Čisti kod u skriptnim jezicima

Tomiek, Tomislav

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:654958

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Tomiek

Čisti kod u skriptnim jezicima

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Tomiek

Matični broj: 45999

Studij: Informacijski sustavi

Čisti kod u skriptnim jezicima

ZAVRŠNI RAD

Mentor:

dr. sc. Novak Matija

Varaždin, rujan 2021.

Tomislav Tomiek

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Upoznavanje pravilnog programiranja baziranog na temeljnim načelima čistog koda. Obrađeni su principi čistog programiranja kao što je pravilno imenovanje raznih dijelova programa, korištenje minimalnih funkcija, metoda i klasa koje posjeduju samo jednu funkcionalnost. Za pisanje pravilnog koda značajna je i primjena prikladnih te ispravnih komentara, korištenje jediničnih testova i upravljanje iznimkama. Čisti programi često se temelje na konceptu objektno orijentiranog programiranja. Ponekad je potrebno postojeće aplikacije prilagoditi kako bi se olakšao daljnji razvoj, nadograđivanje i održavanje aplikacije. Taj proces naziva se refaktoriranje, a upravo to će biti tema praktičnog dijela rada. Kako bi se na stvarnom primjeru pokazale prednosti čistog programiranja, redizajnirana je ranije razvijena web aplikacija temeljena na skriptnom PHP jeziku koristeći MVC arhitekturu.

Ključne riječi: čisti kod, skriptni jezik PHP, web aplikacija, programiranje, MVC arhitektura, OOP, klase.

Sadržaj

1. Uvod

Mladi programeri koji još nemaju dovoljno iskustva, učenici koji žele postati programeri kada odrastu i svi ostali ljudi koji se planiraju baviti programiranjem, započeti će svoj put u programerski svijet pisanjem nekoliko ili više stotina linija koda. Njihov primarni cilj je da pritom napišu kod koji je funkcionalan, uspješno se izvršava i koji u konačnici rješava postavljeni zadatak. Međutim, često se, pogotovo u samim počecima, događa da programeri ne obraćaju pretjeranu pažnju na to kako će taj kod na kraju izgledati ili pak hoće li se zadatak obavljati optimalno. Pritom dolazi do gomilanja nepotrebnih linija ili dijelova koda, nespretnog imenovanja varijabla, funkcija i slično, a često i neprikladne strukturiranosti samog koda. Ovako napisan kod u većini slučajeva će raditi i uspješno izvršavati zadani zadatak. Međutim, kasnije se s uvođenjem promjena ili prilikom faze održavanja nailazi na probleme čiji su uzrok upravo loše postavljeni temelji programskog rješenja.

Česti uzrok nepravilnog i neurednog koda je kratak krajnji rok predaje rješenja te nedovoljno specificirani zahtjevi kupca koji su itekako podložni promjenama tijekom samog razvoja projekta. Shodno tome, programeri ne pridodaju preveliku pozornost na to kako su napisali kod, već se trude napisati ga u što kraćem vremenskom intervalu, a da ispunjava zahtjeve korisnika. Pojedini programeri imaju u mislima da će to kasnije stići izmijeniti, no često nikada ne dođe vrijeme za to. S takvim pristupom razvoju nastaje gomila nepraktičnog, nečitljivog i neurednog koda koji dapače, služi svrsi, no sa sobom nosi velike manjkavosti - pogotovo za one koji taj kod moraju kasnije čitati. Ti nedostaci izlaze na vidjelo jednom kada se odluče uvesti promjene i nove funkcionalnosti, te kada novi programeri naslijede održavanje takvog koda koji je neuredan i loše napisan.

Takva loše napisana programska rješenja učestalo dovode do velikih financijskih troškova iz razloga što se u konačnici mnogo više vremena gubi na popravljanje, nadograđivanje i izmjenjivanje trenutnog koda, nego da se inicijalno cijeli projekt postavio na dobre temelje primjenjujući principe čistog pisanja koda. Samo čitanje i pronalaženje određenog dijela koda kod takvih rješenja iziskuje izrazito puno vremena i truda. Promjena i nadogradnja pojedinog dijela koda također sa sobom nose negativne posljedice. Postoje velike šanse da će izmjene narušiti postojeće funkcionalnosti te da će izmjena na jednom dijelu koda narušiti funkcionalnost ili uzrokovati pogreške na drugim dijelovima koda. Rezultat toga je lančana reakcija koja vodi do redizajniranja cijelog programa, čime se posljedično narušava efikasnost i produktivnost rada programera te se dovodi u pitanje sama isplativost nadogradnje i održavanja takvog programskog rješenja. U tom kontekstu, ponekad je produktivnije i isplativije započeti takav projekt ispočetka.

Rješenje navedenih problema je da se programi pišu čistim kodom (eng. clean code). No, što je uopće čisti kod i kako da nečiji kod postane čist? Kako prepoznati karakteristike neurednog koda i koje sve posljedice on kao takav nosi sa sobom? Odgovori na ova pitanja bit će izneseni kroz naredna poglavlja. Kako bi se potkrijepile prednosti i detaljnije objasnile primjene čistog koda - u posljednjem poglavlju bit će razmotren praktični dio završnog rada, odnosno refaktoriranje postojeće web aplikacije korištenjem skriptnog jezika PHP bazirano na principima čistog koda.

2. Nepravilan i neuredan kod

Nepravilan i neuredan kod će poslužiti i biti od koristi u trenutnoj situaciji kada je napisan i kada rješava zadan zadatak. Usprkos početnom prividnom entuzijazmu, kasnije dolazi do velikih problema - to većih što je duži vremenski period korištenja takvih programskih rješenja. Primjerice, kada kupac/korisnik želi u budućnosti implementirati neke nove funkcionalnosti, nadograđivanje takvog loše napisanog koda, iziskivat će mnogo napora i preinaka, dok će u pojedinim situacijama biti i nemoguće.

Neki od pokazatelja lošega i neurednog koda prema Lvivity Teamu [1] su: teško snalaženje po kodu te nemogućnost pronalaženja sintakse, kod koji posjeduje nepotrebne iteracije te previše kompleksne dijelove, kao i kod čija imena funkcija, varijabli, klasa i slično nisu u skladu s njihovom namjenom. Najučestalija posljedica koju uzrokuje takav način pisanja koda je neprilagođenost na nadogradnju jer jednom kada se takav kod želi modificirati ili nadograditi, on postaje neupotrebljiv te iziskuje kaskadne promjene u ostalim dijelovima programskog rješenja.

Bitan razlog zašto se piše neuredan kod je često vrlo kratak krajnji rok isporuke programskog rješenja, kao i nedovoljna upoznatost programera sa samom domenom problema koji se rješava, a s time i nemogućnost da se predvide moguće nadogradnje rješenja u budućnosti. Loše napisani kodovi se lagano i brzo kreiraju, a da pritom ispunjavaju svoju svrhu, no pitanje je vremena kada će takva programska rješenja izazvati probleme i poteškoće u razvoju i održavanju.

Drugi razlog pisanja neurednog koda prema autoru D. Karanth [2] predstavljaju premala očekivanja od programera, osobito se ovdje misli na početnike. Smatra se da će ljudi kroz radno iskustvo, a često bez nadzora i kontrole kvalitete koda, samostalno naučiti i shvatiti sve neophodno. Međutim, to dovodi do toga da ukoliko programer nema dobru naobrazbu i ne poznaje osnove čistog pisanja koda, krenuti će lakšim putem i napisati program koji će doduše raditi, ali pritom neće voditi brigu o samoj kakvoći koda i njegovoj uporabljivosti u budućnosti.

Ponekad razlog pisanja neurednog koda mogu biti i knjige za početnike te općenito literatura o programiranju. Neprestano se u njima koriste kriva/pojednostavljena imena varijabla, klasa, funkcija, metoda i slično. Primjerice, kod iteracije nazivi varijabli su uglavnom „i“, „j“, „k“ i tako dalje, imena klasa su „primjer“ , dok se za nazive brojčanih varijabli najčešće koristi „n“. Knjige bi trebale educirati da se ipak u stvarnom programiranju koriste ispravna i smislena imena, kako bi ljudi shvatili da je imenovanje raznovrsnih dijelova programa izrazito bitno te korisno za čitljivost i urednost koda. Između ostalog, može se naići i na primjere knjiga gdje niti sam raspored koda nije pravilan, već je sve u jednoj funkciji ili u glavnoj main() metodi.

Slični primjeri bi bili i rijetko korištenje try catch blokova, već svođenje svega na grananje. Upravo zbog ovakvih primjera te neadekvatne upućenosti, čitatelji obično stječu krivi dojam te nastavljaju takvu praksu primjenjivati i u kasnijem programiranju.

2.1. Razlozi zašto pisati čisti kod

Neuredan kod izrazito je lagan za pisanje, no uglavnom dovodi do značajnih negativnih posljedica. Programska rješenja pisana neurednim kodom stvaraju velike poteškoće tvrtkama iz razloga što nadogradnja i proširivanje takvih programskih rješenja zahtjeva pregršt vremena i novaca. Kako bi se to izbjeglo, potrebno je izvršavati čišćenje koda što češće jer u suprotnom, kroz duži vremenski period, akumulirati će se veća količina neurednog koda koja će onda iziskivati puno više vremena i resursa kako bi se ona riješila. Također, redovito se događa da kada se programeru dodijeli neki postojeći neuredan kod, njegova produktivnost drastično pada. Razlog tome je što programer mora najprije pročitati i „dešifrirati“ tuđi kod i tuđe misli, što može biti vrlo zahtjevno, a u krajnjem slučaju predstavlja i veliki gubitak vremena. Neuredni kodovi mogu odužiti i vrijeme potrebno za nadogradnju i dodavanje novih funkcionalnosti ili u najgorem slučaju, onemogućiti bilo kakve promjene u programskom rješenju te se radi jednostavnosti, rađe odlučuju napustiti staru aplikaciju te kreirati novu s traženim funkcionalnostima. Isto tako, velika opasnost neurednih kodova je to što nikada ne možete predvidjeti da li će takvi programi raditi ili ne.

2.2. Čisti kod

Razni autori i programeri imaju različite definicije i mišljenja o čistom kodu, no svi će se složiti u jednom - čisti kod je esencijalan u programiranju i ključan je za razvoj kvalitetnih programskih rješenja.

Tvorac programskog jezika C++ Bjarne Stroustrup [3, str. 7] čisti kod opisuje sljedećim riječima: „Svoj kod nastojim učiniti elegantnim i učinkovitim. Logika bi trebala biti jasna te otežati skrivanje programskih pogrešaka, korištenje ovisnosti bi trebalo svesti na minimum kako bi se olakšalo održavanje, upravljanje pogreškama bi trebalo biti kompletno uz korištenje prikladne strategije, a performanse blizu optimalnih kako ne bi dovele ljude u iskušenje da kod zabrljaju s neprincipijelnim optimizacijama. Čisti kod čini jednu stvar, dobro.“.

Druga definicija čistoga koda prema autoru Gradyu Boochu govori: „Čist kod je jednostavan i izravan. Čita se kao dobro napisana proza. Čisti kod nikada ne skriva autorovu namjeru, već je pun jasnih apstrakcija i jasnih linija kontrole.“ [3, str. 8].

Sažet pogled na čisti kod pruža i autor “Big” Dave Thomas prema kojemu: „Čisti kod se može čitati i poboljšavati i od strane programera koji nije njegov izvorni autor. Sadrži jedinične testove kao i testove prihvatljivosti. Koristi smislena imena sa značajem. Pruža jedan način, a ne mnogo načina za jednu stvar. Ima minimalne ovisnosti, koje su izričito definirane, te pruža jasan i minimalan API. Kod bi trebao biti izražen prirodnim jezikom jer se ovisno o jeziku ne mogu sve potrebne informacije jasno izraziti kroz sam kod.“ [3, str. 9].

Glavne karakteristike čistog koda prema autoru G. Woodfine [4] su: programi se lagano čitaju i razumljivi su, ovo se odnosi na kompletan tijek cijeloga koda, koji mora biti razumljiv i jasan. Imena varijabli, klasa, funkcija kao i svaki izraz koji je definiran je jasan i može mu se odrediti uloga. Također, za svaki parametar koji se prosljeđuje i dobiva nazad može se točno znati što predstavlja i čemu služi. Čisti kod je raspoređen na najjednostavnije funkcionalnosti, nema kompliciranih i dugih funkcija ili main() metoda.

Kod programa koji su pisani čistim kodom, vrlo je jednostavno provesti nadogradnje i razne modifikacije po želji kupaca/klijenta. Pogreške i bugovi se također lagano pronalaze te se relativno brzo mogu ukloniti. Klase i funkcije nisu velike i kompleksne, već su raspodijeljene na manje dijelove s jasno definiranim ciljem i odgovornošću. Kod čistog koda podrazumijeva se i da su testiranja jednostavna i lagana za kreiranje i provođenje, kao i to da su testovi razumljivi i jednostavno se modificiraju.

Prema autoru R. C. Martin [3] kodovi bi se trebali nakon određenog vremena provjeriti i korigirati ukoliko je to potrebno. S vremenom, a kroz različite nadogradnje i modifikacije, moguće i od strane više osoba, akumulirati će se kod koji nije pravilno pisan. Kako bi se to izbjeglo, ponekad je dovoljno samo da se promijene pojedina imena varijabli, funkcija ili klasa u nazivu koja su smislenija za svrhu koju sada ispunjavaju. Ponekad su ipak potrebne veće preinake i reorganizacija koda. Primjerice, funkcije koje se čine predugačkima, bolje je raspodijeliti na manje sa zasebnim zadacima i ciljevima. Kod čišćenja je potrebno, također, pripaziti na pojavu dupliciranog koda, te razriješiti nepotrebne ili možda sada već njegove suvišne dijelove.

.

3. Skriptni jezici

Skriptni jezici, kao što i samo ime govori, su jezici koji podržavaju unaprijed izrađene skripte. Kako bi automatizirali pojedine postupke, ljudi kreiraju skripte koje sadrže programske naredbe. Automatizacija se postiže višestrukim korištenjem blokova naredbi, radnji, programskih datoteka te naredbenih datoteka. Skriptni jezici u naravi su drugačiji od programskih jezika, pošto programski jezici koriste skripte za automatizaciju pojedinih dijelova programa. Da se ne koriste skripte, programeri bi trebali pisati linije i linije dodatnog koda, primjerice kod učitavanja same web stranice kako bi računalo moglo razumjeti što i kako treba napraviti. Općenito, skriptni jezici se koriste kod elementarnih zadaća ili poziva aplikacijskih programskih sučelja (eng. application programming interface - API). Primjene korištenja skriptnih jezika mogu biti sljedeće:

- Aplikacijski softver
- Uređivač teksta
- Web stranica
- Ljudski operacijski sustavi
- Uređenje sustava
- Računalne igre

Skriptni jezici se obično inicijaliziraju tokom izvođenja aplikacije, a ne za vrijeme sastavljanja. Autor Z. Kalafatić [5] koristi riječ „lijepilo“ kako bi bolje opisao skriptne jezike, govori kako skriptni jezici „povezuju“ više neovisnih, nepovezanih i malih programskih datoteka u jednu cjelinu. Programske datoteke su kreirane u programima računalne arhitekture, s ciljem da računalo može razumjeti skriptne jezike. Skriptni jezik možemo protumačiti kao kod ljepila (eng. glue code). Neki od poznatijih primjera takvih skriptnih jezika su: Julia, JavaScript, Python, PHP, Rexx, Ruby, Perl i još mnogi drugi. Kalafatić [5] govori kako je prednost skriptnih jezika to što su kreirani s malim brojem linija koda, pri čemu dolazi do izražaja njihov brz razvoj kao i samih programa koji ih koriste.

Za implementaciju praktičnog dijela završnog rada odabran je skriptni jezik PHP, namijenjen prvenstveno programiranju dinamičnih web stranica te izradi poslužiteljskih aplikacija (eng. server-side).

4. Imenovanje naziva čistog koda

Pravilno i smisleno definirana imena funkcija, parametra, argumenata, klasa, varijabla, lista, konstanti i ostalih dijelova koda čine temelj pisanja dobrog i kvalitetnog čistog koda. Naime, svako ime koje se koristi u kodu trebalo bi pružiti dovoljno informacija o njegovoj namjeni te predočiti krutu ideju o čemu se radi i za što se koristi prema autoru F. Miha [6]. Primjerice, varijabla godina ne daje nikakve informacije o značenju, pa tako može služiti za definiranje godine proizvodnje, trenutne godine ili nešto sasvim drugo. Prikladnije ime za navedenu varijablu bilo bi $godina_proizvodnje_proizvoda, $godina_rodenja ili slično. Preporuka je i da se ne koriste nazivi kojima samo mali krug ljudi poznaje značenje, već je bolje koristiti opće poznate riječi. Ime varijable s nekom nadopunom ili malim opisom govori o svemu što se treba znati o toj varijabli bez ikakve potrebe za dodatnim komentarima. Promjena naziva u prihvatljivije i razumljivije, osjetno povećava kvalitetu koda. Također, trebalo bi izbjeći korištenje hard kodiranih vrijednosti, a umjesto toga koristiti varijable s pripadajućim imenom. Pojedini dijelovi koda će autoru koda možda biti razumljivi i logični, no za druge programere neće imati nikakvog smisla. Primjerice, imamo varijablu "d" koja predstavlja dan koji je odabran, i sljedeće grananje if ($d <= 5) { dio koda }. Ovakvo grananje i hard kodirana vrijednost 5 nam ne govori gotovo ništa. Kod navedenog primjera, prikladnije bi bilo da se promijeni varijabla "d" u $trenutni_dan, a vrijednost 5 da se pridruži novo kreiranoj varijabli $dani_tjedna. Zadano grananje bi zatim izgledalo ovako: if ($trenutni_dan <= $dani_tjedna), čime postaje mnogo jasnije i adekvatnije.

Kod odabira imena trebali bismo pripaziti da ne odaberemo skraćenice ili sinonime određenih dijelova programskog jezika budući da odabirom sinonima možemo dovesti nekoga u zabludu o značenju tog dijela koda. Primjerice, ukoliko imenujemo varijablu za pohranu internacionalne liste građana s $int_lista_gradana, čime smo skratili riječ internacionalna u int te dodali $lista_građana na kraj. Ovaj primjer je podosta zbunjujući, pošto int predstavlja integer brojeva, a ne riječ internacionalan, a lista predstavlja polje (array) u programiranju, dok u zadanom izrazu može biti zadan samo jedan string ili broj. Spomenuto se može protumačiti kao polje brojeva građana, ne znajući što je autor time htio reći.

Neki programeri preferiraju korištenje različitih tipova fontova iz razloga da se bolje i bržim tempom uoče razlike između sličnih riječi, brojeva, velikih i malih slova kod pojedinih naziva imena. Primjerice veliko slovo "O" ima veliku sličnost sa brojem nula, a malo slovo "L" ima veliku sličnost sa brojem jedan (0 O i 1 l).

Nipošto nije poželjno koristiti isti naziv za dvije ili više stvari, s različitim sufiksom ili prefiksom. Primjerice, ako se nazivu doda samo neki broj ili drugi znak, dok je korijen riječi

jednak kao i kod drugih varijabla, time ne dobivamo dodatne informacije o značenju. Istoimeni ili slični nazivi nam ne otkrivaju značenje pojedinih varijabli, nego nas dodatno zbunjuju. Imate li na primjer dva različita datuma, nikako nije dobro koristiti imena varijabli datuma: datum1 i datum2, već je poželjnije navesti neki dodatni opis uz navedeni datum. $pocetni_datum_proizvodnje_serije_005 i $zavrsni_datum_proizvodne_serije_005 su razumljiva imena za varijable, gdje se točno zna na što se misli. Nema smisla niti koristiti naziv sa sinonimima za različite stvari ili dodati sinonim kao nadopunu kod sličnih pojmova. Primjerice, koristimo li sljedeća imena kod klase korisnika: ''dohvati_korisnika'', ''dohvati_ korisnika_podaci'' i ''dohvati_korisnika_informacije''. Navedeni nazivi klasa upućuju da su sve klase iste i ne može se odrediti čemu pojedina klasa služi sve dok se detaljnije ne pogleda njezin kod.

Prema autoru S. Sultan [7] odabir naziva poželjno je koristiti one nazive koji se lako i jednostavno izgovaraju te su općepoznati. Primjerice, riječ otorinolaringologija je teška za izgovor i pisanje, ne bi li bilo jednostavnije i razumljivije koristiti naziv $doktor_uha_grla_nosa. Značenje je jednako i zna se na što se misli kada se pročita naziv. Problem može predstavljati i to što se kod teško izgovorljivih riječi često potkradu pogreške u pisanju što pak kasnije može dovesti do problema. Primjerice, jednom sam krivo napisao naziv mape koja je imala teško izgovorivu riječ u sebi i nikako nisam mogao pronaći zbog čega kod ne radi. Zbog tako male greške, uzaludno gubimo naše dragocjeno vrijeme.

U pravilno pisanom kodu koriste se nazivi varijabla koji se lako mogu pronaći i pretraživati pomoću tražilice. Korištenjem brojeva kao hard kodiranih varijabli, umjesto da se kreira nova varijabla ili konstanta, onemogućujemo pretraživanje pojedinih dijelova programa. Osim korištenja brojeva za nazive, prema autoru R. C. Martin [3] pogrešno je i koristiti nazive sa samo jednim znakom ili slovom. Prema njemu, jedino prihvatljivo rješenje za njihovo korištenje bi bilo kod lokanih varijabli unutar kratkih metoda ili petlji. Kod svih ostalih primjena takvih varijabli - kod postaje nejasan i nemoguć za pretraživanje. Također, varijable gdje se koriste specifične brojčane svote, trebalo bi zamijeniti s novokreiranim varijablama s pridruženim prikladnim nazivom, čime se postiže jasan i čitljiv kod koji je ujedno i lagan za pretraživanje.

Izbjegavanje ''kodiranih'' imena je dobar početak kod pisanja čistog koda, no to ponekad i nije moguće, navodi R.C. Martin [3]. Kada želimo ''sakriti'' sučelje, kako korisnici ne bi znali za njega, tada je poželjno kodirati sam naziv implementacije. Dekodiranje imena naziva je stresno, iz razloga što programeri ne samo da moraju znati napisati pravilan kod već se moraju učiti i dekodiranju i kodiranju, koje opet može ovisiti od firme do firme.

Dobar savjet kod odabira imena je izbjegavanje mentalnog mapiranja, ističe S. R. Dipta [8]. Ljudi koji će čitati kod, ne bi trebali u glavi pretpostavljati imena varijabli, iz razloga što je trenutno ime siromašno opisom te mu se ne zna značenje. Primjer koji je uvijek prisutan u većini koda su varijable i, j, k, n, m kod raznih petlji. Primjer za petlje nalazi se u skoro svim knjigama za programiranje,iako nije apsolutno ništa krivo sa ovim primjerom, jednostavno je tako.

4.1. Nazivi metoda i klasa

Za nazive klase i objekata uvijek je potrebno koristiti imenice, a ne glagole. Osim imenice može se koristiti i imenska rečenica. Primjerice, dobar naziv za klasu bio bi korisnik ili račun, no možete stavljati i neke od imenskih prefiksa prije stvarnih imena poput: koordinator, čitač, pisač, spremnik, tvornica ili slično.

Kod naziva metoda, koristi se suprotna praksa, a za njihovo imenovanje trebalo bi upotrebljavati glagole ili glagolske fraze, napominje S. R. Dipta [8]. Prema standardima, morali bi dodati prefikse get, set i is na predikate. Nazivi funkcija trebali bi biti isti kao i kod metoda, pošto imaju slične funkcionalnosti - samo što metoda koristi instancu ili objekt.

Poželjno je i da se ne koriste različite riječi sa sličnim imenicama kod definiranja metoda ili funkcija s istom funkcionalnošću. Također isto vrijedi i u suprotnom smjeru, za različite pojmove nikako nije dobro koristiti iste riječi.

Uzimajući u obzir da će i ostali čitatelji vašeg koda također biti programeri, kod imenovanja mogu se koristiti i specifična tehnološka imena (IT, strojno učenje) za imena varijabli, metoda, klasa i slično. Prihvatljivo je i kao ime odabrati glavni problem ili cilj koji se funkcijom ili metodom postiže.

Korištenje prefiksa ili sufiksa na postojeća imena nije zabranjeno ni pogrešno, no pripazite da tu varijantu odaberete kao posljednju za željeni naziv. Korištenjem istog prefiksa kod imenovanja varijabli stvaramo grupe koda koje spadaju pod sličnu ili istu strukturu. Bolja varijacija kod sličnih ili povezanih varijabli je kreiranje vlastite klase ili funkcije sa željenim varijablama, kako bi se spriječilo gomilanje varijabli sa jednakim prefiksom.

Kreiranje kraćih naziva za imena je mnogo bolja opcija od dugačkih imena, no, kratka imena moraju biti jasna. Promjena te improvizacija raznih problematičnih imena je itekako potrebna, no neće svi programeri to shvatiti ili željeti primijeniti iz raznih razloga. Prvenstveno, riječ je o strahu od promjene. Dok neki programeri smatraju da su im nazivi imena prihvatljivi i razumljivi, drugi uoče loše napisani kod ali ne žele ništa mijenjati, jer misle da će ih drugi programeri kritizirati nakon promjene programa.

5. Funkcije i klase

Pojam s kojim se programeri svakodnevno susreću su funkcije. One predstavljaju izdvojeni dio koda koji se, dok se jednom napiše, može po želji koristiti u bilo kojem dijelu koda. Naravno, pod uvjetom da je željena funkcija u kodu programa ili je uključena u kod. Funkcija posjeduje svoj tip, ime i argumente. Prema autorima M. Koneckome i A. Lovrenčiću [9] funkcija se sastoji od četiri dijela: Povratni tip podataka, ime funkcije, argumenti ili parametri i tijelo funkcije. Povratni tip podataka je tip podataka koji će biti vraćen nakon završetka rada funkcije, ime funkcije je identifikator za pozivanje funkcije. Dok su argumenti funkcije zapravo ulazne varijable funkcije koje se definiraju nakon imena funkcije u okruglim zagradama. Tijelo funkcije je predviđeno za dio koda koji će se izvršavati.

Funkcije čistoga koda bi trebale biti što manje, jer manje koda doprinosi tome da je kod čitljiv i da se iz njega lakše može shvatiti sam cilj funkcije. Pravo pitanje je koliko bi funkcija trebala biti mala. Davnih vremena funkcija ne bi smjela biti veća od veličine ekrana, no ta tvrdnja više nije točna, pošto su današnji ekrani poprilično veliki. Savršena funkcija bi trebala imati samo par linija koda (otprilike od tri do pet linija koda). Prilikom korištenja nekih blokova naredbi, (if, else, for, switch, while) kod bi se trebao razdvojiti u zasebne funkcije gdje će se te naredbe izvršiti. Pozivanje bloka naredbi grananja, iteracije te njihovih uvjeta bi trebalo uvijek biti zapisano u jednoj liniji koda, a njihov blok naredbi bi se trebao nalaziti unutar vitičastih zagrada uvučen za jedan tab udesno. Korištenjem novih funkcija kod iteracije ili grananja, smanjuje se prenatrpanost koda, a ujedno se dobivaju detaljnija imena za svaki blok naredbi koja opisuju što točno taj dio koda radi te čemu služi.

Pravilno bi bilo da svaka funkcija posjeduje samo jedan cilj ili zadatak. Ponekad funkcije ne mogu imati samo jedan cilj, no, to nije toliki problem. Primjerice, grananje switch je u velikim slučajevima prilično nezgodno za koristiti, pošto switch ima N grananja, a s time i N ciljeva. Kod switcha se stoga mogu koristiti manje funkcije ili metode koje ne smiju posjedovati duplicirani kod. Funkcije mogu posjedovati različite pozive drugih funkcija ili slično, sve dok su pozvane funkcije jednu razinu niže od glavne funkcije.

Primjer:

```
const $_minimalna_kolicina_namirnica = 100;

function provjeri_skladiste_namirnica ($id_namirnice) {
  $podaci_namirnice = $this->namirnice_repo->dohvati_prema_id
    ($id_namirnice);
  if ($this->_provjeri_kolicinu_namirnica($id_namirnice)){
```

```
      $this->naruci_namirnicu ($namirnica);
    }
      $this->izvjestaj_repo->kreiraj_izvještaj_namirnice($namirnica);
      $this->preusmjeri("namirnice/prikaz_popis_namirnica/");
}

private function _provjeri_kolicinu_namirnice($namirnica){
  return ($podaci_namirnica->kolicina_namirnice =< this->$_minimalna_
      količina_namirnica);
}
```

Funkcije koje posjeduju više razina apstrakcije (posjeduju više nepovezanih zadataka/ciljeva) zbunjujuće su tijekom čitanja. Pravilno bi bilo da funkcije posjeduju samo jednu razinu apstrakcije. Kod pisanja funkcija, one bi trebale biti pisane po redu jedna iza druge, budući da je tako preglednije, a ujedno je i logičnije za čitati – za razliku od situacije gdje bi one bile navedene „gore pa dolje“. Funkcije niže razine se smještaju nakon funkcija viših razina.

Imena funkcija također bi trebala biti logična i otkrivati što pojedina funkcija radi. Odabir imena funkcija je bitan, stoga je poželjno odvojiti određeno vrijeme za njihovo imenovanje. Neke od dobrih praksa kod imenovanja funkcija su: korištenje istih fraza kada je moguće, odabir smislenih imena, korištenje ako i treba dužih imena kako bi se pojasnio cilj funkcije, povezivanje imena funkcija nižih razina sa onima više razine koje te funkcije koriste čime se doprinosi većem smislu samog programa.

5.1. Argumenti funkcije

Ulazni argumenti funkcije su argumenti koji se prosljeđuju u zadanu funkciju te tamo koriste. Za njihovu idealnu količinu preporuča se da je što manja, točnije nula po autoru R. C. Martinu [3].

Maksimalna količina argumenata bi trebala biti ograničena na tri pod određenim uvjetima. Najveći razlog tome je to što se kod testiranja moraju uzeti u obzir sve kombinacije različitih ulaza, što dodatno komplicira razvoj testova utoliko više što je broj argumenata veći. S druge strane, funkcije sa nijednim ili jednim argumentom se mogu lagano i jednostavno testirati. Kod većih funkcija koje zahtijevaju veći broj argumenata, poželjno je takve parametre slati kao polje ili idealno kroz vlastite klase koje će potrebne informacije dohvaćati preko objekta.

Funkcije osim ulaznih varijabli posjeduju i izlazne argumente, pa tako one prilikom završetka mogu vraćati određenu varijablu ili ponekad, ukoliko to slučaj zahtjeva, ne moraju

vraćati ništa. Kod čistog programiranja praksa je da funkcije ili vraćaju neke informacije ili vrše neku obradu i izvršavaju linije koda, no nikada se ne bi smjelo raditi oboje u jednoj funkciji.

Izrazito je poželjno u funkcijama koristiti rukovoditelj pogrešaka (eng. try catch(Exeptions)) kroz koji se definira kod koji će se izvršiti ukoliko se dogodi neki problem ili se naiđe na pogrešku u radu. Try/catch blokovi ne samo da olakšavaju pronalaženje pogreški ili bugova u kodu, nego ujedno i zabranjuju prosljeđivanje pogreški. Tijelo try/catch blokova je poželjno izdvojiti u posebne funkcije, radi preglednosti i razumljivosti. Primjerice, try/catch blokovi mogu u nekim slučajevi biti poveći, no to je u redu, pošto oni posjeduju jedan cilj, a to je niz naredbi koje će se izvršiti kada dođe do određenih problema u programu.

Autor E. W. Dijkstra [10] navodi kako svaka funkcija i svaki blok funkcije treba posjedovati samo jedan ulaz i jedan izlaz. Prema njegovim navodima, funkcija bi na kraju trebala imati jednu povratnu (eng. return) naredbu koja vraća argumente. Precizira kako funkcije ne bi smjele posjedovati naredbu za prekid (eng. break) ili naredbu za nastavljanje (eng. continue) kod blokova iteracije, kao i to da funkcija nikako ne bi smjela posjedovati naredbe za preskakanja (eng. goto).

5.2. Klase

Klase su temelj kreiranja objektno orijentiranih programa. Klase opisuju stvarne pojmove, stvari i situacije iz realnog svijeta. Varijable kojima dodijelimo detaljno razrađene klase postaju objekti, a nazivamo ih i instancama klasa. Klase posjeduju varijable koje se nazivaju svojstvima klase, a funkcije koje su definirane u klasama se nazivaju metodama. Vidljivost svojstva i metoda klasa može biti definirana kao privatna (eng. private), javna (eng. public) i zaštićena (eng. protected). Svojstva i metode koje su označene kao privatne se mogu koristiti samo kod zadane klase, dok im druge klase ili programi ne mogu pristupiti. Javne metode i svojstva su dostupne svima, a svojstvima i metodama koja su protected može se pristupiti samo iz iste ili naslijeđene klase. Klase posjeduju metodu koja se naziva konstruktor te se izvodi na samom početku tijekom kreiranja njezine instance. Konstruktori služe za inicijalizaciju svojstva te ne vraćaju nikakvu vrijednost. Konstruktori postavljaju svojstva na neku početnu vrijednost prema autoru P. Brođanacu [11]. Autor govori kako klase mogu slobodno posjedovati više istoimenih metoda, no svaka metoda onda mora posjedovati različiti broj parametra ili različiti tip parametra. Tako definirane metode se tada nazivaju preopterećene (eng. overloadane) metode. Nakon što smo definirali sva svojstva i metode klase, možemo konačno inicijalizirati objekt. Objektu pridružujemo ime sa naredbom new čime i inicijaliziramo objekt željene klase ($naziv_objekta = new naziv_klase($parametri);).

5.2.1. Organizacija klasa

Klase čistoga koda, kao i svaki drugi pojedini dio čistoga koda, morala bi biti što manja. Njezin naziv trebao bi otkrivati odgovornost i zadaću klase. Autor R. C. Martin [3] iskazuje mišljenje da ako ne možemo izmisliti sažeto ime za klasu, onda je klasa prevelika i označava preveliku odgovornost. Klase bi trebale sadržavati jedinstvenu odgovornost, a nipošto ne više različitih odgovornosti. Klase se trebaju kreirati i organizirati po zajedničkim funkcionalnostima, tako da se za sve metode može logički zaključiti gdje se nalaze. Velike metode kod klasa se mogu rastaviti na manje metode i time se dobije bolja preglednost. Kod rastavljanja metoda ujedno možemo i rastaviti klasu na manje, prihvatljivije klase, koje će posjedovati svoj vlastiti cilj. Primjerice, kod izrade aplikacije korišteno je više klasa repozitorija koje služe za dohvaćanje i ažuriranje vrijednosti iz baze. Svaka pojedina klasa repozitorija posjeduje istoimene ili slične metode (posjeduje zajednički cilj, primjerice zapis podataka u bazu, dohvaćanje podataka i slično, no svaka metoda se različito izvodi) i umjesto da je korištena jedna velika, nepregledna i nerazumljiva klasa, odabrano je kreiranje desetak manjih nezavisnih klasa sa zajedničkim sučeljem repozitorija. Novo kreirane klase koriste koncepte objektno orijentiranog programiranja kako se ne bi duplicirao kod. Sljedeći primjer pokazuje jednu od novokreiranih zasebnih klasa koje su kreirane na temelju komunikacije prema bazi podataka.

```
class Glasovanje_db_repozitorij implements
Glasovanje_repozitorij_interface {
  private $_db;
  private $_dnevnik;
  private $_koriscnisko_ime;

  public function __construct() {…}
  public function kreiraj($uneseni_podaci) {…}
  private function _kreiraj_model_glasovanje($uneseni_podaci) {…}
  public function dohvati_listu() {…}
  public function obrisi($id_zapisa) {…}
  public function azuriraj($uneseni_podaci) {…}
  public function dohvati_glasovanje_prema_izlozbi($id_izlozbe) {…}
}
```

6. Komentari

Komentari u programiranju su poželjni, no pod imperativom da su dobro napisani. Komentare najčešće koristimo kada ne možemo odrediti dobra i smislena imena za nazive koda ili su ta imena previše skromna i ne opisuju dobro dodijeljeni zadatak. Ponekad programeri napišu zbunjujuće i preopširne dijelove programa, pa smatraju da je jedino dobro rješenje dodati komentar. Komentari pobliže objasne kodove, no to dovodi do loše napisanih kodova, jer u pravilu čisti kod ne bi trebao uopće sadržavati komentare.

Komentari nam pomažu razjasniti dijelove kodova te daju opise zadanih kodova. Compiler ne gleda komentare kao dio koda, nego ih ignorira. Komentari kod većine programskih jezika započinju navođenjem dvostruke kose crte (//) kod jedno linijskih komentara, a kod više linijskih kodova na početku komentara se stavlja kosa crta i zvjezdica, a na kraju komentara se doda zvjezdica pa kosa crta (/* komentar */). Drugi programski jezici možda koriste drugačiji zapis komentara, primjerice kod HTML-a se koriste znakovi veće i manje s kombinacijom uskličnika i crticom (<!—komentar -->). Drugi najčešći znakovi za početak komentara su #, - -, =, % i slično. Jednostruki linijski komentari mogu se dodati bilo gdje u kodu. Oni mogu opisivati neku varijablu, tako da se doda komentar nakon varijable u istoj liniji koda i slično.

Stari inicijalno napisani komentari tokom vremena postaju loši ukoliko se ne ažuriraju, pošto se kodovi svakodnevno unaprjeđuju i popravljaju te značenje komentara sve više i više odstupa od pravog cilja. Programeri čistih kodova bi trebali u pravilu posvećivati više vremena osmišljavanju dobrih imena, strukturi, definiranju razumljivih, čitljivih i jednostavnih funkcija i klasa, te tako smanjiti potrebu za komentarima kao rješenjem kojim objašnjavaju svrhu programa.

6.1. Dobri komentari

Dobri komentari su rijetki, ali korisni. Najčešći dobar primjer korištenja komentara su izjave o autorskim pravima i autorstva koja se navode na početku programa. Komentari nam doprinose korisnim informacijama o kodovima, ali jedino ako se redovito održavaju (pogotovo kod modifikacije kodova). Primjerice, ako se želi promijeniti ili koristiti regular expression usporedba, onda je dobro koristiti komentare koji će taj izraz detaljno razjasniti. Korištenje komentara kao "slanje poruka" ili ostavljanje savjeta i informacija je sasvim u redu. Ukoliko pojedini dijelovi kodova nisu bili testirani ili nisu predviđeni za rad s više dretva, poželjno je da

se ostavi komentar koji će to razjasniti s ciljem da se uštedi vrijeme i da se spriječe potencijalni budući problemi u programu.

Komentari nam mogu poslužiti i kao popis zadataka koje moramo obaviti i napisati u skoroj budućnosti, pa se tako budući zadaci upišu na predviđeno mjesto u kodu kroz komentare. Komentare možemo i koristiti kada želimo nešto kod koda posebno istaknuti te time obratiti pažnju drugih programera koji će taj kod u budućnosti čitati.

Primjer dobrog komentara:

```
function provjeri_uneseni_datum(){
  $uneseni_podaci = $this->provjeri_unesene_podatke();
  // Varijabla format datuma predstavlja tri puna ili skraćena formata
  // datuma ( mm/dd/yyyyy ili mm-dd-yyyy ili mm.dd.yyyy)
  $format_datuma = "/^(?:(?:(?:0?[13578]|1[02])(\/|-|\.)31)\1|"
    . "(?:(?:0?[1,3-9]|1[0-2])(\/|-|\.)(?:29|30)\2))(?:(?:1[6-9]|"
    . "[2-9]\d)?\d{2})$|^(?:0?2(\/|-|\.)29\3(?:(?:(?:1[6-9]|[2-9]\d)"
    . "?(?:0[48]|[2468][048]|[13579][26])|(?:(?:16|[2468][048]|"
    . "[3579][26])00))))$|^(?:(?:0?[1-9])|(?:1[0-2]))(\/|-|\.)"
    . "(?:0?[1-9]|1\d|2[0-8])\4(?:(?:1[6-9]|[2-9]\d)?\d{2})$"
  if (preg_match($format_datuma, $uneseni_podaci->datum_rodenja)){
    return $uneseni_podaci;
  } else {
    return FALSE;
  }
}
```

6.2. Loši komentari

Loše napisani komentari mogu u konačnici napraviti kod lošijim, nego da je on uopće bez komentara. Loši komentari lako nas mogu uputiti u "slijepu ulicu", primjerice, mislimo da kod radi jednu stvar, a zapravo radi pola te stvari ili nešto sasvim treće. Takvo djelovanje dovodi do raznih pogrešaka u kodu i gubljenju vremena, jer ne samo da će programeri prvenstveno trebati tražiti gdje je pogreška nastala, već će trebati shvatiti kako taj kod kojeg opisuje komentar, zapravo radi.

Loši komentari nastaju zbog brzopletosti, te kada se ne uloži dovoljno vremena kod samog pisanja komentara. Loši komentari se mogu prepoznati na način da se iz njih ne mogu izvući sve potrebne informacije, već je dodatno potrebno proučavati i sam kod kako bi se saznalo što on uistinu radi. Također, nepotrebno je dodavati i "beskorisne" komentare koji ne otkrivaju ništa posebno ili više od samoga koda.

Komentari koji su inicijalno nastali i koji opisuju svaku modifikaciju na kodu su loši komentari. Takvi komentari nas ne zanimaju pošto su dugi i opisuju nepostojeći i prethodni kod. Nekvalitetni komentari su komentari koji nam ne daju nove informacije, nego ponavljaju istu informaciju koja se može iščitati iz naziva koda. Takve komentare bi trebalo izbjegavati. Poželjno je odabrati smislena imena za funkcije ili varijable koja će dati dovoljno informacija, a ne koristiti neke dodatne komentare koji će se samo ponavljati. Ponekad programeri komentiraju završetak grananja i petlji, što je korisno kod dugačkih petlji i grananja, jer se time može lagano uočiti gdje je njihov kraj. Međutim, ukoliko se već trebaju koristiti komentari za označavanje kraja koda, bolja solucija bi bila kada bi se takvi dijelovi kodova razdvojili na manje funkcije i cjeline.

Za sve zapisane komentare, bili oni loši ili dobri, ljudi će smatrati da su oni korisni i da doprinose razumljivosti samog koda. Takvo mišljenje dovodi do gomilanja komentara i prljanja kodova. Suvišne komentare trebalo bi izbrisati. Kod svake modifikacije koda koja se vrši, trebalo bi također provjeriti i komentare, te ako je potrebno - obrisati ih ili ažurirati. Nepoželjno je da se programi prenatrpaju s velikim komentarima koji sadrže nepotrebno detaljne informacije. Komentari se najčešće koriste na početku neke funkcije ili klase te opisuju što će taj dio koda raditi, no ako funkcija posjeduje samo jedan cilj, nije potrebno koristiti komentare, nego je sasvim dovoljno smisliti dobro ime za tu funkciju.

Primjer lošeg komentara:

```
$kljuc_ormarica = array [ '0' => 'Viktor' ]        // dodajemo Viktorov
  //ključ  ormarića u listu pod rednim brojem nula
```

Navedeni primjer za korištenje komentara je suvišan, pošto se sve potrebne informacije mogu pročitati iz koda. Korištenje neprihvatljivih komentara, primjerice korištenje ružnih riječi ili iskazivanje humora je definitivno loš primjer komentiranja koda, pogotovo kod velikih informatičkih tvrtki ili poduzeća, jer se time (ne) pokazuje koliko su profesionalni programeri, a ujedno i njihovu zainteresiranost za projekte.

7. Tehnike formatiranja koda

Tehnika formiranja ili oblikovanja koda podrazumijeva način pisanja programa. Ona definira kojim redoslijedom i kojim stilom se kod piše. Kod razvoja većih projekata gdje sudjeluje više programera, potrebno je najprije razjasniti kako će se programirati i formatirati kodovi, a osim toga, potrebno je i voditi dokumentaciju o kodu. Prema autoru N. Schäferhoff, [12] oblikovanje kodova se bazira na korištenju dogovorenih povlaka koda, bijelih prostora (praznih linija kodova), definiranju standarda imenovanja imena (mala i velika slova), stilu i pravopisu funkcija i varijabli, korištenju komentara i stilu komentiranja.

Što se tiče vertikalne i horizontalne veličine, odnosno broja linija programa ili broja znakova po liniji koda, ne postoji točno zadano pravilo o veličini. Razni programski jezici ili programski okviri koriste programe koji su malih vertikalnih opsega, prosječno im programi posjeduju oko 200 linija kodova, a maksimalna gornja granica programa im doseže do 500 linija koda.

7.1. Vertikalno formatiranje

Pojam vertikalno formatiranje kodova odnosi se na veličinu kodova, točnije koliko pojedini programi posjeduju linija koda. Vertikalno formatiranje kodova čita se s lijeva na desno, te odozgo prema dolje, po redu od složenih funkcija do najjednostavnijih funkcija. Time se kreira dobar protok i redoslijed kodova. Koriste se prazne linije za razdvajanje raznih nepovezanih i povezanih dijelova kodova s ciljem da se poveća čitljivost. Kod vertikalnog oblikovanja nazivi programa trebali bi biti jednostavni, početak kodova bi trebao sadržavati neke bitne informacije ili opis programa (ne previše informacija i detalja), što će raditi i kako će raditi. Kako sve dublje i dublje ulazimo u kod, linije kodova bi trebale sadržavati sve više i više informacija i detalja. Samo dno programa sadrži krajnje funkcije koje se posljednje izvode i koje se više ne razdvajaju te ne vode više nikamo dalje u dubinu.

Povezani dijelovi programa su po vertikalnoj udaljenosti što bliže, to znači funkcije koje su jako povezane su jedna ispod druge, dok slabije povezani dijelovi (funkcije koje nisu toliko ovisne jedna o drugoj) se mogu razdvojiti i mogu biti više udaljeniji jedan od drugog. Isto pravilo vrijedi i za pozivanje raznih datoteka, povezane datoteke se nalaze u jednoj mapi. Funkcije koje pozivaju druge razne funkcije bi također trebale biti jedna do druge, kako bi se drugi programeri što lakše snalazili po kodu. Pozvana funkcija se uvijek nalazi ispod funkcije koja ju je pozvala.

Inicijalizacija varijabli po pravilima bi se uvijek trebala nalaziti na početku pojedinih klasa, iz razloga što bi svaki programer trebao odmah znati gdje se je varijabla inicijalizirala i gdje treba dodati ili modificirati varijable.

7.2. Horizontalno formatiranje

Horizontalno formatiranje odnosi se na dužinu pojedinih linija programa. Prosječan broj znakova po liniji koda iznosi oko 45 znakova. Neke linije su kraće od samo 10 znakova, neke linije su duže pa sadrže do 100 ili do 120 znakova, no neka gornja granica bi trebala biti do 120 znakova po jednoj liniji. Kod horizontalnog formatiranja također se koriste praznine - (eng. tabovi) poput lijeve praznine s kojom grupiramo slične i povezane dijelove kodova. Novu funkciju odvajamo s jednom praznom linijom, a otvaranje blokova petlji, grananja i ostalih dijelova kodova ne odvajamo s prazninama ili s novim redovima, ali zato zatvaranje odvajamo kako bi stekli bolju preglednost, čitljivost i povezanost kodova. Kod kreiranja varijabli ne bi smjeli koristiti tab između tipa varijable i naziva varijable, pošto time gubimo na jasnoći koda, a pojedinci bi mogli ignorirati tip varijable i samo se fokusirati na naziv varijable.

Spomenuto možemo predočiti/demonstrirati na sljedeća dva primjera, prvi predstavlja loše horizontalno formatiranje, a drugi dobro.

Krivo formatirani kod:

```
function dohvati_podatke_korisnika ( $idkorisnika ) {
  $korisnicko_ime          =  $_POST['korisnicko_ime'];
  $lozina                  = $_POST['lozinka'];   }
```

Prihvatljivo formatiranje koda:

```
function dohvati_podatke_korisnika($id_korisnika) {
     $korisničko_ime = filter_input(INPUT_POST, 'korisnicko_ime');
     $lozina = filter_input(INPUT_POST, 'lozinka');
}
```

Korištenje uvlaka u programiranju je neizbježno budući da pomoću njih prepoznajemo razne dijelova kodova izrazito lagano i uočljivo. Uvlake koristimo kako bi dobili hijerarhiju koda. Uvlačimo unutarnje elemente funkcija, klasa, petlji, grananja jednu razinu udesno (jedan tab udesno), kako bi stvorili njihovu grupu. Kod korištenja novih povezanih blokova, ponovno uvlačimo kod za jednu razinu udesno i tako dalje. Kada su pojedini blokovi završeni i zatvoreni, onda se ponovno vraćamo na prijašnju razinu. Kada se ne bi koristile uvlake, kodovi bi bili nepregledni i zbunjujući.

8. Objektno orijentirano programiranje

Objektno orijentirani (eng. object oriented - OO) su programi temeljeni na klasama koje se realiziraju preko objekata. Globalne varijable "ne postoje", već kod OO programiranja svaki objekt koristi svoju varijablu. Isto tako vrijedi i za funkcije, funkcije pripadaju klasi. Navedene varijable u OO programiranju nazivamo podatkovnim članovima klase ili svojstvima, dok su funkcije koje se izvršavaju imenovane kao funkcijski članovi klase ili kraće metode. Prema autoru S. Srinidhi [13] OO programiranje zasniva se na četiri principa:

- Nasljeđivanje
- Učahurivanje ili enkapsulacija objekta
- Apstrakcija
- Polimorfizam ili višeobličje

8.1. Nasljeđivanje

Nasljeđivanje je pojam koji možda već u samome nazivu govori o tome što radi. Veliki projekti koriste pregršt različitih klasa, ali i veliki broj sličnih i identičnih klasa te klasa koje međusobno dijele neke od svojih aspekata. Takve klase je najbolje spojiti u zajedničku klasu koja posjeduje sva zajednička svojstva i metode. Potom preostale klase nasljeđuju tu klasu, dijele njezina svojstva i metode, dok razlike proširuju sa samo sebi svojstvenim svojstvima i metodama. Objektno orijentirani programi takve zajedničke klase nazivaju baznom klasom ili roditeljskom klasom, a klase koje nasljeđuju svojstva roditeljske klase su subklase ili klase djeteta.

OO programi čistoga koda bi u svakom slučaju trebali posjedovati nasljeđivanje klasa, kako bi se što više izbjeglo ponavljanje koda. Sva dodatna svojstva koja su potrebna, mogu se zatim dodati s drugom klasom koja će proširiti željenu postojeću klasu. Drugu klasu koja će proširiti pravu klasu kreiramo tako da ju samo proširimo sa naredbom class Subklasa_klasa extends Roditeljska_klasa {}.

Autor P. Brođanac [11] navodi jednostavni primjer nasljeđivanja koji se sastoji od DVD-a koji sadrže glazbu i filmove. Kreirao je jednu zajedničku klasu, kako ne bi trebao višestruko pisati ista svojstva i iste metode za različite klase. Klasu DVD definiraju svojstva: naslov, veličina, komentar te metode dodaj_naslov, dodaj_komentar, dohvati_naslov, dohvati_komentar i dohvati_veličinu. Naslijeđene klase glazba i film će imati ista svojstva i iste metode kao i klasa DVD, no klasu glazbu i film ćemo proširiti sa dodatnim svojstvima i metodama. Svojstva klase film se proširuju sa trajanje i režiser, a metode s dodaj_trajanje,

dohvati_trajanje, dodaj_režisera, dohvati_režisera. Dok će klasa glazba koristiti malo drugačije dodatne metode i svojstva, koja se onda lako po potrebi mogu dodati.

8.2. Enkapsulacija

Enkapsulacija je poznata i pod pojmom učahurivanje objekta, a temelji se na načelu ''sakrivanja'' podataka. Autor D. Phillips [14] kod enkapsulacije koristi frazu ''zakopavanje vremenske kapsule''. Objašnjava je kao zakopavanje gomile podataka u vremensku kapsulu, s ciljem sakrivanja podataka. Metaforički govori i da ukoliko se vremenska kapsula ne zakopa, ili ukoliko se napravi od prozirnog materijala, da će podaci i dalje biti nedostupni, no ne i skriveni. Enkapsulacijom postižemo sigurnost podataka kod višestrukog korištenja iste klase. Podaci i njihovi tipovi ostaju isti i nepromijenjeni tokom korištenja klase, pošto su privatnog oblika, a metode koje se koriste su javne i dostupne svima.

Enkapsulacija je potrebna kod čistoga koda pošto sakriva nepotrebne značajke, a prikazuje samo bitne. S njome se postiže održivost, fleksibilnost i proširivost aplikacije. Čisti kod koristi enkapsulaciju kako bi zaštitio varijable, a to postiže tako da se varijable inicijaliziraju najčešće s private metodom. Kreiraju se metode pristupa koje zamjenjuju izravan pristup samoj varijabli čime se onemogućava njezina promjena.

8.3. Apstrakcija

Postoje raznolike klase koje dijele iste funkcionalnosti, no njihova implementacija je različita. Korištenjem apstrakcije rješava se navedeni problem. Apstrakcija koristi zajedničke metode prikaza koje će definirati vanjsko sučelje, odnosno opise metoda koje njezine implementacije trebaju podržavati i implementirati svaka na svoj način. Kako se ne bi ponavljali dijelovi koda koji pozivaju iste metode ali sa različitim implementacijama, u čistome kodu koristi se apstrakcija. Primjer toga bi bila metoda koja služi za izračun ulaznice za kino prema određenim godinama osobe. Kod svake osobe trebamo ispisati iznos računa, no implementacija je različita ovisno o njihovim godinama. Maloljetnim osobama iznos će biti besplatan, osobe od 18 do 50 godina plaćat će iznos od 30kn, a za starije osobe vrijedit će popust od recimo 20%.

8.4. Polimorfizam

Polimorfizam se koristi kod klasa koje posjeduju više različitih metoda, no s istim nazivom. Metode su drugačije implementirane, ali koriste se ista imena jer dijele istu

odgovornost. Prema autoru S. Srinidhi [13] polimorfizam se dijeli na statički i dinamički polimorfizam. Dinamički polimorfizam je kada program odlučuje koju će metodu odabrati u trenutku izvođenja, a ne kod sastavljanja programa. Statički polimorfizam predstavlja metode sa istim nazivom, no sa različitim brojem proslijeđenih parametara.

D. Phillips [14] navodi jako dobar primjer za primjenu polimorfizma. Radi se o programu za prikazivanje filmova, kod kojeg je potrebno definirati metodu za pokretanje filmova, no postoji problem, pokretanje ovisi o formatu koji može biti različit od filma do filma. Pa tako neki filmovi nisu komprimirani (.wav), dok su ostali komprimirani ali s različitim algoritmima (.mp3, .wma i .ogg). Ovdje je idealno koristiti polimorfizam kod klasa, dok će njihovi objekti sami odlučiti koju metodu za pokretanje treba odabrati ovisno o ekstenziji filma.

Korištenje polimorfizma u čistome kodu predstavlja veliku prednost, pošto se klase ne moraju više proširivati sa dodatnim klasama, već se mogu koristiti i postojeće klase samo sa različitim inačicama iste metode.

9. Pogreške i testiranje

Pisanje kodova ponekad ide bez problema, dok ponekad rezultat nije ono što smo prvobitno zamislili. Željeli mi to ili ne, uvijek dobijemo neke čudne i nezamislive pogreške (eng. error) ili bugove. Pogreške se mogu pojaviti bilo gdje u kodu, ne znajući gdje se uopće skrivaju, što može biti neopisivno frustrirajuće za programere. Njihov pronalazak možemo olakšati pomoću gotovih klasa koje će upravljati s pogreškama te pomoću testnih klasa.

9.1. Upravljanje pogreškama

Čisti kod nas upućuje na korištenje iznimka (eng. try catch exception) umjesto metoda koje vraćaju vrijednost grešaka. Prednosti korištenja iznimaka je čišći kod. Kada se ne koriste iznimke, potrebno je još dodatno provjeravati vraćenu vrijednost kako bi provjerili postoji li greška ili ne. Time se kreiraju dodatni i nepotrebni blokovi programa, dok s iznimkama možemo jednostavno odvojiti program. Iznimke odvajaju kod koji se vrši kada nema iznimka, znači kada je sve uredu, od onoga kada se dogodi iznimka. Iznimke se dohvaćaju dok nije nešto u redu, te se zatim u skladu s zadatkom obrađuju. Na primjeru iz dijela praktičnog rada, kod pojave iznimke, ispisuje se obavijest s greškom na ekranu korisnika:

```
try {
  $popis_gresaka = array("nazivTematike" => "naziv tematike",
    "opisTematike" => "opis tematike");
  $uneseni_podaci = $this->provjeri_popunjesnost_obaveznih_podataka
    ($popis_gresaka);
  if (!$uneseni_podaci) {
    return $this->prikaz_podataka_tematike($id_tematike);
  }
  $podaci_tematike = $this->_pripremi_podatke_azuriranja($id_tematike);
  $this->_odluka_o_zapisu_tematike($podaci_tematike);
  $this->preusmjeri(Postavke::dohvati_server_url() .
    "izlozbe/prikaz_administracija_izlozba/");
} catch (Exception $e) {
  $this->pripremi_greske($e->getMessage());
  $this->prikaz_podataka_tematike($id_tematike);
}
```

Suprotno, ukoliko ne bi koristili iznimke, trebalo bi dodatno provjeravati pojedine unesene podatke i modele. Druga prednost korištenja iznimaka je bolja preglednost pogreške. Umjesto da dobijemo brdu linija sa nerazumljivim opisom greške, dobivamo odgovarajuću

prilagođenu poruku. Prema principima čistoga koda R. C. Martin [3] ističe da iznimke poboljšavaju čitljivost i urednost kodova pomoću odvajanja ispravnog izvođenja koda i koda vezanog uz iznimke. Kodovi tako ne sadrže nepotrebne linije vezane za provjeru pogrešaka, već se taj dio koda izvodi dio tek kada se dohvati pojedina iznimka.

Nedostatak korištenja iznimaka je kreiranje različitih puteva izvođenja programa umjesto korištenja statusa, slično poput naredbe idi na (eng. goto), što je devijacija u programiranju. Druga negativna posljedica je kreiranje dodatnih zahtjeva u programu koji se također moraju uzeti u obzir.

9.2. Jedinični testovi

Jedan od bitnih dijelova izrade aplikacije čine testovi koji se kreiraju prilikom pisanja kodova, a ponekad i prije samog pisanja programa. Jedinični testovi su testovi koji se kreiraju kako bi se provjerila ispravnost izvođenja svakog pojedinog dijela aplikacije. Kreira se više jediničnih testova kako bi se „izolirao“ kod i provjerio izolirani kod, odnosno, radi li uistinu kako se očekuje.

Testovi čistih kodova također moraju biti čitljivi i jasni. Jedinični testovi nam pomažu kod brzog otkrivanja raznih pogrešaka programa, štede vrijeme i novac, mogu se višestruko koristiti i pouzdani su, poboljšavaju pokrivenost koda, smanjuju kompleksnost koda te pomažu kod procjene performansi i kreiranja dokumentacije aplikacije.

Jedinične testove skladno opisuje definicija prema R. C. Martinu [3] čija je skraćenica F.I.R.S.T. . Prema njoj, jedinični testovi predstavljaju brzinu (eng. Fast), čime se podrazumijeva da testovi moraju biti brzi. Neovisnost (eng. Independent), testovi ne bi trebali ovisiti jedan o drugome, nego se mogu pokretati zasebno. Ponovljivost (eng. Repeatable), govori kako se testovi moraju koristiti u bilo kojem okruženju. Samopotvrđivanje (eng. Self-Validating), testovi bi trebali vraćati jednostavne rezultate, točnije da li je kod prošao test ili nije. Pravovremenost (eng. Timely), jedinični testovi trebaju se kreirati prije produkcijskog koda ili paralelno s produkcijskim kodom, ali nikako ne nakon pisanja cijele aplikacije pošto se time otežava testiranje.

10. Refaktoriranje

Refaktoriranje ili redizajniranje koda je zahtjevan proces koji optimizira ili modificira postojeći kod kako bi se postigle bolje performanse programa ili kvaliteta samoga koda. Tokom procesa refakturiranja, program ne smije izgubiti svoj prvobitni cilj. Redizajniranje kodova nailazi odličnu primjenu kod čišćenja kodova. Refaktoriranje kodova ne znači da izmjene moraju biti velike, refaktoriranje se više fokusira na manje promjene koje su toliko male, da se ne isplate mijenjati govori autor M. Fowler [15]. Pojedine male promjene mogu puno značiti. Redizajniranje manjih promjena na programu smanjuje rizik od potencijalnih pogreški tokom daljnjeg procesa razvoja.

Refaktoriranje je proces koji troši novac i zahtijeva vrijeme, ali ne donosi nove prednosti kod aplikacije, pa čemu onda provoditi refaktoriranje programa? Proces redizajniranja ne ostvaruje direktno nove i uspješne prednosti, ali znatno povećava kvalitetu samoga koda aplikacije. Refaktoriranje olakšava buduće poslovne zahtjeve oko nadogradnje, modifikacije i održavanja programa. Pravilno refaktoriranje koje prati principe čistoga koda, može drastično povećati kvalitetu koda. Prednosti su mnogobrojne, poput bolje i lakše čitljivosti i razumljivosti koda, povećanja brzine i performansi programa, smanjenja količine koda, ali ipak potrebno je uzeti u obzir i da proces refakturiranja posjeduje i negativne posljedice.

Jedna od negativnih posljedica koje mogu nastati tokom procesa refakturiranja su potencijalne greške, bugovi ili upozorenja. Nitko ne može garantirati da će aplikacija nakon refakturiranja raditi kako je i do sada radila bez ikakvih poteškoća. Sljedeće negativno svojstvo procesa refakturiranja su novčani izdaci i potencijalni rizici. Programeri ne mogu odrediti točno vrijeme koje je potrebno za refaktoriranje aplikacije jer razna kašnjenja i promjene koje sa sobom povlači, predstavljaju veliki rizik za uspješno izvršavanje aplikacije. Aplikacije su usko povezane preko njihovim skriptnih jezika s bazom podataka, gdje dolazi do problema. Shemu baze podataka je teško izmijeniti, jer je uglavnom velika te njezino refaktoriranje zahtjeva mnogo vremena.

10.1. Pokazatelji refakturiranja

Potreba za procesom refakturiranja se javlja kada aplikacija nije pisana prema načelima čistog programiranja. Prvenstveno se tu smatra dupliciranje kodova, kada se pojave kodovi, funkcije i metode koje posjeduju sličnu ili identičnu strukturu. Duplicirani kodovi se mogu razriješiti sa malim izmjenama, primjerice ako se radi o klasama koje koriste iste metode, može se uvesti nasljeđivanje i definirati zajednička klasa sa tom metodom koju će naslijediti.

Ostali pokazatelji nepravilno pisanih kodova koji zahtijevaju refaktoriranje prema autoru M. Fowler-u [15] su:

- Velike metode – metode koje su prevelike trebaju se skratiti na manje, jednostavnije metode s zasebnim ciljem, jer čim je metoda duža, postaje sve teža za razumjeti
- Ogromne klase – su klase koje sadrže previše ciljeva, time posjeduju veliki broj inicijalizacije, gdje će se najvjerojatnije duplicirati kod. Rješenje velikih klasa je proširivanje klase ili subklase (eng. Extract Class or Extract Subclass.)
- Dugi popis parametra – rješenje je kreiranje objekta i pripadajućih klasa koje sadrže željene informacije
- Divergentna promjena – promjene pojedinih klasa mogu uzrokovati probleme kod drugih klasa, pa sa zatim moraju i ostale klase mijenjati. Kod češćih promjena korisno je kreirati posebnu klasu koju ćete modificirati da se mogu prilagoditi promjenama, dok ostale klase ostaju iste i proširuju (eng. extract) novo kreiranu klasu koja se mijenja.
- Operacija sačmaricom – modifikacija može uzrokovati potrebu za promjenom kod ostalih klasa, kako bi se izbjegao taj problem možete koristiti pomakni metodu i pomakni polje (eng. move method i mode field). Metode kreiraju rješenje za klasu koja se modificira, tako da se promjene izvršavaju u posebnoj klasi
- Lijene klase – suvišne klase, koje nemaju dovoljno smisla, a refaktoriranje je preskupo.
- Lanci poruka – niz poredanih metoda koje stalno dohvaćaju nove objekte, savjet kod ovog problema je proširivanje modela tamo gdje možete, kako bi se smanjilo broj dohvaćanja objekta.
- Komentari – korištenje komentara kod lošeg napisanog koda.

Proces refakturiranja se ne provodi kada je rok isporuke aplikacije blizu i ako ne posjedujete vrijeme predviđeno samo za testiranje aplikacije nakon promjene. Proces se ne provodi ako je potrebno više vremena i ako je refaktoriranje veće od ponovnog pisanja koda od početka. Redizajniranje koda se ne preporučuje kod stabilnih programa.

11. Izrada praktičnog dijela aplikacije

Ostatak rada posvećen je izradi praktičnog dijela rada, a to je refaktoriranje web aplikacije koristeći principe čistoga koda. Aplikacija je namijenjena kreiranju i održavanju izložbi vlakova, gdje se korisnici mogu samostalno prijaviti kako bi prezentirali svoj vlak. Na početnoj stranici, korisnici se prijavljuju na svoj korisnički račun ili ako ga ne posjeduju, mogu se registrirati. Kod kreiranja računa korisnik dobiva povratnu informaciju na svoj email i aktivacijski link, kojega mora aktivirati unutar 14 sati. Korisnici aplikacije su podijeljeni po ulogama, a to su ne-registrirani korisnik, registrirani korisnik, moderator i administrator. Svaka pojedina uloga je detaljno razrađena i objašnjena u priloženom prilogu 1.

Za arhitekturu aplikacije je odabran MVC (Model View Controller) model, koji će biti razrađen u nastavku.

Prikaz ERA modela se nalazi u prilogu 2 . Model prikazuje potrebne tablice za samu pohranu podataka aplikacije i njihovu međusobnu povezanost. Osim navedenih tablica kreirana je i posebna funkcija za dohvaćanje statusa izložbi, koja vraća podatke iz tablice status izložbe na temelju proslijeđenog id-a izložbe i virtualnog datuma.

11.1. Metode i tehnike rada

Sama aplikacija napravljena je većinskim dijelom u skriptnom jeziku PHP, a kontrola prikazivanja iskočnih prozora je kreirana korištenjem programskog jezika JavaScript. Aplikacijsko okruženje koje je korišteno za pisanje PHP i JavaScript kodova je Apache NetBeans IDE 12.2. Testiranje aplikacije je provedeno koristeći lokalno podignutu aplikaciju pomoću okruženja XAMPP. Za kreiranje baze podataka i ERA modela korišten je MySQL Workbench, a za pohranu stvarnih podataka i upravljanje administracijom MySQL-a putem weba je korišten softverski alat phpMyAdmin koji je također pokrenut lokalno putem XAMPP-a. Za prikaz stvarnih podataka i prikaz dizajna korišten je Smarty sustav predložaka, koji odvaja prikaz prezentacije (HTML i CSS) od same logike (PHP). Cijeli dizajn je vlastoručno kreiran preko prezentacijskog jezika HTML-a i stilskog jezika CSS-a.

11.2. Imenovanje kod PHP-a i struktura aplikacije

Jedna od bitnih stavki prije nego se krene u samu izradu aplikacije je odabir stila pisanja koda, a jednom kada se on odabere, njega se potrebno držati do kraja razvoja aplikacije. Refakturirana aplikacija napisana je koristeći skriptni programski jezik PHP te je stoga odabran

njegov standardni stil pisanja prema britanskoj softverskoj instituciji za tehnologije CodeIgniter [16].

Kod imenovanja mapi prvo početno slovo je veliko, dok su ostala slova mala. Imena koja u nazivu sadrže više riječi povezuju se u jednu koristeći donju povlaku. Ista praksa je i kod svih ostalih dijelova koda (funkcije, varijable, metode, svojstva …). Nazivi klasa koje predstavljaju model u aplikaciji prilagođeni su nazivima tablica koje se koriste iz baze. Svaki naziv klase započinje sa velikim početnim slovom, a ostala slova su mala. Nazivi funkcija, javnih metoda, varijabli i javnih svojstva započinju sa malim slovom, a privatna svojstva i metode započinju sa donjom povlakom i malim početnim slovom.

Kod povratnih vrijednost gdje se koristi tip boolean ili prazna vrijednost, sva slova su velika (TRUE, FALSE i NULL). Velika slova su i u slučaju logičkih operacija (OR i AND), a između svake riječi operacije dolazi po jedan razmak. Praznine se koriste za razdvajanje nepovezanih dijelova koda (najčešće s praznom linijom) i dijelova koji posjeduju unutarnje blokove (unutarnji blok se povlači za jedan tab udesno). Poduže linije kodova razdvojene su na više smislenih redova, poput većih SQL upita i kreiranja sesije.

11.2.1. Primjeri imenovanja kod aplikacije

Imena funkcija i metoda u skladu su sa svrhom i odgovornošću koju ta funkcija obnaša. Uglavnom se iz samog naziva može zaključiti koja je namjena funkcije te stoga nisu korišteni dodatni komentari. Sljedeći kod prikazuje neke od implementiranih funkcija:

```
private function _provjeri_podatke_kod_prijave($korisnik) {
  if ($this->_podaci_prijave_su_ispravni($korisnik)) {
  $this->_pripremi_podatke_korisnika($korisnik);
  } elseif ($korisnik->broj_neuspijesnih_prijava >= 3) {
    throw new Exception("Korisniči račun  {$korisnik->korisnicko_ime}
      Vam je blokiran. Molimo Vas kontaktirajte administratora");
  } else {
    $this->_trazi_ponovnu_prijavu($korisnik);
  }
}
private function _podaci_prijave_su_ispravni($korisnik) {
  $heshirana_lozinka = hash("sha256", $korisnik->lozinka .
    $korisnik->salt);
  return $heshirana_lozinka === $korisnik->lozinka_sha1 and
    $korisnik->broj_neuspijesnih_prijava <= 3;
}
private function _pripremi_podatke_korisnika($korisnik) {
```

```
    Sesija::kreiraj_korisnika( $korisnik->korisnicko_ime,
      $this->_virtualno_vrijeme, $korisnik->tip_korisnika_id);
    $korisnik->broj_neuspijesnih_prijava = 0;
    $this->_korisnici_repo->azuriraj($korisnik);
    $this->preusmjeri($this->_url_servera . "pocetna_stranica/index/");
  }
```

11.3. MVC okvir

Model View Controler (MVC) je okvir softverske arhitekture koji se koristi za odvajanje znatno različitih dijelova aplikacije u zasebne komponente ovisno o njihovoj zadaći i svrsi. MVC arhitektura se sastoji od triju dijelova: model (eng. Model), pogled (eng. View) i upravitelj (eng. Controller). Model služi za reprezentaciju podataka zajedno s njihovim svojstvima. Kroz model se definiraju i poslovna pravila te logika vezana uz same podatke. Pogled definira na koji način će se podaci iz modela prikazati krajnjem korisniku, a podaci se dohvaćaju i obrađuju koristeći upravitelj. Upravitelj zapravo čini poveznicu između modela i pogleda, on koristi modele kako bi stvorio podatke koje potom prosljeđuje pogledu za prikaz podataka. Glavna zadaća upravitelja je upravljanje korisničkim zahtjevima te kontrola i implementacija same logike aplikacije.

Temelj izrađene aplikacije je upravo MVC arhitektura. Kako svi modeli u aplikaciji posjeduju zajedničko svojstvo id, definirana je klasa model.php sa tim svojstvom te metodama za njegovo dohvaćanje i ažuriranje. Tu klasu potom nasljeđuju sve specifične klase modela. Klase modela su napravljene i nazvane prema tablicama definiranim u bazi podataka, a njihova svojstva su kreirana na temelju atributa tih tablica.

```
class Model {
  private $id;
  private function dohvati_id() {
    return $this->id;
  }
  private function postavi_id($id) {
    $this->id = $id;
  }
  public function __set($ime, $vrijednost) {
    $ime_funkcije = 'postavi_' . $ime;
      return $this->$ime_funkcije($vrijednost);
    }
  public function __get($ime) {
    $ime_funkcije = 'dohvati_' . $ime;
```

```
        return $this->$ime_funkcije();
      }
    }
```

Za prikaz podataka unutar pogleda, korišteni su Smarty sustavi predložaka. Svaka stranica posjeduje svoj zasebni predložak u kojem se nalazi kod HTML-a. Sve stranice dijele zajednički predložak navigacija i podnožje. Predložak za navigaciju sadrži sami početak prezentacijskog koda HTML-a te prikaz navigacije. Ovisno o ulozi prijavljenog korisnika navigacijski predložak prikazuje sve moguće dostupne stranice korisniku. Dok predložak podnožja zatvara HTML kod i prikazuje pojedine elemente stranice, poput prikaza autora stranice i virtualnog vremena. Prikaz svake pojedine stranice se omogućuje nakon što se učita navigacijski predložak, zatim se učita predložak same stranice i na kraju se učita podnožje stranice. Stvarne informacije koje se trebaju prikazati na stranici dobivene su putem pridruženog upravitelja. Upravitelj prosljeđuje sve potrebne informacije i učitava odabrani predložak.

Upravitelj za dohvaćanje i upravljanje podacima iz baze koristi posebno definirane klase repozitorija. Svrha repozitorija je komuniciranje i obavljanje SQL upita nad bazom podataka, a zadaća upravitelja je brinuti o logici aplikacije. Klase repozitorija definirane su tako da implementiraju određeno sučelje repozitorija u kojem su definirane sve potrebne metode koje taj repozitorij mora posjedovati. Uporaba sučelja nam omogućuje da u budućnosti lako zamijenimo ili dodamo nove repozitorije. Primjerice, ukoliko se javi potreba da se podaci osim iz MySQL baze učitavaju dodatno i iz MongoDb baze ili iz tekstualne datoteke, sve što je potrebno je da se kreira njihova implementacija tog repozitorija i ona se potom može koristiti na svim mjestima gdje se očekuje sučelje takvih repozitorija. Primjena repozitorija i sučelja nam tako omogućuje razdvajanje logike aplikacije i logike vezane uz bazu podataka (dohvaćanje, zapisivanje i ažuriranje podataka s baze podataka). Prednost ove strukture je to što nam olakšava upravljanje podacima te omogućuje neovisno pretraživanje i pristup sadržaju. Razne promjene i ažuriranja se jednostavno mogu provesti, što je ujedno i cilj i prednost čistoga koda. Repozitorij se može koristiti bilo gdje i onoliko puta koliko je potrebno.

Repozitoriji su kreirani na temelju pojedine grupe modela. Svaki repozitorij se implementira prema svojem vlastitom sučelju, dok je svako sučelje dodatno prošireno baznim sučeljem koje sadrži zajednička svojstva i metode svih repozitorija. Bazno sučelje repozitorija prikazano je sljedećim kodom.

```
interface Bazni_repozitorij_interface{
    public function dohvati_prema_id($id);
    public function kreiraj($uneseni_podaci);
    public function dohvati_listu();
```

```
        public function obrisi($id_zapisa);
        public function azuriraj($uneseni_podaci);
    }
```

Logika svake pojedine stranice se nalazi unutar upravitelja. Bazni upravitelj koji se uvijek i na svakoj stranici učitava je controller.php, on posjeduje zajedničke metode koje se koriste kod svih ostalih, iz njega izvedenih upravitelja. Primarna zadaća baznog upravitelja je omogućiti „iscrtavanje" potrebnih dijelova stranice. U ovu metodu se prosljeđuju osnovne informacije koje su potrebne kod učitavanja navigacijskog predloška. Najprije se u samome upravitelju dodijele potrebna svojstva, zatim se inicijalizira Smarty i kreira sesija, te ako je korisnik prijavljen provjerava se isticanje sesije i zapisuje se u dnevnik posjet stranici. Bazni upravitelj posjeduje još i metode za pripremu pogrešaka pripremi_greske ($greske) koja ujedno služi i za njihov prikaz.

Ostale metode koje implementira bazni upravitelj su:

- pripremi_podatke_stranice ($model) – spaja sva polja podataka stranice u jedno, što je potrebno kod prikaza željenih podataka na frontu.
- preusmjeri($url_stranice) – preusmjerava na željenu stranicu.
- provjera_mandatornih_podataka($popis_uvjeta) – metoda provjerava ispunjenost unesenih podataka kod raznih obrazaca, pomoću proslijeđene liste uvjeta se dobiva greška. Ako nije ispunjeno obavezno polje, metoda vraća FALSE. Dok inače, ako nema greške, će vraćati model unesenih podataka.
- autorizacija_korisnika($uloga_autorizacije) - metoda uspoređuje i provjerava pristup prijavljenog korisnika, na temelju proslijeđene uloge koja je minimalna da se može posjetiti stranica. Ne prihvatljivog korisnika vraća na početnu stranicu, dok prihvatljivi korisnici mogu neometano posjetiti stranicu.

Ostali PHP upravitelji su kreirani na temelju pet različitih cjelina aplikacije, ovisno o njihovoj namjeni i zadaćama koje obavljaju. Navedeni upravitelji su: upravitelj za izložbe, upravitelj korisnika, upravitelj početne stranice, upravitelj postavkama stranice i upravitelj za vlakove. Upravitelj izložbi upravlja sa stranicama koje su usko vezane uz izložbe. Pa tako upravlja sa stranicama za kreiranje i modifikaciju tematike, dodjelu moderatora tematici, prikaz izložbi korisnicima, upravljanje izložbama i ocjenjivanje izložbi. Upravitelj korisnika upravlja sa stranicama prijave, registracije i odjave korisnika. Upravitelj početne stranice upravlja početnom stranicom, te provjerava aktivacijski link novih korisnika. Početna stranica je dostupna svim ulogama korisnika, te iz tog razloga upravitelj početne stranice sadrži još jednu dodatnu ulogu, a to je prikaz stranice o autoru. Upravitelj postavkama stranice posjeduje sljedeće odgovornosti: prikaz stranice o postavkama, ažuriranje sesije, kolačića i uvjeta korištenja aplikacije, prikaz dnevnika korištenja aplikacije uz mogućnost pretraživanja te

kreiranje i vraćanje sigurnosne kopije baze podataka o vlakovima, prijavama vlakova, materijalima i ocjenama korisnika. Posljednji izrađeni upravitelj namijenjen je upravljanju sa vlakovima. On posjeduje prikaz podataka o dodanim vlakovima korisnika i logiku o ažuriranju vlakova, osim vlakova posjeduje i metode za dodavanje željenih materijala korisnika i prikaz prijave vlakova na određenu izložbu, gdje moderator ili administrator može prihvaćati ili odbijati prijave korisnika. Pojedini upravitelji bit će detaljnije razjašnjeni uz prikaz njihovih kodova u daljnjim poglavljima.

11.4. Struktura mapa

Struktura mapa igra veliku ulogu kod čistih kodova. Njihova imena moraju biti dobro i smisleno nazvana - ovisno o njihovim sadržajima radi lakšeg pretraživanja. Struktura mapi izrađene aplikacije prilagođena je MVC arhitekturi. Korijenska mapa naziva se projektni zadatak, a sadrži mape controllers, CSS, izvorne datoteke, JavaScript, models, multimedija, repositories, templates, templates_c, vanjske_biblioteke i webroot.

Mapa controllers sadrži jednu roditeljsku klasu controller.php i ostale klase koje su djeca klase controller.php. Mapa CSS sadrži sve stilske datoteke koje su korištene tokom projekta, a napisane su stilskim jezikom CSS-om. Mapa izvorne datoteke koristi se za pohranu podataka stranice poput trajanja kolačića, sesije, sigurnosne kopije i baze podataka. Sljedeća mapa naziva se JavaScript, u nju su pohranjeni svi JavaScript kodovi. Oni se koriste uglavnom kod prikazivanja i zatvaranja pop up prozora. Mapa models sadrži jednu klasu roditelja model.php te ostale klase koje su djeca klase model.php, pojedine klase se još dodatno proširuju, no o tome će biti govora u dijelu o objektno orijentiranom programiranju. Mapa multimedija sadrži potrebne slike kod prikaza stranica. Tamo se pohranjuju i materijali definirani od strane korisnika. Prvotno se kreira početna mapa sa imenom korisnika, zatim ovisno o odabranoj vrsti materijala se kreira mapa vrste materijala (točnije slika, video, gif ili audio), a unutar te mape se pohranjuju stvarni materijali.

Mapa repositories sadrži sve datoteke klasa odgovornih za dohvaćanje i ažuriranje podataka modela. Konkretne implementacije repozitorija podatke dohvaćaju i ažuriraju koristeći SQL upite nad bazom podataka. Repozitoriji su raspodijeljeni i nazvani prema sučeljima (eng. interface) koje implementiraju i imaju sufiks _db_repozitorij. Dok se sama specifikacija njihovih sučelja pohranjuje u mapu interfaces. Sučelja su nazvana prema modelima čijim podacima upravljaju te im naziv sadrži sufiks _repozitorij_interface. Mape templates i templates_c koristi Smarty sustav predložaka. Mapa templates sadrži datoteke prezentacijskog jezika HTML-a, te predstavlja model. Ona je također podijeljena na slične uloge kao što je slučaj i kod upravitelja. Dodatno, sadrži još i zajedničke predloške koji se

koriste kod prikaza svih stranica. Sustav predložaka Smarty pohranjen je u mapu vanjske biblioteke. Mapa webroot sadrži datoteke vezane uz početno pokretanje stranice i provjeru linkova. Posljednja mapa je mapa utils koja sadrži klase korištene od strane većine ostalih klasa. Njih čine klase za upravljanje sa sesijom te za upravljanje postavkama, odnosno općenitim zahtjevima (poput dohvaćanja virtualnog vremena i url-a servera). Izvorna datoteka projektni zadatak dodatno još sadrži PHP datoteke dispatcher, request i router koje su zadužene za prosljeđivanje parametara na druge stranice i općenito za upravljanje url linkovima.

11.5. Izrada aplikacije

Na samom početku izrade aplikacije kreirane su konfiguracijske datoteke htaccess. Prva služi za preusmjerivanje svih zahtjeva na mapu webroot, gdje se učitava druga htaccess datoteka. Ona provjerava ispravnost linkova (primjerice da li sadrže slova i brojeve, kose crte i slično), a zatim ako su ispravnog formata prosljeđuje ih na PHP datoteku indeks.php koja inicijalizira klasu dispatcher. Dispatcher inicijalizira klase request i router koje služe za usmjeravanje na odabrani upravitelj i metodu koja se u njemu treba izvršiti, također prosljeđuju varijable iz zahtjeva ukoliko je to potrebno. Klasa request dohvaća puni link stranice na koju se pristupa, dok klasa router dobivene linkove razdjeljuje na: upravitelj, akciju (metode kod pojedinih upravitelja) i parametar. Ukoliko router ne dobije nikakav link, preusmjerava korisnika na pretpostavljenu stranicu "Pocetna_stranica", na akciju "indeks" s praznim parametrom. Inače, ukoliko je link ispravan, korisnika se preusmjeruje na stranicu upravitelja s pridruženom metodom i parametrima. Primjer tih klasa nalazi se u nastavku.

```
class Dispatcher {
  private $_zahtjev;
  public function dispatch() {
    $this->_zahtjev = new Request();
    Router::parse($this->_zahtjev->url, $this->_zahtjev);
    $controller = $this->load_controller();
    call_user_func_array([$controller, $this->_zahtjev->akcija],
      $this->_zahtjev->parametar);
  }
  public function load_controller() {
    $name = $this->_zahtjev->kontroler . "_controller";
    $file = ROOT . 'Controllers/' . $name . '.php';
    require($file);
    $controller = new $name();
    return $controller;
```

```
  }
}
class Router {
  static public function parse($url, $zahtjev){
    $bazni_url = trim($url);
    if ($bazni_url == "/Projektni_zadatak/"){
      $zahtjev->kontroler = "Pocetna_stranica";
      $zahtjev->akcija = "index";
      $zahtjev->parametar = [];
    } else {
      $puni_url = explode('/', $bazni_url);
      $podijeljeni_url = array_slice($puni_url, 2);
      $zahtjev->kontroler = $podijeljeni_url[0];
      $zahtjev->akcija = $podijeljeni_url[1];
      $zahtjev->parametar = array_slice($podijeljeni_url, 2);
    }
  }
}
```

11.5.1. Klase modela

Klase modela služe za kreiranje i strukturiranje samih podataka, kao i za njihovu definiciju poslovne logike. Bazna klasa koja se koristi kod modela je klasa model.php, nju nasljeđuju sve ostale specifične klase modela te je njezin kod prikazan ranije u poglavlju. Sljedećim kodom prikazana je klasa Korisnik koja nasljeđuje klasu Model:

```
class Korisnik extends Model {
  private $_ime;
  private $_prezime;
  private $_korisnicko_ime;
  private $_lozinka;
  …
  private $_salt;
  private $_datum_kreiranja;
  public function dohvati_ime() {
    return $this->_ime;
  }
  public function postavi_ime($ime) {
    $this->_ime = $ime;
  }
  public function dohvati_korisnicko_ime() {
```

```
    return $this->_korisnicko_ime;
  }
  public function postavi_korisnicko_ime($korisnicko_ime) {
    if (empty($korisnicko_ime)) {
      throw new Exception("Niste unijeli korisnicko ime.");
    }
    $this->_korisnicko_ime = $korisnicko_ime;
  }
  public function dohvati_lozinka() {
    return $this->_lozinka;
  }
  public function postavi_lozinka($lozinka) {
    $uzorak = '/^(?!.*(.)\1{3})((?=.*[\d])(?=.*[A-Za-z])|(?=.*[^\w\d\s])
      (?=.*[A-Za-z])).{8,20}$/';
    if (empty($lozinka)) {
      throw new Exception("Niste unijeli lozinku.");
    }
    if (!preg_match($uzorak, $lozinka)) {
      throw new Exception("Format: Lozinka ima manje od 8 znakova "
        . "ili više od 20 znakova "
        . "ili nema 1 alfanumerički znak "
        . "ili nema najmanje 1 broj "
        . "ili nema specijalni znak "
        . "ili se ponavljaju 3 ista znaka!"
        . "<br>");
      }
      $this->_lozinka = $lozinka;
    }
    public function dohvati_salt() {
      return $this->_salt;
    }
    public function postavi_salt($salt) {
      if (empty($salt)) {
        $text = md5(uniqid(rand(), TRUE));
        $this->_salt = substr($text, 0, 3);
      } else {
        $this->_salt = $salt;
      }
    }
...
```

11.5.2. Klase repozitorija

Nakon izrade modela slijedi razrada sučelja i klasa repozitorija koji će biti odgovorni za punjenje podataka modela kao i njihovo ažuriranje. Najprije je kreirano njihovo bazno sučelje koje je ranije opisano, a zatim na sličan način i ostala sučelja koja ga proširuju.

Potom su definirane konkretne klase repozitorija koje dohvaćaju i ažuriraju podatke modela putem baze. One implementiraju pripadna sučelja, a za povezivanje na samu bazu podataka koriste klasu Baza.php dobivenu za korištenje na kolegiju Web dizajn i programiranje. Za pristup bazi koristi se lokalni phpMyAdmin server pokrenut preko aplikacije XAMPP. Klasa Baza refaktorirana je prema načelima čistoga koda, te je prilikom njezinog refakturiranja upotrjebljen oblikovni obrazac Singleton koji osigurava da se samo jedna instance te klase nalazi u cijeloj aplikaciji. Metoda spoji_db služi za kreiranje veze na bazu, dok metoda zatvori_db raskida vezu prema bazi, metoda select_db dohvaća podatke iz baze na temelju proslijeđenog SQL upita, dok metoda update_db pohranjuje i ažurira proslijeđene podatke iz modela na pripadnu tablicu iz baze. Kreirane su i dvije dodatne metode koje služe za jednostavniji i brži postupak dohvaćanja i zapisivanja podataka. Njih koriste ostale metode namijenjene upravljanju bazom, a kreirane su iz razloga kako bi se izbjegnulo dupliciranje koda i doprinijelo čistoći koda.

Klase repozitorija kao izvor podataka koriste bazu, te su shodno tome njihova imena izvedena na temelju pripadnih tablica iz baze. Kreirano je ukupno dvanaest repozitorija od ukupno šesnaest tablica iz baze, iz razloga što pojedine tablice nisu potrebne za nadogradnju ili modifikaciju, a koriste se kao pomoćne tablice i služe uglavnom za enumeraciju pojedinih svojstava modela. Repozitoriji implementiraju svoje istoimeno sučelje, koje sadrži pet osnovnih metoda naslijeđenih iz baznog sučelja te ostale specifične metode ovisno o modelima kojima konkretni repozitorij upravlja. Iznimka su repozitoriji za upravljanje podacima dnevnika i vrstama materijala vlakova koji ne zahtijevaju metode iz baznog sučelja, već imaju nekoliko svojih specifičnih metoda namijenjenih dohvaćanju i/ili zapisivanju podataka u bazu. Repozitorij koji se često koristi te posjeduje svih pet metoda baznog sučelja je vlak_db_repozitorij.php:

```
class Vlak_db_repozitorij implements Vlak_repozitorij_interface {
     private $_db;
     private $_dnevnik;
     private $_korisnicko_ime;
     public function __construct() {
       $this->_db = Baza::dohvati_instancu();
       $this->_dnevnik = new Dnevnik_db_repozitorij;
       $this->_korisnicko_ime = Sesija::dohvati_korisnicko_ime();
```

```
  }
  public function dohvati_prema_id($id_vlaka) {
    $sql = "SELECT * FROM vlak v WHERE v.id = {$id_vlaka}";
    $rezultat_sqla_vlaka = $this->_db->dohvati_podatke($sql);
    Return $this->_kreiraj_model_vlaka_od_baze($rezultat_sqla_vlaka[0]);
  }
  private function _kreiraj_model_vlaka_od_baze($rezultat_sqla_vlaka) {
    $vlak = new Vlak;
    $vlak->id = $rezultat_sqla_vlaka["id"];
    $vlak->naziv = $rezultat_sqla_vlaka["naziv"];
    $vlak->max_brzina = $rezultat_sqla_vlaka["max_brzina"];
    $vlak->broj_sjedala = $rezultat_sqla_vlaka["broj_sjedala"];
    $vlak->opis = $rezultat_sqla_vlaka["opis"];
    $vlak->vrsta_pogona_id = $rezultat_sqla_vlaka["vrsta_pogona_id"];
    $vlak->vlasnik_id = $rezultat_sqla_vlaka["vlasnik_id"];
    return $vlak;
  }
  public function kreiraj($uneseni_podaci) {
    $vlak = $this->_kreiraj_model_vlaka_od_fronta($uneseni_podaci);
    $sql = "INSERT INTO vlak (naziv, max_brzina, broj_sjedala, opis,
      vrsta_pogona_id, vlasnik_id) VALUES "
    . "('{$vlak->naziv}', {$vlak->max_brzina}, {$vlak->broj_sjedala}, "
    . "'{$vlak->opis}', {$vlak->vrsta_pogona_id}, {$vlak->vlasnik_id})";
    $this->_dnevnik->kreiraj_dnevnik($this->_korisnicko_ime,
      Tip_dnevnika::RAD_S_BAZOM, $sql);
    $this->_db->zapisi_podatke($sql);
  }
  private function _kreiraj_model_vlaka_od_fronta($uneseni_podaci) {
    $vlak = new Vlak;
    if (isset($uneseni_podaci["id"])) {
      $vlak->id = $uneseni_podaci["id"];
    }
    $vlak->naziv = $uneseni_podaci["nazivVlaka"];
    $vlak->max_brzina = $uneseni_podaci["maxBrzina"];
    $vlak->broj_sjedala = $uneseni_podaci["brojSjedala"];
    $vlak->opis = $uneseni_podaci["opisVlaka"];
    $vlak->vrsta_pogona_id = $uneseni_podaci["vrstaPognona"];
    $vlak->vlasnik_id = $uneseni_podaci["vlasnik_id"];
    return $vlak;
} …
```

Navedeni primjer prikazuje način kreiranja klase repozitorija. Unutar metode konstruktora dohvaća se instanca baze koja se potom pohranjuje u privatnu varijablu $_db, ona će se kasnije koristiti kod samih upita nad bazom. Osim instance baze, unutar konstruktora inicijalizirana su i privatna svojstva za repozitorij dnevnika te za pohranu korisničkog imena. Ova svojstva kasnije se koriste kod ažuriranja kako bi se pohranile akcije korisnika (bilo da je to kreiranje, ažuriranje ili brisanje podataka od strane korisnika) u dnevnik korištenja aplikacije unutar baze. Kod ovog primjera poveće metode razdvojene su na niz manjih privatnih metoda od kojih svaka obavlja svoj dio zadatka. Primjerice metoda kreiraj ima jednu zadaću, a to je unos podataka modela u bazu podataka, no najprije je potrebno kreirati model podataka, a zatim i SQL upit kojim će se izvršiti zapis podataka u bazu. Stoga je radi preglednosti, čitljivosti i urednosti metoda kreiraj razdvojena na dva dijela, prvi dio služi za kreiranje modela, dok drugi dio obavlja samo zapisivanje podataka u bazu. Repozitorij vlaka osim pet baznih metoda sadrži još četiri metode deklarirane njegovim sučeljem:

```
interface Vlak_repozitorij_interface extends Bazni_repozitorij_interface
{
  public function prikaz_vlakova_korisnika();
  public function dohvati_slobodne_vlakove_korisnika($model);
  public function zapisi_podatke_iz_sigurnosne_kopije($zapis);
  public function obrisi_sve_vlakove();
}
```

Svaka od navedenih metoda kreira model vlakova (ako su povezani sa stvarnim podacima) i vrši pripadne SQL upite nad bazom podataka. Pojedine metode ne kreiraju model podataka, već samo vrše SQL upit, primjerice metode za brisanje podataka i njihovo vraćanje iz sigurnosne kopije. Kod njih nije potrebno korisniku prikazivati realne podatke, nego ih je potrebno samo obraditi kroz komunikaciju s bazom. Sljedeći kratki primjer pokazuje dvije navedene metode:

```
public function zapisi_podatke_iz_sigurnosne_kopije($zapis) {
  $sql = "INSERT INTO vlak VALUES " . $zapis;
  $this->_db->zapisi_podatke($sql);
  $this->_dnevnik->kreiraj_dnevnik($this->_korisnicko_ime,
    Tip_dnevnika::RAD_S_BAZOM, $sql);
}
public function obrisi_sve_vlakove() {
  $sql = "DELETE FROM vlak";
  $this->_db->zapisi_podatke($sql);
  $this->_dnevnik->kreiraj_dnevnik($this->_korisnicko_ime,
    Tip_dnevnika::RAD_S_BAZOM, $sql);
}
```

Ostali repozitoriji kreirani su na sličan način, samo što su drugačiji upiti i modeli koje koriste za upravljanje podacima u bazi.

11.5.3. Klase upravitelja

Već u ranijem poglavlju kod MVC arhitekture dane su osnovne informacije i iznesene zadaće svakog pojedinog upravitelja. U ovom poglavlju slijedi njihov detaljan pregled te primjeri iz koda izrađene aplikacije. Bazni upravitelj i njegove metode opisani su u ranijem poglavlju, a sljedećim kodom prikazane su njegove najbitnije metode:

```
function iscrtaj($naziv_predloska) {
  $url_predloska = ucfirst(str_replace('_controller', '',
    get_class($this))) . '/' . $naziv_predloska . '.tpl';
  $navigacijski_podaci["putanja"] = $this->_putanja;
  $navigacijski_podaci["dizajn"] = Sesija::DIZAJN;
  $navigacijski_podaci["virtualnoVrijeme"] = $this->_virtualno_vrijeme;
  $navigacijski_podaci["url_stranice"] = "$_SERVER[REQUEST_URI]";
  $this->pripremi_podatke_stranice($navigacijski_podaci);
  $this->smarty->assign($this->podaci_stranice);
  $this->smarty->display("dijeljeno/navigacija.tpl");
  if (isset($this->podaci_stranice["greske"])) {
    $this->smarty->display("dijeljeno/greske.tpl");
  }
  $this->smarty->display($url_predloska);
  $this->smarty->display("dijeljeno/podnozje.tpl");
}
function pripremi_podatke_stranice($podaci) {
  $this->podaci_stranice = array_merge($this->podaci_stranice, $podaci);
}
function pripremi_greske($greske) {
  if (isset($greske)) {
    if (!isset($this->podaci_stranice["greske"])) {
      $this->podaci_stranice["greske"] = [];
    }
    if (!is_array($greske)) {
      $greske = array($greske);
    }
    $this->podaci_stranice["greske"] = array_merge($this->podaci_
      stranice["greske"], $greske);
  }
}
```

Metoda iscrtaj najprije kreira link predloška kojeg je potrebno učitati te dodjeljuje osnovne informacije za ispravno prikazivanje navigacijskog predloška. Informacije pohranjuje u privatno polje podataka koje je namijenjeno prosljeđivanju potrebnih informacija pogledu kako bi se tamo prikazale korisniku. Polje se modificira koristeći metodu "pripremi podatke stranice" koja spaja ulazne podatke s navedenim poljem u jedno. Tako kreirano polje zatim se dodjeljuje predlošku korištenjem naredbe assign. Na samome kraju metode iscrtaj, učitavaju se potrebni Smarty predlošci (navigacija, greške ukoliko su se pojavile, zadani predložak upravitelja i podnožje). Metoda za pripremu grešaka služi za spajanje grešaka u jedno polje, koje se potom pohranjuje unutar privatne varijable s podacima stranice koji zatim služe za prikaz podataka unutar predložaka.

Ostali upravitelji nasljeđuju prethodno prikazan bazni upravitelj, a osim njegovih metoda implementiraju i sebi svojstvene metode. Njihova zadaća je upravljanje zahtjevima klijenta i logikom prikaza stranice kao i provjera ispunjenosti obrazaca tamo gdje je to potrebno. Kontrole vezane uz podatke modela nalaze kod njihovih metoda za postavljanje pojedinih svojstava (primjerice provjera ispravnosti email adrese i slično) te u slučaju krivog unosa vraćaju prilagođenu iznimku sa opisom greške. Za provjeru poslovnih pravila kao što je mandatornost popunjenosti pojedinih polja upravitelji pozivaju metodu baznog upravitelja "provjeri popunjenost obaveznih podataka", kojoj se prosljeđuje popis obaveznih polja. U njoj se potom na temelju zadanih uvjeta provjeravaju sva input polja te ukoliko se naiđe da neko od polja ne zadovoljava uvjet dodaje se greška u popis grešaka.

```
public function provjeri_popunjesnost_obaveznih_podataka($popis_uvjeta){
  $lista_greskaka = [];
  $model = array();
  foreach ($_POST as $kljuc => $vrijednost) {
    $model[$kljuc] = $vrijednost;
    if (empty($popis_uvjeta[$kljuc])) {
      CONTINUE;
    }
    if (empty($vrijednost)) {
      $greska = "Niste popunili: " . $popis_uvjeta[$kljuc] . "<br>";
      array_push($lista_greskaka, $greska);
    }
  }
  if (empty($lista_greskaka)) {
    return $model;
  }
  $this->pripremi_greske($lista_greskaka);
  return FALSE; }
```

Kada klijent posjeti odabranu stranicu klasa Router preusmjerava zahtjev na odabrani upravitelj te pripadnu metodu s odabranim parametrima, ukoliko se oni šalju. Primjerice, posjetom stranici "vaši vlakovi" zahtjev se preusmjeruje na upravitelj "vlakovi" te njegovu metodu "prikaz vlakova korisnika". Metoda dodjeljuje sve potrebne informacije za prikaz predloška, a zatim se poziva metoda za pripremu podataka i njihovo iscrtavanje. Ostali prikazi stranica rade na isti način.

```
function prikaz_vlakova_korisnika() {
  $this->provjera_autorizacije_korisnika(uloga_korisnika::PRIJAVLJENI_
    KORISNIK);
  $podaci_fronta["naslov_stranice"] = "Vaši vlakovi";
  $podaci_fronta["opis_stranice"] = "Stranica prikazuje dodane vlakove
    korisnika i omogućuje upravljanje s njima.";
  $podaci_fronta["lista_vlakova"] = $this->_vlak_repo->prikaz_vlakova_
    korisnika();
  $this->pripremi_podatke_stranice($podaci_fronta);
  $this->iscrtaj("prikaz_vlakova");
}
```

Ažuriranje i kreiranje novih zapisa modela vrši se preko metode ažuriraj. Kod se izvršava koristeći try catch blok, pomoću kojeg se dohvaćaju iznimke koje mogu nastati prilikom kreiranja modela. Također tamo se provjerava i da li podaci zadovoljavaju uvjete o mandatornosti polja te se odlučuje o načinu zapisa, ukoliko je proslijeđen id postojeći model se samo ažurira u bazi, a ako nije tada se kreira novi zapis. Kod dohvaćanja iznimki kreira se poruka kojom se traži ponovan unos podataka. Ostale metode za unos raznih podataka kreirane su na sličan način:

```
function azuriraj_vlak_korisnika($id_vlaka) {
  $this->provjera_autorizacije_korisnika(uloga_korisnika::
    PRIJAVLJENI_KORISNIK);
  try {
    $popis_gresaka = array("nazivVlaka" => "naziv vlaka",
      "maxBrzina" => "maksimalnu brzinu vlaka",
      "brojSjedala" => "broj sjedala vlaka",
      "opisVlaka" => "opis vlaka",
      "vrstaPognona" => "odabir vrste pogona",
      "noviNazivPogona" => "naziv novog pogona",
      "noviOpisPogona" => "opis novog pogona");
    $uneseni_podaci = $this->provjeri_popunjesnost_obaveznih_podataka
      ($popis_gresaka);
    if (!$uneseni_podaci) {
```

```
        return $this->prikaz_podataka_kod_azuriranja_vlak($id_vlaka);
      }
      $podaci_korisnika = $this->_korisnici_repo->dohvati_prema_korisnicko
        _ime($this->_korisnicko_ime);
      $uneseni_podaci["vlasnik_id"] = $podaci_korisnika->id;
      $uneseni_podaci["id_azuriranja"] = $id_vlaka;
      $this->_odluka_zapisa_podataka($uneseni_podaci, $id_vlaka);
      $this->preusmjeri(Postavke::dohvati_server_url() . "vlakovi/prikaz_
        vlakova_korisnika/");
    } catch (Exception $e) {
      $this->pripremi_greske($e->getMessage());
      $this->prikaz_podataka_kod_azuriranja_vlak($id_vlaka);
    }
  }
  private function _odluka_zapisa_podataka($uneseni_podaci, $id_vlaka) {
    if ($this->_provjera_unosa_novog_pogona($uneseni_podaci)) {
      $uneseni_podaci["vrstaPognona"] = $this->_pripremi_odabranu_vrstu_
        pogona($uneseni_podaci);
    }
    if (empty($id_vlaka)) {
      $this->_vlak_repo->kreiraj($uneseni_podaci);
    }
    if (!empty($id_vlaka)) {
      $uneseni_podaci["id_vlaka"] = $id_vlaka;
      $this->_vlak_repo->azuriraj($uneseni_podaci);
    }
  }
  private function _provjera_unosa_novog_pogona($uneseni_podaci) {
    return $uneseni_podaci["noviNazivPogona"] != "nijePopunjeno" AND
      $uneseni_podaci["noviOpisPogona"] != "nijePopunjeno";
  }
  private function _pripremi_odabranu_vrstu_pogona($uneseni_podaci){
    $this->_vrsta_pogona_repo->kreiraj($uneseni_podaci);
    $vrsta_pogona = $this->_vrsta_pogona_repo->dohvati_id_vrste_pogona
      ($uneseni_podaci);
    return $vrsta_pogona->id;
  }
```

Primjer iznad vjerno prikazuje karakteristike čistog koda. U njemu su metode svedene na minimalnu razinu odgovornosti, pa tako poneke od njih sadrže tek nekoliko linija koda. Samu logika u blokovima grananja je razdvojena na manje privatne metode kako glavna

metoda ne bi sadržavala raznolike provjere i odgovornosti. U suprotnom ona bi bila nepregledna i nerazumljiva. Razdvajanjem na manje metode koje vraćaju podatak tipa boolean postignuta je bolja čitljivost koda. Također pošto se koristi više manjih metoda, svaka od njih ima odgovarajući naziv koji ujedno opisuje njenu funkciju čime je izbjegnuta potreba za dodatnim komentarima i razjašnjenjima koda. Metode za brisanje pojedinih modela su također vrlo jednostavne i pregledne. Kod njih se prvo provjerava ima li korisnik potrebna ovlaštenja. Ukoliko ima, pozove se odgovarajuća metoda za brisanje iz repozitorija, a zatim se korisnika preusmjeri na odgovarajuću stranicu za nastavak rada:

```
function obrisi_valk_korisnika($id_vlaka) {
  $this->provjera_autorizacije_korisnika(uloga_korisnika::
    PRIJAVLJENI_KORISNIK);
  $this->_vlak_repo->obrisi($id_vlaka);
  $this->preusmjeri(Postavke::dohvati_server_url() . "vlakovi/prikaz_
    vlakova_korisnika/");
}
```

11.5.4. Posebne klase

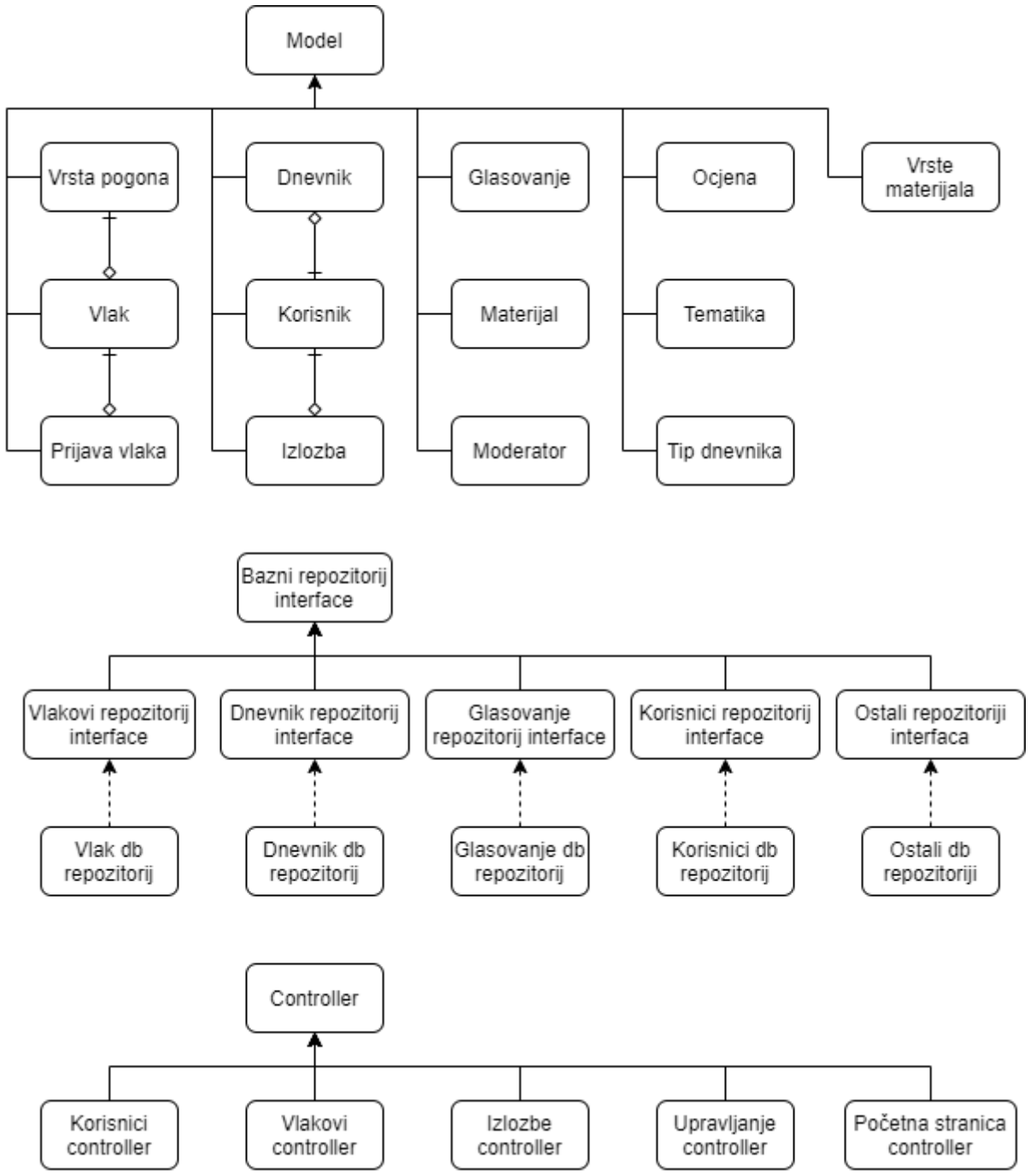
Metode koje posjeduju opće funkcionalnosti, kao što je primjerice dohvat url adrese stranice, dohvat virtualnog vremena ili dohvat određenog teksta iz txt dokumenta nalaze se u klasi postavke. Metode te klase se mogu zatim jednostavno i lako koristiti u drugim dijelovima aplikacije. Druga izdvojena klasa sa općim funkcionalnostima je klasa sesija koja služi za upravljanje podacima sesije, a preuzeta je na kolegiju web dizajn i programiranju. Njezin kod je redizajniran u skladu s potrebama trenutne aplikacije prilikom čega su dodane metode za brzo i jednostavno dohvaćanje potrebnih podataka iz sesije.

11.6. Objektno orijentirano programiranje

Prvi korak kod objektno orijentiranog programiranja je odabir prikladne arhitekture na kojoj će se temeljiti aplikacija. Za izradu aplikacije iz praktičnog dijela rada odabrana je MVC arhitektura pošto se radi o web aplikaciji koja na temelju zahtjeva korisnika upravlja podacima o izložbama. Objektno orijentirani programi zasnivaju se na skupu međusobno povezanih objekata koji se realiziraju preko samih klasa. Pritom objektno orijentirano programiranje koristi sljedeće koncepte: učahurivanje, apstrakciju, nasljeđivanje i polimorfizam. Upravo ti koncepti korišteni su i prilikom izrade aplikacije iz rada.

Nasljeđivanje je korisno kako bi se skratilo vrijeme pisanja samoga koda i izbjeglo nepotrebno ponavljanje određenih dijelova koda. U izrađenoj aplikaciji nasljeđivanje se prvenstveno koristi kod klasa modela, zatim kod izrade repozitorija i upravitelja kako je bilo i

navedeno u prethodnim poglavljima. Skriptni jezik PHP ne posjeduje mogućnost višestrukog nasljeđivanja, već kako bi proširili klase sa većim brojem klasa mora se kreirati niz klasa koje jedna drugu proširuju. Na sljedećoj slici prikazan je dijagram klasa izrađene aplikacije.



Slika 1. Dijagram klasa [autorski rad]

Apstrakcija je u aplikaciji primijenjena kod implementacije baznog sučelja repozitorija kao i njegovih specifičnih inačica koje to sučelje nasljeđuju. Konkretne klase koje potom implementiraju zadana sučelja su namijenjene upravljanju podacima iz baze, kao što govori i njihov sufiks u nazivu „_db_repozitorij“. One izvršavaju iste zadatke definirane baznim

sučeljem svaka na sebi svojstven način koristeći pripadne modele podataka. Osim tih metoda definiraju i metode specifičnog sučelja koje implementiraju. Primjerice metoda kreiraj iz baznog sučelja služi za kreiranje generičkog modela koji ovisi od repozitorija do repozitorija, sljedeći kod predstavlja implementaciju te metode unutar konkretne klase vlakovi_db_repozitorij.

```
public function kreiraj($uneseni_podaci) {
  $vlak = $this->_kreiraj_model_vlaka_od_fronta($uneseni_podaci);
  $sql = "INSERT INTO vlak (naziv, max_brzina, broj_sjedala, opis,
    vrsta_pogona_id,“
  . vlasnik_id) VALUES "
  . "('{$vlak->naziv}', {$vlak->max_brzina}, {$vlak->broj_sjedala}, "
  . "'{$vlak->opis}', {$vlak->vrsta_pogona_id}, {$vlak->vlasnik_id})";
  $this->_dnevnik->kreiraj_dnevnik($this->_korisnicko_ime,
    Tip_dnevnika::RAD_S_BAZOM, $sql);
  $this->_db->zapisi_podatke($sql);
}
```

Klasa prijave vlaka:

```
public function kreiraj($uneseni_podaci) {
  $prijava_vlaka = new Prijava_vlaka;
  $prijava_vlaka->id_vlaka = $uneseni_podaci["odabirVlakaZaPrijavu"];
  $prijava_vlaka->izlozba_id = $uneseni_podaci["id_izlozbe"];
  $sql = "INSERT INTO prijavavlaka (vlak_id, izlozba_id) VALUES ("
  . "{$prijava_vlaka->id_vlaka}, {$prijava_vlaka->izlozba_id})";
  $this->_dnevnik_repo->kreiraj_dnevnik($this->_korisnicko_ime,
    Tip_dnevnika::RAD_S_BAZOM, $sql);
  $this->_db->zapisi_podatke($sql);
}
```

Učahurivanje se koristi radi zaštite stvarnih podataka te onemogućavanja njihove izmjene ukoliko unesen podatak nije ispravan. Unutar aplikacije ono najviše dolazi do izražaja kod kreiranja specifičnih modela. Tamo se stvarnim podacima, odnosno svojstvima klase ne pristupa direktno, već koristeći metode za dohvaćanje i postavljanje podataka. Na primjeru, model materijala izgleda ovako:

```
private $_url;
private $_vrsta_materijala;
private $_prijava_vlaka;
public function dohvati_url() {
  return $this->_url;
}
public function postavi_url($url) {
```

```
    if (empty($url)) {
      throw new Exception("Niste popunili url materijala.");
    }
    $this->_url = $url;
  }
  public function dohvati_vrsta_materijala() {
    return $this->_vrsta_materijala;
  }
  public function postavi_vrsta_materijala($vrsta_materijala) {
    if (empty($vrsta_materijala)) {
      throw new Exception("Niste popunili id vrste materijala.");
    }
    $this->_vrsta_materijala = $vrsta_materijala;
  }
```

Polimorfizam jedini nije naišao na primjenu u izrađenoj aplikaciji. Moguće bi ga bilo koristiti na primjer kod unosa i ažuriranja zapisa, pri čemu bi jedna metoda sadržavala id, a druga istoimena metoda bi bila bez njega. Ukoliko bi bio proslijeđen id radilo bi se o ažuriranju, a ukoliko ne bi bilo id-a, potrebno bi bilo kreirati novi zapis. Međutim, radi bolje preglednosti odabrano je korištenje dviju metoda različitih naziva na temelju kojih se odmah može i razaznati njihova svrha, odnosno da li se radi o kreiranju ili ažuriranju zapisa.

11.7. Refaktoriranje

Za izradu praktičnog dijela rada odabrano je refaktoriranje završnog projekta iz predmeta web dizajn i programiranje, umjesto izrade nove web stranice iz početka. Ovim pristupom bolje su upoznate mane i nedostaci nepravilnog koda, te su naučeni i primijenjeni principi pisanja čistog koda. Na prvu pomisao, proces refakturiranja web stranice zvuči jako primamljivo premda se ne mora kretati iz početka, međutim, i on ima svoje prednosti kao i mane.

Veliki nedostatak kod refakturiranja je bilo to što prvotna aplikacija nije imala dobro razrađenu strukturu te je zbog toga bila vrlo nepregledna i teška za snalaženje. Najprije ju je trebalo dobro analizirati, a potom rastaviti na dijelove i ponovno rekreirati. Aplikacija je prethodno bila razdvojena na dva dijela, prezentacijski dio koji je sadržavao Smarty predloške te drugi dio koji se sastojao od JavaScript i PHP skripti. Namjena drugog dijela je bila svakojaka, u istim metodama sadržana je logika same stranice, logika za dohvaćanje i zapisivanje podataka na bazu, prosljeđivanje podataka za prikaz predlošcima, upravljanje autorizacijom i slično. Javasript je prvenstveno korišten za upravljanje prezentacijskim slojem, primjerice za sakrivanje i otkrivanje pojedinih dijelova ekrana ovisno o ulozi korisnika. Dok su

ajax pozivi uglavnom korišteni za komunikaciju s bazom te vršenje obrada nad podacima, kako oni u redizajniranoj aplikaciji više nisu nalazi svrhu, njihov kod je uglavnom premješten uz male preinake u klase repozitorija. Smarty predlošci s druge strane su bili u redu, jedino je bilo potrebno ih premjestiti u odgovarajuće mape prema upravitelju čime je postignuta bolja preglednost i lakše pretraživanje. Također, kako se mijenjao način dohvaćanja i slanja podataka u pogled, predlošci su prilagođeni tako da koriste podatke dobivene iz novokreiranih upravitelja.

Kreiranje nove MVC strukture nije bilo pretjerano teško, ali je zahtijevalo dosta vremena i prilagodbe. Problemi su se javljali i kod promjena naziva varijabli u aplikaciji u prikladnije, međutim nazive je trebalo mijenjati na svim dijelovima aplikacije pa ukoliko se zaboravilo na jedan javljale bi se greške u radu. Također, promjena načina slanja podataka iz polja u objekte dovela je do određenih pogrešaka te tako odužila refaktoriranje.

Sam prelazak na novu MVC strukturu posjeduje i poneke nedostatke. Redizajnira aplikacija je sada na neki način zahtjevnija i kompliciranija, pošto programeri moraju posjedovati više programskog znanja i iskustva kako bi mogli koristiti pojedine elemente. Nadalje, moraju i dobro poznavati samu strukturu MVC arhitekture kako bi se jednostavnije snalazili u projektu. Mladi i neiskusni programeri, koji još nisu došli u doticaj s ovom strukturom, iz prve neće znati kako što funkcionira, već će se morati educirati te proučiti pojedine funkcionalnosti, što iziskuje vrijeme i novac.

Međutim prednosti koje su dobivene redizajnom aplikacije su neusporedivo veće. Prije svega, samo snalaženje i razumljivost koda su se drastično povećali. Točno se zna gdje se što nalazi, logika za komuniciranje s bazom implementirana je kroz klase repozitorija, logika i dodjela realnih podataka kao i upravljanje zahtjevima korisnika nalazi se kod upravitelja, reprezentacija stvarnih podataka našla je mjesto u modelima, a prikaz klijentskog sučelja koristeći HTML nalazi se unutar Smarty predložaka razvrstan prema njihovoj namjeni. Svrha i zadaća svih pojedinih dijelova aplikacije su sada veoma jasni te sadrže jedan cilj, a funkcionalnosti su implementirane prema standardima i konceptima čistoga koda. Daljnja nadogradnja aplikacije je moguća i jednostavna za implementirati.

Sljedeći primjeri pokazuju znatnu razliku između koda prijašnje aplikacije te redizajniranog koda u novoj aplikaciji:

```
function unesiTematiku($podaci) {
  global $putanja;
  $veza = new Baza();
  $veza->spojiDB();
  global $direktorij;
  $virtualnoVrijeme = strtotime(ispis_konfiguracije($direktorij));
```

```
  $time = date("Y-m-d H:i:s", $virtualnoVrijeme);
  if (isset($_SESSION["id"])) {
    $idPromjeneZapisa = $_SESSION["id"];
    unset($_SESSION['id']);
  }
  $sql = "SELECT id FROM korisnik WHERE korisnicko_ime =
    '{$_SESSION["korisnik"]}'";
  $rezultat = $veza->selectDB($sql);
  while ($red = mysqli_fetch_array($rezultat)) {
    $redovi[] = $red;
  }
  if (isset($idPromjeneZapisa)) {
    $sql = "UPDATE tematika SET"
      . " naziv = '{$podaci["nazivTematike"]}',"
      . " opis = '{$podaci["opisTematike"]}',"
      . " azurirao_korisnik_id = '{$redovi[0]["id"]}',"
      . " datum_azuriranja = '{$time}'"
      . " WHERE id = '{$idPromjeneZapisa}';";
    zapisiDnevnik($sql);
  } else {
    $sql = "INSERT INTO tematika (naziv, opis, kreirao_korisnik_id,
      datum_kreiranja) VALUES " . "('{$podaci["nazivTematike"]}',
      '{$podaci["opisTematike"]}','{$redovi[0]["id"]}','{$time}')";
    zapisiDnevnik($sql);
  }
  $veza->updateDB($sql);
  $veza->zatvoriDB();
  Echo "<script>window.location.href='$putanja/administrator/tematika
    Vlakova.php';</script>";
}
```

Navedeni primjer prikazuje unos i ažuriranje postojećih zapisa tablice tematike prije nego li je izvršeno redizajniranje. Prethodno je sva logika bila ispremiješana, a pojedine funkcije su bile toliko opširne i velike da su bile nerazumljive i vrlo nečitke. Metode redizajnirane aplikacije su kreirane prema principima čistoga koda što je vrlo jasno vidljivo na sljedećem primjeru koji obavlja istu funkcionalnost kao i gornji primjer, ali na puno pregledniji i logičniji način:

```
function azuriraj_tematiku($id_tematike) {
  $this->provjera_autorizacije_korisnika(uloga_korisnika::
    ADMINISTRATOR);
```

```
    try {
      $popis_gresaka = array("nazivTematike" => "naziv tematike",
        "opisTematike" => "opis tematike");
      $uneseni_podaci = $this->provjeri_popunjesnost_obaveznih_podataka
        ($popis_gresaka);
      if (!$uneseni_podaci) {
        return $this->prikaz_podataka_tematike($id_tematike);
      }
      $podaci_tematike = array_merge($uneseni_podaci, $this->_pripremi_
        podatke_azuriranja_tematike($id_tematike));
      $this->_odluka_o_zapisu_tematike($podaci_tematike);
      $this->preusmjeri(Postavke::dohvati_server_url() .
        "izlozbe/prikaz_administracija_izlozba/");
    } catch (Exception $e) {
      $this->pripremi_greske($e->getMessage());
      $this->prikaz_podataka_tematike($id_tematike);
    }
  }
  private function _pripremi_podatke_azuriranja_tematike($id_tematike) {
    $podaci_korisnika = $this->_korisnici_repo->dohvati_prema_korisnicko_
      ime($this->_korisnicko_ime);
    $podaci_azuriranja["id_administratora"] = $podaci_korisnika->id;
    $podaci_azuriranja["id_tematike"] = $id_tematike;
    return $podaci_azuriranja;
  }
  private function _odluka_o_zapisu_tematike($podaci_tematike) {
    if (empty($podaci_tematike["id_tematike"])) {
      $this->_tematika_repo->kreiraj($podaci_tematike);
    } else {
      $this->_tematika_repo->azuriraj($podaci_tematike);
    }
  }
```

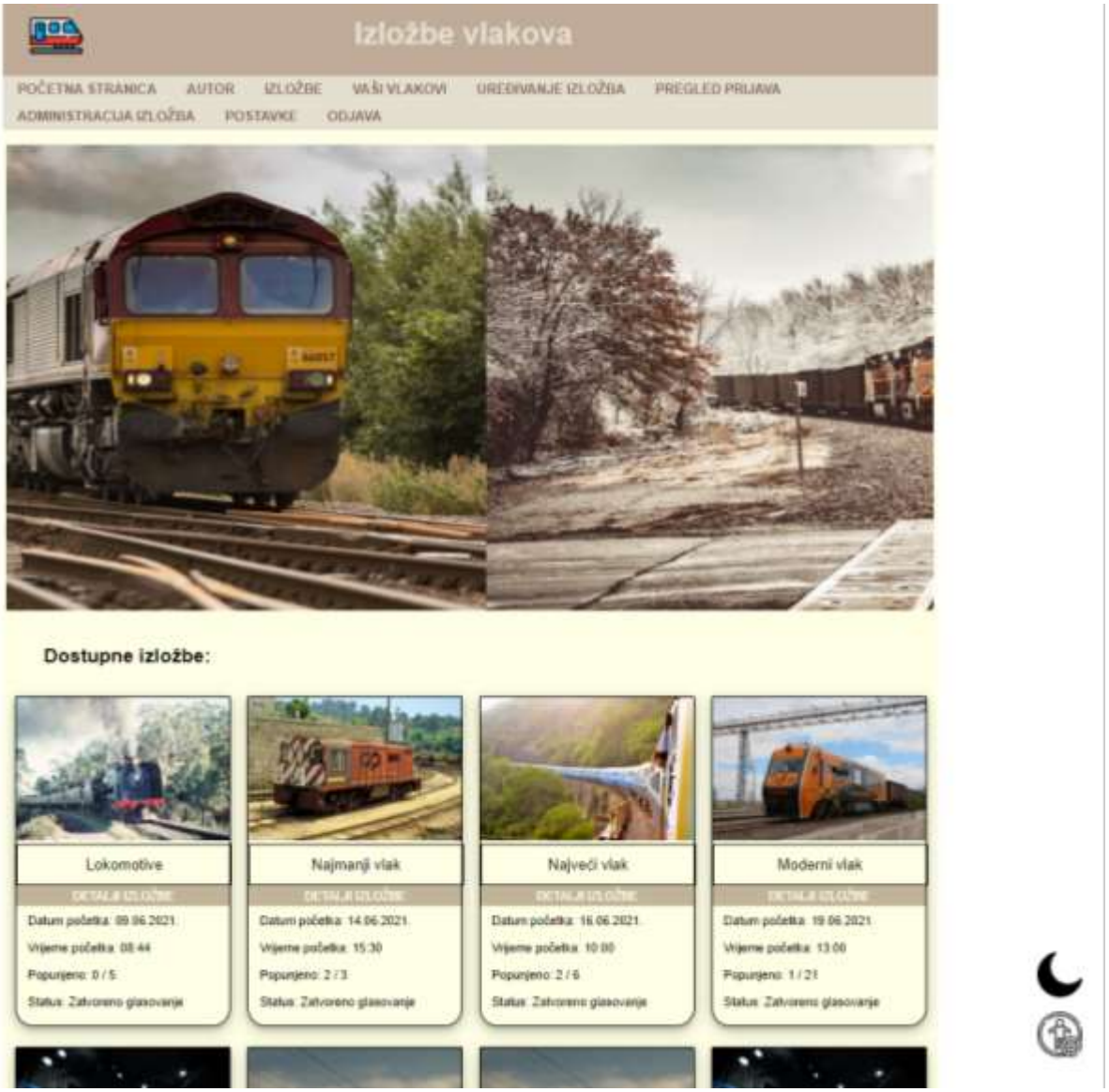
U ovom primjeru proces unosa zapisa u tablicu tematike je raspodijeljen na nekoliko osnovnih funkcionalnosti koje su definirane kroz manje privatne metode. Svrhe koje one obavljaju su: provjera ispunjenosti obaveznih polja, priprema podataka, te kreiranje ili ažuriranje zapisa ovisno da li je proslijeđen id tematike. Ovakvim strukturiranjem koda, svaka metoda posjeduje samo jednu odgovornost koja se može odrediti na temelju samog imena, kod je stoga pregledniji i razumljiviji za čitanje.

Testiranje i kreiranje jediničnih testova nije provedeno u sklopu projekta, no bilo bi poželjno. Njime bi se mogla provjeriti cijela funkcionalnost i rad aplikacije nakon redizajniranja koda, točnije, radi li kako treba i bez poteškoća. Bez provedenih testova ne može se procijeniti radi li nova aplikacija brže, te da li istovremeno ispunjava svrhu stare aplikacije.

Osim navedenog primjera u prilozima 3 i 4 su dodatno prikazani PHP programski kodovi prijašnje verzije aplikacije.

11.8. Dizajn aplikacije

Dizajn aplikacije Vlakovi može se vidjeti na sljedećem primjeru. Navedeni dizajn (prikaz zaglavlja s navigacijom i podnožja, korištenje „kartica“ za prikaz liste podataka, nijanse boje, prikaz slika) se koristi kroz cijelu aplikaciju, uz male modifikacije.



Slika 2 Prikaz web aplikacije Vlakovi, stranica o izložbama vlakova [autorski rad].

12. Zaključak

Korištenje principa čistog koda uvelike olakšava programerima sam razvoj, nadogradnju i uvođenje novih funkcionalnosti te održavanje aplikacija. Pisanje čistog koda počinje odabirom prikladnih i smislenih imena svakog pojedinog elementa aplikacije. Na taj način poboljšava se razumljivost samog koda. Iz naziva metoda, funkcija i svojstava moguće je raspoznati njihovu namjenu i svrhu čime upotreba komentara postaje suvišna. Također, kod čistog programiranja funkcije i metode bi trebale biti što kraće te imati jasno određen i definiran samo jedan cilj.

Klase čistih kodova su također male i jasne. Preporuča se korištenje većeg broja manjih klasa koje su lijepo raspoređene te svaka pojedinačna klasa sadrži svoj vlastiti cilj. Velike klase koje posjeduju veliki broj inicijalizacija svojstva i metoda, sadrže najvjerojatnije veći broj ciljeva te bi se takve klase trebale raspodijeliti na manje pojedinačne klase.

Čisti kodovi se čitaju poput romana, gdje je početak programa kratak i jasan, s jednostavnim i kratkim imenima funkcija, metoda, svojstva i varijabla. Čitanjem sve dubljih i dubljih dijelova programa, metode i funkcije postaju zahtjevnije i kompleksnije, a razna imena postaju sve opširnija, no ne i nerazumljiva.

Izrazito bitno je i upravljanje pogreškama, dohvaćanjem i obradom iznimaka povećava se kvaliteta samog koda. Prednosti korištenja iznimaka su mogućnost razdvajanja dobrog koda od koda koji je namijenjen obradi pogrešaka te brzo i jednostavno pronalaženje pogrešaka kao i njihovo ispravljanje.

Čisti kod temelji se na objektno orijentiranom programiranju čijom upotrebom se omogućuje komunikacija raznih dijelova aplikacije putem objekata. Okosnicu objektno orijentiranog programiranja čine četiri glavna koncepta, a to su: apstrakcija, nasljeđivanje, učahurivanje i polimorfizam. Apstrakcija se koristi kod klasa koje posjeduju iste odgovornosti ali različite načine implementacije, na taj način koristi se apstrakcija sučelja umjesto korištenja konkretne klase. Kako bi zaštitili stvarne podatke primjenjuje se metoda učahurivanja, podacima se tako ne pristupa direktno već preko metoda za njihovo dohvaćanje i postavljanje. Nasljeđivanjem se omogućuje klasama da posjeduju zajednička svojstva i metode te se tako izbjegava potreba za uporabom dupliciranog koda. Za metode s istom funkcionalnošću ali drugačijim skupom ulaznih podataka koristi se polimorfizam koji omogućuje da se kreiraju istoimene metode, koje će se vršiti ovisno o proslijeđenim varijablama.

Refaktoriranje je proces koji postojeću aplikaciju optimizira ili modificira kako bi se postigle bolje performanse programa ili povećala kvaliteta samoga koda. To je zahtjevan proces i stoga se mora provoditi u prihvatljivo vrijeme uz prethodnu detaljnu analizu koda kako

ne bi negativno utjecao na sam rad aplikacije. Refaktoriranjem se uklanjaju nepravilni dijelovi koda kao što su duplicirani kodovi, velike klase i metode koje posjeduju raznolike odgovornosti, nepotrebni komentari i slično. Na taj način kod postaje pregledan i razumljiv, te najbitnije od svega omogućena je lakša daljnja nadogradnja i održavanje programskog rješenja.

Popis literature

[1] L. Team, »What Is Bad Code, Why It Is Dangerous, and How to Write Good Code,« Lvivity, 29 05 2021. [Mrežno]. Available: https://lvivity.com/how-to-write-good-code. [Pokušaj pristupa 27 06 2021].

[2] D. Karanth, »3 Reasons Why People Write Insanely Bad Code,« Software Yoga, [Mrežno]. Available: https://www.softwareyoga.com/3-reasons-people-write-insanely-bad-code/. [Pokušaj pristupa 28 06 2021].

[3] R. C. Martin, Clean code A Handbook of Agile Software Craftsmanship, Boston: Pearson Education, 2009.

[4] G. Woodfine, »What is Clean Code ?,« Threenine, 08 10 2018. [Mrežno]. Available: https://garywoodfine.com/what-is-clean-code/. [Pokušaj pristupa 29 06 2021].

[5] Z. Kalafatić, Skriptni jezici Materijali za predavanja, Zagreb, 2012.

[6] F. Miha, »How to Better Name Your Functions and Variables,« Medium, 17 07 2020. [Mrežno]. Available: https://medium.com/swlh/how-to-better-name-your-functions-and-variables-e962a4ef335b. [Pokušaj pristupa 29 06 2021].

[7] S. Sultan, »What is Bad Code? How to Write Clean Code?,« Medium, 29 11 2020. [Mrežno]. Available: https://sunnysultan1640.medium.com/what-is-bad-code-how-to-write-clean-code-a9b7b539ad8. [Pokušaj pristupa 26 06 2021].

[8] S. R. Dipta, »Clean Code: Naming,« Betterprogramming, 06 05 2020. [Mrežno]. Available: https://betterprogramming.pub/clean-code-naming-b90740cbae12. [Pokušaj pristupa 29 06 2021].

[9] A. Lovrenčić i M. Konecki, Programiranje u 14 lekcija C++, Varaždin: Mini-print-logo d.o.o., 2017.

[10] E. W. Dijkstra, A Method of Programming, Austin: Addison-Wesley, 1988.

[11] P. Brođanac, Java i objektno orijentirano programiranje, Knjiga iz informatike za 4. razred prirodoslovno-matematičke gimnazije, Zagreb.

[12] N. Schäferhoff, »Code Formatting and Code Comments – A Beginner's Guide to Do It Right,« Torquemag, 17 02 2019. [Mrežno]. Available: https://torquemag.io/2019/07/code-formatting-guide/. [Pokušaj pristupa 12 07 2021].

[13] S. Srinidhi, »OOPs and Clean Code Part 1: Principles of Object Oriented Programming,« Geektrust, 26 03 2021. [Mrežno]. Available: https://www.geektrust.in/blog/2021/03/26/oops-and-clean-code-part-1/. [Pokušaj pristupa 27 06 2021].

[14] D. Phillips, Python 3 Object Oriented Programming, Birmingham: Packt Publishing, 2010.

[15] M. Fowler, Refactoring Improving the Design of Existing Code, Addison-Wesley, 2002.

[16] CodeIgniter, »PHP Style Guide,« British Columbia Institute of Technology, 19 09 2019. [Mrežno]. Available: https://codeigniter.com/userguide3/general/styleguide.html. [Pokušaj pristupa 14 08 2021].

Popis slika

Prilog 1 – Prikaz korisničkih uloga aplikacije Vlakovi

Vlakovi

Kratak opis projekta:
Sustav služi za upravljanje izložbom vlakova.
Uloge: Neregistrirani korisnik, Registrirani korisnik, Moderator, Administrator

Detaljne upute:

Administrator

- Pregledava/kreira/ažurira tematike izložbi vlakova (Lokomotive, Motorni vlakovi, Vlakovi velikih brzina, ...) i dodjeljuje moderatore. Jedan moderator može upravljati s više tematika izložba vlakova, a jedna tematika izložba vlakova može imati više moderatora.
- Može napraviti sigurnosnu kopiju (eng. backup) svih vlakova i materijala iz baze u obliku SQL skripte. Ne radi se sigurnosna kopija datoteka.
- Može vratiti podatke iz sigurnosne kopije pri čemu se brišu trenutni podaci u bazi. Za svaki vlak automatski se provjerava da li postoje fizičke datoteke materijala koje su postavljene od strane korisnika i ako ne šalje se zahtjev na e-mail korisniku da se ponovo postave.
- Administrator vidi statistiku korištenja sustava pristupa stranicama (min 5), uz mogućnost filtriranja po korisnicima i vremenskom razdoblju (od – do).

Moderator

- Pregledava/kreira/ažurira izložbe za tematike za koje je zadužen. Kod kreiranja definira datum početka izložbe i broj korisnika koji se mogu prijaviti na izložbu. Prijave nije moguće poslati nakon datuma početka izložbe. Ažuriranje je moguće do početka izložbe. Izložba može imati status: otvorene prijave, izložba u tijeku, otvoreno glasovanje i zatvoreno glasovanje.
- Za svaku izložbu zatvara/otvara glasovanje. Pobjednik se automatski određuje nakon zatvaranja glasovanja.
- Vidi popis prijave vlakova za izložbu te potvrđuje/odbija prijave. Posebno su označene prijave koje nisu obrađene.

Registrirani korisnik

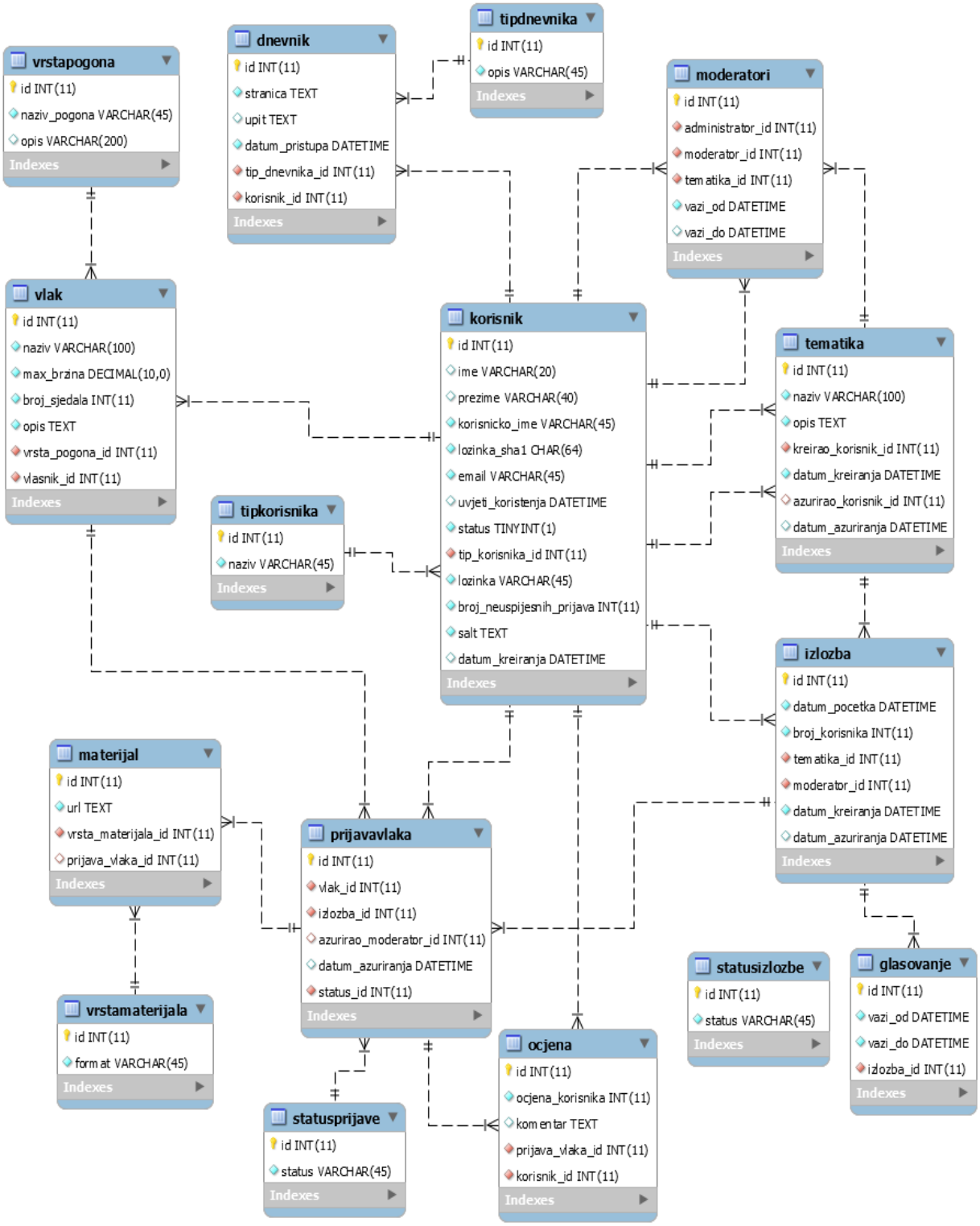
- Može pregledavati i poslati prijavu za dodavanje vlaka na izložbu. U prijavi se odabire izložbu i unosi naziv vlaka, pogon vlaka (npr. dizel, električni, parni, ...), maksimalna brzina, broj sjedala i opis. Prijavu je moguće otkazati sve do datuma početka.
- Vidi popis prijavljenih vlakova po otvorenoj izložbi i može glasati za najbolji vlak ako glasovanje otvoreno. Dopušten je najviše jedan glas po izložbi vlakova.
- Može postaviti materijale za vlak pri čemu bira vrstu materijala (slika, video, audio, ...). Ne može dodati materijale sve dok moderator ne prihvati prijavu.
- Vidi galeriju izložbe vlakova. Može filtrirati po pogodnu vlaka i definirati koliko materijala se prikazuje po stranici.

Neregistrirani korisnik

- Na početnoj stranici prikazana je statistika broja vlakova po izložbi vlakova.
- Vidi popis prijavljenih korisnika grupirano po izložbama i označenim pobjednikom.
- Odabirom pobjednika može vidjeti detalje pobjednika i sve materijale pobjednika.
- Putem RSS kanala može se pratiti zadnjih 10 potvrđenih prijava po svakoj izložbi.

Slika 3. Prikaz korisničkih uloga aplikacije Vlakovi.

Prilog 2 – ERA model aplikacije Vlakovi



Slika 4. Prikaz ERA modela aplikacije Vlakovi [autorski rad].

Prilog 3 – Prikaz PHP programskog koda stare verzije aplikacije kod prijave korisnika

```
function ulogirajKorisnika($podaci) {
  global $greska;
  global $virtualniDatumVrijeme;
  $sql = "SELECT * FROM korisnik WHERE  korisnicko_ime
    = '{$podaci["korime"]}'";
  $rezultat = dohvatiPodatke($sql);
  if (!empty($rezultat[0]["korisnicko_ime"])) {
    if ($rezultat[0]["datum_kreiranja"] != "") {
    $lozinka = hash("sha256", $podaci["lozinka"]. $rezultat[0]["salt"]);
      if ($rezultat[0]["lozinka_sha1"] == $lozinka and $rezultat[0]
      ["broj_neuspijesnih_prijava"] <= 3) {
        $autenticiran = true;
        Sesija::kreirajKorisnika(
          $rezultat[0]["korisnicko_ime"], $virtualniDatumVrijeme,
          $rezultat[0]["tip_korisnika_id"], "disabled", "disabled");
        resetirajBrojPokusaja($rezultat);
        zapisiDnevnik($rezultat);
        provjeraKorisnika();
      } elseif ($rezultat[0]["broj_neuspijesnih_prijava"] >= 3) {
        $greska .= "Korisniči račun {$rezultat[0]["korisnicko_ime"]} Vam
          je blokiran. Molimo Vas kontaktirajte administratora";
      } else {
        povecajBrojPokusaja($rezultat);
      }
    } else {
      $greska .= "Niste aktivirali svoj račun preko aktivacijskog koda";
    }
  } else {
    $greska .= "Nepostojeće korisnićko ime";
  }
}
```

Prilog 4 – Prikaz PHP programskog koda stare verzije aplikacije kod dodavanja vlakova korisnika

```
function dodavanjeMaterijala() {
  if (isset($_POST["dodavanjeNovogVlaka"])) {
    global $greska;
    $podaciObrasca = provjeriPodatke();
    if (empty($greska)) {
      if ($podaciObrasca["noviNazivPogona"] == 'nijePopunjeno' and
      $podaciObrasca["noviOpisPogona"] == 'nijePopunjeno') {
        dodajNoviVlak($podaciObrasca);
      } else {
        dodajNoviVlakSaNovimPogonom($podaciObrasca);
}}}}
function dodajNoviVlak($podaciObrasca) {
  global $putanja;
  $id = dohvatiIdKorisnika();
  if (isset($_SESSION["id"])) {
    $sql = "UPDATE vlak SET"
    . " naziv = '{$podaciObrasca["nazivVlaka"]}',"
    . " max_brzina = {$podaciObrasca["maxBrzina"]},"
    . " broj_sjedala = {$podaciObrasca["brojSjedala"]},"
    . " opis = '{$podaciObrasca["opisVlaka"]}',"
    . " vrsta_pogona_id = {$podaciObrasca["vrstaPognona"]},"
    . " vlasnik_id = {$id} WHERE id = {$_SESSION["id"]}";
    zapisiDnevnik($sql);
    } else {
      $sql = "INSERT INTO vlak (naziv, max_brzina, broj_sjedala, opis,
      vrsta_pogona_id, vlasnik_id) VALUES ('{$podaciObrasca
        ["nazivVlaka"]}', {$podaciObrasca["maxBrzina"]},
        {$podaciObrasca["brojSjedala"]},'{$podaciObrasca["opisVlaka"]}',
        {$podaciObrasca["vrstaPognona"]}, {$id})";
      zapisiDnevnik($sql);
    }
  zapisiPodatke($sql);
  echo "<script>window.location.href='$putanja/korisnik/prikazVlakova.
    php';</script>";
}
```