracijsko testiranje

Nakon što su uspješno testirane pojedinačno sve komponente softverskog rješenja, iste se počinju smisleno kombinirati, odnosno integrirati i testirati kao grupa. Ovim se testiranjem pronalaze nedostatci, greške i *bugovi,* koji se pojavljuju prilikom integracije pojedinih modula softvera, a u većini slučajeva je uzrok njihovom pojavljivanju razvijanje pojedinih jedinica od strane različitih programera.

Za integracijsko testiranje se može odabrati, u prethodnom poglavlju opisana, metoda testiranja bijele kutije te metode testiranja crne ili sive kutije, ovisno o dogovoru programera i testera s obzirom na softversko rješenje koje se razvija. „Testiranje crne kutije, također poznato kao bihevioralno testiranje, softverska je metoda ispitivanja u kojoj ispitivač ne poznaje unutrašnju strukturu / dizajn / implementaciju predmeta koji se testira (http://softwaretestingfundamentals.com/black-box-testing/, pristupano 16.01.2020.). Testiranje sive kutije kombinacija je metode testiranja crne kutije i metode testiranja bijele kutije. Kod testiranja sive kutije djelomično je poznata unutarnja struktura. To uključuje pristup unutrašnjim strukturama podataka i algoritmima za potrebe dizajniranja testnih slučajeva, ali testiranje se vrši na korisničkoj ili razini okvira. (http://softwaretestingfundamentals.com/gray-box-testing/, pristupano 16.01.2020.)“.

Postoje različiti pristupi integracijskom testiranju koje mogu provoditi sami programeri koji su razvijali softversko rješenje ili programeri testeri.

- Big Bang pristup je pogodan za manje projekte jer se sve komponente, ili većina njih, testiraju odjednom;
- Inkrementalnim pristupom se spajaju dvije ili više komponenata koje su u logičkoj korelaciji, a zatim se nadodaju i testiraju druge povezane komponente, sve dok se ne testira cijelo softversko rješenje. Inkrementalni se pristup može podijeliti na metode: Odozdo prema gore (eng. Bottom Up), Odozgo prema dolje (eng. Top down) i Sendvič ili Hibridna metoda. U provođenju navedenih metoda testiranja pomažu testni pokretači (eng. test Drivers) i *test Stubs* koji simuliraju ponašanje softverskih komponenti o kojima ovisi modul na kojem se provodi testiranje.
    - U metodi Odozdo prema gore, prvo se testiraju jedinice nižih razina koje se zatim postepeno spajaju i testiraju s jedinicama viših razina, u čemu pomažu testni pokretači koji po potrebi simuliraju jedinice viših razina;
    - U metodi Odozgo prema dolje, prvo se testiraju jedinice viših razina te se postepeno spajaju i testiraju s jedinicama nižih razina, u čemu pomažu *test Stubsi* koji po potrebi simuliraju jedinice nižih razina;
    - Sendvič metoda kombinira prethodno dvije objašnjene metode te se u isto vrijeme pomoću testnih pokretača i *test Stubsa* testiraju niže komponente s višima i obrnuto.

7.3. Testiranje sustava

Nakon uspješne integracije svih komponenti softverskog rješenja provodi se testiranje sustava čija je glavna svrha ustanoviti kolika je usklađenost implementiranog softverskog rješenja s korisnikovim zahtjevima. Testiranje sustava spada u treću razinu testiranja softverskog rješenja koje izvršavaju, konvencionalnim aktivnostima testiranja, isključivo testeri, a u tome im pomaže, prethodno opisana metoda, testiranje crne kutije. Na ovoj razini testiranja, osim što se mora testirati koliko je softversko rješenje ispunilo korisničke zahtjeve, softver se povezuje s hardverom i testira se rad sustava u cjelini.

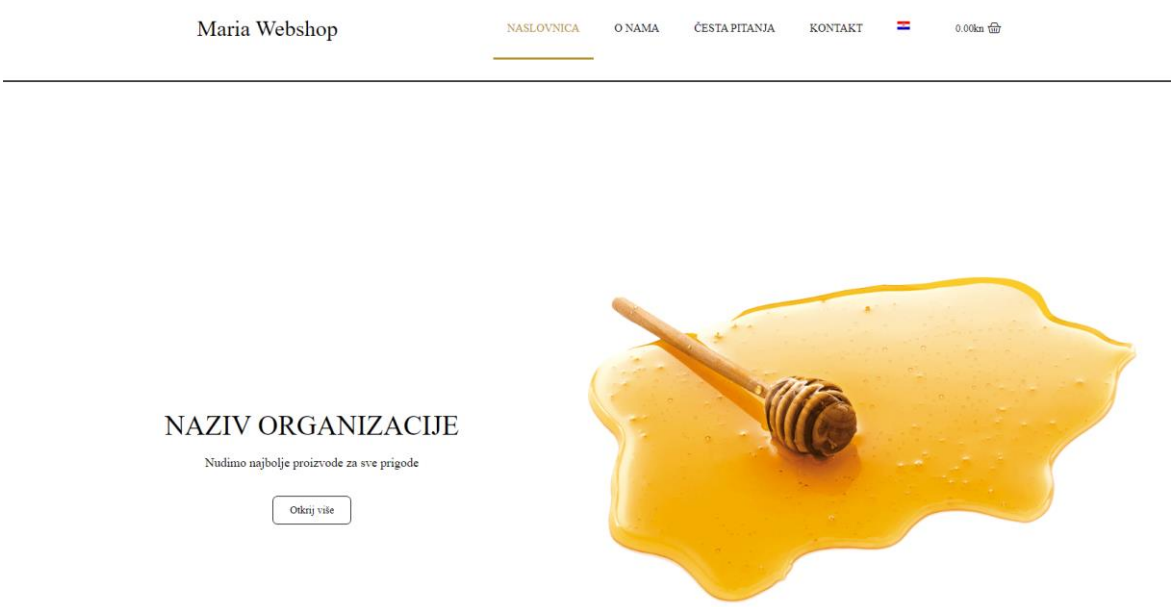
7.4. Testiranje prihvatljivosti

Po završetku uspješnog testiranja sustava, testiranjem prihvatljivosti se utvrđuje usklađenost softverskog rješenja s poslovnim zahtjevima i procjenjuje je li proizvod prihvatljiv za isporuku. Softversko rješenje se predaje korisnicima, kupcima ili nekom od ovlaštenih subjekata koji provjeravaju točnost implementiranog proizvoda u zadovoljavanju kritičnih i glavnih poslovnih zahtjeva te odlučuju hoće li se prihvatiti takav sustav. Konvencionalnim postupcima testiranja i metodom crne kutije, po potrebi se vrše testovi prihvatljivosti, nakon čega se sustav može staviti u stvarnu upotrebu. Testiranje prihvatljivosti ima sljedeću podjelu:

- Testiranje internog prihvaćanja, poznato i kao Alfa testiranje, vrše članovi projekta koji nisu bili uključeni direktno u razvoj ili testiranje softverskog rješenja;
- Testiranje eksternog prihvaćanja se provodi od strane osoba koje uopće nisu zaposlenici organizacije koja se bavila implementacijom softverskog rješenja, a ovo testiranje se još može podijeliti na:
    - Testiranje prihvaćanja kupca (eng. Customer acceptance testing) koje izvršavaju kupci koji su dali zahtjev za razvojem softverskog rješenja;
    - Testiranje prihvaćanja korisnika (eng. User acceptance testing), poznato i kao Beta testiranje, koje provode krajnji korisnici softvera. Ovo testiranje je usko povezano s fazom puštanja softverskog rješenja u rad te će stoga biti detaljnije objašnjeno u idućem poglavlju.

8. Faza puštanja softverskog rješenja u rad

Nakon što se uklone i korigiraju svi *bugovi* te uspješno provede i završi faza testiranja, tada započinje završni proces izdavanja i puštanja softverskog rješenja u rad (eng. deployment), kojem je konačan izlaz stavljanje rješenja u proizvodno okruženje, čime ono postaje dostupno publici. Cilj i izlaz ove faze je osigurati da je softversko rješenje funkcionalno kako bi se moglo pustiti i funkcionirati u živom okruženju te kako bi ga kupci mogli nesmetano koristiti. Na konkretnom rješenju izgled naslovnice u živom radu izgleda kao na *slici 15.*



Slika 15 Prikaz naslovnice gotovog rješenja

Testiran proizvod, koji je bez *bugova* i spreman za uporabu, se pušta u odgovarajuće okruženje, a cijeli proces puštanja softverskog rješenja u rad se, ovisno o poslovnoj strategiji pojedinog poduzeća i očekivanju kupca, može odvijati u fazama. Primjerice, u poduzećima visoke razine zrelosti ova se faza niti ne osjeti nego se softversko rješenje pušta u rad onog trena kada se potvrdi kao dovršen proizvod i bez grešaka, a u poduzećima srednje ili niže zrelosti, te u nekim visoko reguliranim industrijama, proces uključuje prolaz kroz nekoliko stadija i dobivanje određenih odobrenja. Ukoliko je dogovor organizacije da se proizvod izda u ograničenom segmentu i testira u stvarnom poslovnom okruženju, provodi se testiranje prihvatljivosti korisnika UAT i stvara se replika proizvodnog okruženja u kojemu kupac zajedno s programerima provodi dodatno testiranje i provjerava ima li problema s puštanjem softverskog rješenja u rad. „Ispitivanje prihvatljivosti korisnika, poznato i kao

beta ili testiranje krajnjeg korisnika, definira se kao testiranje softvera od strane korisnika ili klijenta kako bi se utvrdilo može li ga prihvatiti ili ne. Ovo je završno ispitivanje koje se provodi nakon što su funkcionalna, sistemska i regresijska ispitivanja završena (https://www.softwaretestinghelp.com/what-is-user-acceptance-testing-uat/, pristupano 17.01.2020.)“. Na temelju povratnih informacija, ako se ne prijave nikakvi problemi i kupac smatra da je aplikacija onakva kakvom je i zamišljena, proizvod se pušta u proizvodno okruženje takav kakav jest ili s predloženim poboljšanjima, ukoliko ih ima.

8.1. Aktivnosti pokretanja i pripreme puštanja u rad

U svakom slučaju, neovisno o strategiji organizacije, proces i aktivnosti pripreme softverskog rješenja za pokretanje i puštanje istog u rad, se svode na instalaciju, konfiguraciju, testiranje i po potrebi unošenje promjena, radi optimiziranja performansi softverskog rješenja. Navedene aktivnosti puštanja softverskog rješenja u rad omogućuju prije svega uštedu vremena i resursa te povećavaju sigurnost dobrim praksama. Olakšano je upravljanje softverskim rješenjem jer se mogu nadzirati aktivnosti korisnika te prepoznati, a samim time i riješiti problemi koji se pojave u radu softverskog rješenja. Istodobno, implementacija softverskog rješenja pomaže u ažuriranju istog automatiziranim pretragama potrebnih verzija. Za što uspješniju implementaciju softverskih rješenja su se tijekom godina razvile sljedeće dobre prakse:

- Implementacija popisa za puštanje softverskog rješenja u rad, kojim se stvara proces rada i izvršavanja zadataka, kako bi se osiguralo da se sprovedu svi koraci neophodni za uspješnu implementaciju softvera;
- Odabir ispravne metode za implementaciju, kojom se neće prekoračivati predviđeni budžet, a ipak će se olakšati integraciju softverskog rješenja s postojećim lokalnim aplikacijama i alatima;
- Automatizacija postupka implementacije omogućava izbjegavanje ručnog uvođenja i ažuriranja novih verzija softvera koje, osim što uzrokuje gubitak vremena, može izazvati pojavu brojnih grešaka i *bugova*;
- Usvajanje kontinuirane isporuke koja se postiže prvotno puštanjem aplikacije u prototipno okruženje, gdje se provjerava funkcioniranje aplikacije i njeno ispunjenje korisničkih zahtjeva, što konačno osigurava isporuku kôda za potrebnu implementaciju;

- Korištenje poslužitelja kontinuirane integracije omogućava izbjegavanje nerijetkih potencijalnih problema integracije i stvara veću sigurnost rada razvijenog softverskog rješenja na stroju programera te time čini ključ uspješne agilne implementacije.

Navedene aktivnosti i cijeli postupak implementacije i puštanja softverskog rješenja u rad olakšavaju prikladni alati, prilagođeni za rad na različitim platformama i vrstama infrastrukture, provođenjem automatskih ili predodređenih zadataka. Alati za puštanje softverskih rješenja u rad pogodni su posebice za same programere koji neprestano rade na poboljšanju i ažuriranju softvera za korisnike, a alati za implementaciju im olakšavaju upravo taj proces te kontinuiranu integraciju. Članak https://www.dnsstuff.com/software-deployment-tools navodi SolarWinds Patch Manager, Octopus Deploy, Jenkins i Bamboo kao 4 visoko specijalizirana alata za puštanje softverskih rješenja u rad u 2020. godini.

8.2. Prednosti puštanja u rad

Implementacija softverskog rješenja pruža mnoge prednosti od kojih prije svega treba navesti unaprjeđenje poslovanja organizacije jer ono olakšava i ubrzava poslovne procese, pomaže s upravljanjem velikim podacima, korisnici mogu pristupiti softverskom rješenju s bilo kojeg uređaja u bilo kojem vremenu te pomaže u integraciji s Internetom stvari (eng. Internet of things, skraćeno IoT) jer osigurava povezanost korisničkih i drugih aparata koji olakšavaju život korisniku. „Internet stvari ili IoT odnosi se na milijarde fizičkih uređaja širom svijeta koji su sada povezani s internetom, a svi prikupljaju i dijele podatke. Zahvaljujući dolasku super jeftinih računalnih čipova i sveprisutnosti bežičnih mreža, moguće je pretvoriti bilo što, od nečega malog poput pilule do nečega velikog poput aviona, u dio IoT-a (https://www.zdnet.com/article/what-is-the-internet-of-things-everything-you-need-to-know-about-the-iot-right-now/ , pristupano 01.02.2020.)“. Zahvaljujući provođenju implementacije konvencionalnim praksama, ista se automatizirala te omogućava automatsko ažuriranje novih verzija softvera te uvođenje softvera novim korisnicima i uređajima. Kako bi se smanjilo traženje pomoći od strane korisničke podrške, korisnicima se olakšava samostalna instalacija softverskog rješenja.

Međutim, iako se provode brojna testiranja te ispravci grešaka i *bugova* nije moguće niti jednim od načina testiranja pronaći sve uzroke problema, što fazu puštanja softverskog rješenja u rad ne čini posljednjom fazom SDLC ciklusa. Održavanje softverskog rješenja je

posljednja faza SDLC-a jer će se u njoj prijaviti i ispraviti krajnji problemi softverskog rješenja, a po potrebi uvoditi te ugrađivati nove funkcionalnosti i značajke.

9. Faza održavanja

Nakon što su ispunjeni svi korisnički zahtjevi i značajke softverskog rješenja te je provedeno testiranje i ispravljene su sve greške, proizvod se pušta u rad, postaje dostupan publici na korištenje i započinje faza održavanja softverskog rješenja. Ovom fazom ne završava SDLC ciklus jer ona traje sve dok postoji softversko rješenje i dostupno je korisnicima te ju se stoga naziva i „krajem početka“.

Softversko rješenje u radu potrebno je neprestano nadzirati kako bi se osigurao pravilan rad njegovim korisnicima. Tri su neizbježna događaja tijekom ove faze:

- Pojavljivanje, prijava te ispravljanje *bugova* i nedostataka u scenarijima koji se nisu mogli predvidjeti i riješiti u fazi testiranja;
- Nadogradnja (eng. upgrade) novih verzija softverskog rješenja;
- Poboljšanje (eng. update) postojećeg softverskog rješenja dodavanjem novih značajki.

Prilikom ispravljanja bilo kakvih grešaka i *bugova* kreiraju se izvješća koja se predaju programerima i premda se ne može ponovno prolaziti kroz cijeli SDLC ciklus, potrebna je barem skraćena verzija postupka kako bi se spriječila regresija i pojavljivanje novih poteškoća. Kupci tijekom korištenja proizvoda prijavljuju pronađene greške te se ovisno o razini poteškoće i hitnosti programeri odlučuju o vremenu njihovog popravka te nadogradnje i poboljšanja softverskog rješenja. Iako ova faza SDLC-a može biti u nekim slučajevima pasivna za razvojne timove i timove testera, neizbježno je otkriti sva pitanja i probleme koji se pojavljuju kod korisnika kako bi se osiguralo nesmetano funkcioniranje i korištenje softverskog rješenja.

Prema članku http://ecomputernotes.com/software-engineering/types-of-software-maintenance različite načine održavanja softverskih rješenja može se podijeliti u četiri osnovne skupine po načinu kojim održavaju softverska rješenja te prema postotku zastupljenosti tipa i utrošenog vremena:

- Korektivni tip održavanja čiji pristup se koristi za ispravljanje pronađenih problema, grešaka i *bugova* koje otežavaju ili onemogućavaju upotrebu softvera, zastupljen je svega 20%;
- Prilagodljivi tip, zastupljen 25%, radi na održavanju softvera kako bi ga učinilo prilagodljivim radu na novim operacijskim sustavima i novim okruženjima;

- Perfektivni tip, najzastupljeniji od čak 50%, održava softver kako bi nesmetano funkcionirao i nakon dodavanja novih značajki softverskom rješenju;
- Preventivni tip održavanja, najmanje zastupljen sa svega 5%, provodi promjene kojima se pokušava spriječiti pojavljivanje problema i grešaka.

9.1. Korektivan tip održavanja

Pod korektivan tip održavanja spada svaka aktivnost kojom se ispravlja problem, popravlja kvar, nedostatak ili greška neke od funkcija softverskog rješenja. Zadaci korektivnog tipa održavanja mogu biti planirani ili neplanirani, a mogu se pojaviti zbog grešaka u dizajnu softverskog rješenja, kodiranju i razvoju ili logici rada sustava. Prema članku https://www.fiixsoftware.com/corrective-maintenance/ iz 2020. godine, mogu se izdvojiti tri glavne situacije koje zahtijevaju akcije korektivnog održavanja:

- Kada se prilikom praćenja stanja softverskog rješenja pronađe problem;
- Kada se otkrije potencijalni problem prilikom rutinske inspekcije softverskog rješenja;
- Kada se već dogodi kvar u sustavu.

Ovisno o aktivnosti i problemu koji zahtijevaju korektivan tip održavanja, može ih se riješiti planiranim ili neplaniranim pristupom. Ukoliko se provodi strategija „Održavanja prema neuspjehu“ (eng. Run-to-failure), iskoristit će se planirano korektivno održavanje jer se u tom slučaju pušta dijelu softvera (eng. asset) djelovanje (eng. run) dok se ne pokvari kako bi se zatim moglo popraviti ili zamijeniti, uz napomenu da se ovaj pristup može primijeniti samo ako je riječ o lako i jeftino zamjenjivom dijelu softvera. Drugi slučaj planiranog pristupa je kad se korektivno održavanje kombinira s preventivnim tipom održavanja jer se potencijalni problemi pokušavaju pronaći prije nego se pojave kako bi se moglo isplanirati održavanje. Međutim, ukoliko dođe do neočekivanog i nepredviđenog prekida u radu planiranih aktivnosti zbog kvara u sustavu ili nedostupnosti osoblja i korištenih alata, te se planirani zadaci ne mogu obaviti odmah, radi se o neplaniranom korektivnom održavanju.

9.2. Prilagodljiv tip održavanja

Prilagodljiv tip održavanja softverskog rješenja se primjenjuje kad se u jednom dijelu sustava pojavi nepredviđena promjena izvana koja iziskuje promjenu i prilagodbu softverskog rješenja u njegovim drugim dijelovima, što se može odnositi na promjene u

njegovom okruženju, primjerice promjene hardvera ili operativnog sustava. Kad je riječ o vanjskom utjecaju na softversko rješenje, prvenstveno se misli na promjene povezane s poslovnim pravilima, obrascima rada i vladine politike. Promjene „okoliša“, ili drugim riječima promjene operacijskih sustava ili hardvera, se mogu dogoditi u vrlo kratkim razdobljima, a kako softverska rješenja moraju komunicirati s nekim od operativnih sustava i hardverskih platformi, neizbježno je prilagodljivo održavanje kako bi softversko rješenje opstalo. Ono može iziskivati i promjene funkcionalnosti samog softverskog rješenja kako bi se prilagodilo nastalim promjenama i izbjeglo potencijalne novonastale greške i *bugove*.

9.3. Perfektivan tip održavanja

Kada je riječ o perfektivnom tipu održavanja, ono se pojavljuje u većini slučajeva jer korisnik ima nove zahtjeve i ideje za novim funkcionalnostima i značajkama koje želi uvesti za softversko rješenje. Perfektivno održavanje stoga može, ali ne mora nužno biti potrebno i uzrokovano greškama ili *bugovima* pronađenim u trenutnom softverskom rješenju nego korisnikovim zahtjevom za poboljšanjem postojećih ili uvođenjem novih zahtjeva. To od programera zahtijeva promjenu funkcionalnosti kôda, pazeći pritom na njegovu učinkovitost, kako bi se softversko rješenje moglo prilagoditi novim potrebama i zahtjevima korisnika. Osim promjena uzrokovanih novim korisničkim zahtjevima, iste se mogu pojaviti zbog povratnih informacija kupaca softverskog rješenja. Pri tome se, osim zadovoljenja korisničkih zahtjeva i zadovoljenju kupaca softverskim rješenjem, pazi na optimizaciju brzine rada sustava, neprestano poboljšanje korisničkog sučelja i njegove upotrebljivosti. Sve su to razlozi koji objašnjavaju zastupljenost perfektivnog tipa održavanja od čak 50%.

9.4. Preventivan tip održavanja

Preventivan tip održavanja koristi se u svrhu sprječavanja manifestiranja grešaka u softverskom rješenju kako bi se spriječili mogući problemi i nepravilan rad istog. Mogući načini postizanja preventivnog održavanja softverskog rješenja uključuju smanjenje složenosti programa i kôda optimizacijom te restrukturiranjem. Sve promjene softverskog rješenja znače promjene i prikladne dokumentacije koja mora biti prilagođena te se moraju evidentirati sve promjene softverskog rješenja.

Odrađivanje aktivnosti preventivnog tipa održavanja spomenute su u kombinaciji s korektivnim tipom održavanja, ali ono osim toga može imati i druge slučajeve u kojima se koristi. Može se podijeliti na 2 osnovna slučaja u kojima se koristi:

- Aktivnosti preventivnog tipa održavanja se izvršavaju u predodređenim vremenskim okvirima, primjerice nadgledanjem sumnjivog ili poznato kritičnog dijela softverskog rješenja;
- Aktivnosti preventivnog tipa održavanja izvršavaju se nakon određenih jedinica aktivnosti upotrebe, što mogu biti prijeđeni kilometri ili ciklusi upotrebe od strane korisnika softverskog rješenja.

Zaključak

SDLC okvir, ukoliko se smisleno i valjano iskoristi, pruža izrazito veliku kontrolu upravljanja projektom i nadgledanja razvoja softverskog rješenja. Uzevši u obzir raznolikost projekata, s obzirom na njihovu kompleksnost, veličinu i različite značajke koje pojedino softversko rješenje mora ispuniti kako bi se postigli ciljevi i zadovoljstvo korisnika, tijekom godina su se razvili i usavršavali različiti SDLC modeli. Ne postoji SDLC model koji se može navesti kao univerzalan, pa čak ni Agilni model koji kombinira pristupe različitih modela i ima veliki postotak uspješnosti, nije primjenjiv na svaki tip projekta. Svaki od SDLC modela pruža konvencionalne aktivnosti i faze koje se, ovisno o modelu drukčije spajaju, dok se sam model za razvoj softverskog rješenja odabire ovisno o karakteristikama i potrebama projekta, ali rezultat korištenja svakog od modela je isti, a to je olakšano upravljanje projektom i veća mogućnost za konačnim uspjehom.

Kako se ne može izdvojiti jedan model za razvoj softverskih rješenja, tako se ni pojedina SDLC faza ne može izdvojiti kao najvažnija jer je svaka neophodna za uspješno izvršenje cijelog projekta. SDLC ciklus se razbije na šest faza i stvori se plan rada i razvoja softverskog rješenja što olakšava detaljan nadzor, praćenje i izvršavanje svake od faza, a time i uspješno izvršenje cijelog projekta. Pojedina faza može biti najduža, dok će se druga izvršiti brže, a trećoj će se morati posvetiti više pažnje. Međutim, činjenica je ako se propusti i jedna od njih može se izraditi funkcionalni proizvod, ali isti neće postići željeni rezultat i očekivano zadovoljstvo korisnika, a u nerijetkim slučajevima će doći do propadanja cijelog projekta. Svako softverskog rješenje je u početku samo ideja nekog korisnika koja, kad se provuče kroz faze SDLC-a, postaje funkcionalnim, iskoristivim i održivim proizvodom.

Popis kratica

SDLC	*Software Development Life Cycle*	Životni ciklus razvoja softvera
SRS	*Software Requirement Specification*	Specifikacija softverskog razvoja
DDS	*Design Document Specification*	Dokument specifikacije dizajna
SEO	*Search Engine Optimization*	Optimizacija pretraživača
PHP	*Hypertext Preprocessor*	Hipertekstualni predprocesor
HTML	*HyperText Markup Language*	Jezik za označavanje hiperteksta
CSS	*Cascading Style Sheet*	Kaskadna lista stilova
MySQL	*My Structured Query Language*	Moj strukturirani jezik upita
IoT	*Internet of Things*	Internet stvari

Popis slika

Literatura

[1] LANGER, ARTHUR M. *Guide to Software Development_ Designing and Managing the Life Cycle*. London, Springer, Verlag, 2012

[2] https://wpforms.com/the-ultimate-list-of-online-business-statistics/, pristupano listopad 2019.

[3] https://online.husson.edu/software-development-cycle/, pristupano studeni 2019.

[4] https://www.tutorialspoint.com/sdlc/index.htm, pristupano studeni 2019.

[5] https://4pm.com/2019/05/26/project-failure/, pristupano studeni 2019.

[6] https://www.tutorialspoint.com/sdlc/sdlc_overview.htm, pristupano studeni 2019.

[7] https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm, pristupano studeni 2019.

[8] https://www.guru99.com/what-is-sdlc-or-waterfall-model.html, pristupano studeni 2019.

[9] https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano studeni 2019.

[10] https://u-tor.com/topic/software-development-life-cycle-definitions-phases-models-and-simple-examples, pristupano studeni 2019.

[11] https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm, pristupano studeni.2019

[12] https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano studeni 2019.

[13] https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm, pristupano studeni 2019.

[14] https://www.techopedia.com/definition/3759/build, pristupano studeni 2019.

[15] https://www.guru99.com/software-development-life-cycle-tutorial.html#4, pristupano prosinac 2019.

[16] https://4pm.com/2019/05/26/project-failure/, pristupano prosinac 2019.

[17] https://kinsta.com/knowledgebase/what-is-wordpress/, pristupano prosinac 2019.

[18] https://woocommerce.com/, pristupano prosinac 2019.

[19] https://www.wpbeginner.com/beginners-guide/what-are-wordpress-plugins-how-do-they-work/, pristupano prosinac 2019.

[20] https://www.techopedia.com/definition/24864/software-bug-, pristupano siječanj 2020.

[21] http://softwaretestingfundamentals.com/white-box-testing/, pristupano siječanj 2020.

[22] http://softwaretestingfundamentals.com/black-box-testing/, pristupano siječanj 2020.

[23] http://softwaretestingfundamentals.com/gray-box-testing/, pristupano siječanj 2020.

[24] https://www.softwaretestinghelp.com/what-is-user-acceptance-testing-uat/, pristupano siječanj 2020.

[25] https://www.dnsstuff.com/software-deployment-tools, pristupano siječanj 2020.

[26] http://ecomputernotes.com/software-engineering/types-of-software-maintenance, pristupano veljača 2020.

[27] https://www.fiixsoftware.com/corrective-maintenance/, pristupano veljača 2020.

[28] https://clearcode.cc/blog/agile-vs-waterfall-method/, pristupano veljača 2020.

[29] https://www.fiixsoftware.com/corrective-maintenance/, pristupano veljača 2020.

ALGEBRA
VISOKO UČILIŠTE

NASLOV ZAVRŠNOG RADA

Pristupnik: Marija Šuško, 0321006873

Mentor: Renato Barišić, v. pred.