Oblikovni obrasci i primjena u razvoju mobilnih aplikacija na Android platformi

Pleša, Jurica

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

OBLIKOVNI OBRASCI I PRIMJENA U RAZVOJU MOBILNIH APLIKACIJA NA ANDROID PLATFORMI

Diplomski rad

Jurica Pleša

Osijek, 2016.

SADRŽAJ:

1. UVOD

Oblikovni obrasci su općeniti načini za rješavanje čestih problema pri dizajniranju programske podrške. Oni se ne mogu direktno postaviti u kod programa, već su to smjernice prema kojima programer prilikom razvoja korištenjem objektno orijentirane paradigme osmišljava rješenje problema. Oblikovni obrasci su bitni zbog znanja da se problem rješava na najbolji način i da neće doći do naknadnih problema. Prvi puta se pojavljuju u građevinarstvu, a u informatici se dokumentiraju krajem dvadesetog stoljeća gdje su i danas iznimno važni za izradu programske podrške. Oblikovni obrasci se koriste i pri stvaranju Android mobilnih aplikacija gdje su neophodni za ispravan rad aplikacije.

Cilj diplomskog rada je prikazivanje oblikovnih obrazaca, opisivanje njihove uloge i načina korištenja. Osim oblikovnih obrazaca, cilj je prikazati i načela u objektno orijentiranom dizajnu i anti-obrasce. Načela su temeljne odredbe koje treba imati na umu pri pisanju koda, a anti-obrasci, suprotno od oblikovnih obrazaca, opisuju česte krive načine dizajniranja programske podrške. Potrebno je i prikazati oblikovne obrasce prikladne za razvoj Android aplikacija. Isto se namjerava ostvariti izradom Android mobilne aplikacije koja korisniku aplikacije omogućava pregled lokacija u pojedinom gradu koje imaju povijesnu priču.

Drugo poglavlje ovog rada posvećeno je razlozima postojanja oblikovnih obrazaca, vrstama i kratkim pojedinačnim opisima obrazaca. Uz to, govori se i o anti-obrascima, odnosno primjerima lošeg rješavanja problema i načelima u objektno orijentiranom dizajnu. U trećem poglavlju riječ je o Android operacijskom sustavu i njegovom načinu funkcioniranja. Opisani su i primjeri korištenja najčešćih oblikovnih obrazaca pri stvaranju Android mobilnih aplikacija. Četvrto poglavlje prikazuje Android mobilnu aplikaciju izrađenu u sklopu diplomskog rada. Analizirani su korisnički zahtjevi za aplikaciju i opisan je dizajn, funkcionalnost i način na koji su izrađeni zahtjevi. Također, dan je opis rješenja s dijagramom toka i način rada aplikacije potkrijepljen prikazom aplikacije u radu. Posljednje poglavlje donosi zaključak kao i smjernice za budući rad.

2. OBLIKOVNI OBRASCI U RAZVOJU PROGRAMSKE PODRŠKE

Objektno orijentirano programiranje (OOP) jedna je od paradigmi programiranja koja je bazirana na objektima i interakciji među njima. Svaki objekt stvoren je prema klasi kojom su opisane njegove karakteristike. Točnije, klasom se definiraju atributi i metode koje će sadržavati objekt stvoren prema toj klasi. Prema [1], glavne značajke OOP-a su enkapsulacija objekata, nasljeđivanje i polimorfizam. Enkapsulacija prikriva atribute i metode objekta kako bi se povećala sigurnost i ispravnost objekta. Pri stvaranju nove klase moguće je naslijediti drugu klasu. Tada će nova klasa preuzeti atribute i metode koje su definirane u osnovnoj klasi. Nasljeđivanjem je omogućen i polimorfizam koji predstavlja mogućnost tretiranja objekta izvedene klase kao objekta osnovne klase i sposobnost promjene metode osnovne klase kako bi u objektu izvedene klase ta metoda odrađivala drugačiji zadatak.

Dizajniranje objektno orijentirane programske podrške je zahtjevno i zbog brojnih mogućnosti teško je odlučiti se za način kojim će se riješiti problem. Oblikovni obrasci su recepti za rješenja problema koji se često pojavljuju pri razvoju programa. Oni olakšavaju ponovno korištenje uspješnih arhitektura i pomoću jasnih propisa unaprjeđuju upravljanje klasama i interakciju među objektima.

2.1. Potreba za oblikovnim obrascima

Čestim nailaskom na problem u životu, čovjek pronalazi najbolje rješenje za njega i nastavlja ga koristiti jer tada sa sigurnošću zna da će problem biti uspješno riješen. Jednako je i pri dizajniranju objektno orijentirane programske podrške, zbog potrebe za uspješnim rješavanjem čestih problema stvorila se i potreba programera za oblikovnim obrascima. Osim sigurnosti u svladavanje problema, oblikovni obrasci programerima omogućuju i druge pogodnosti [2]. Kod se ne piše samo za programski prevoditelj (engl. *compiler*) već i za ostale programere. Pojam „ostali programeri“ podrazumijeva tim programera koji trenutno radi na projektu, tim programera koji će u budućnosti raditi sa programskom podrškom koja se piše što uključuje i programera koji je napisao kod. Kod napisan uz uporabu oblikovnih obrazaca je potpuno jasan i neće zbuniti drugog programera pa se može reći da se oblikovnim obrascima pospješuje komunikacija među programerima. Komunikacija se poboljašava i time što neće biti potrebno objašnjavati zašto se problem riješio baš na taj način jer se zna da je oblikovni obrazac kvalitetan. Također, oblikovni obrasci su jednaki za sve objektno orijentirane programske jezike, što znači da oblikovni obrazac Prilagodnik označava jednaku

stvar u Java-i, C#-u i u Python-u. Kada programer rješava problem na svoj način, drugi programeri će biti skeptični oko njegovog rješenja, teže će ga razumjeti i stvoriti će prema njemu otpor. Korištenje oblikovnih obrazaca uklanja skeptičnost i stvara sigurnost da u budućnosti neće biti problema oko rješenja. Osobina sigurnosti oblikovnih obrazaca dolazi zbog činjenice da su prijašnji programeri uklonili nedostatke oblikovnih obrazaca i osigurali da zbog takvog rješenja kasnije neće doći do dodatnih problema.

U [3] je opisano da su oblikovni obrasci nastali prije četrdesetak godina kada je građevinski inženjer Christopher Alexander primijetio da ponavlja načine na koji rješava probleme pri gradnji „građevina“. Zatim je dokumentirao svoje znanje i vještine kako bi ostali građevinski inženjeri mogli koristiti njegove obrasce za rješavanje problema. Oblikovni obrasci za razvoj programske podrške dokumentirani su 1995. godine kada su Eric Gamma, Richard Helm, Ralph Johnson i John M. Vlissides napisali knjigu u kojoj su opisali najčešće korištene obrasce, njih dvadeset i tri. Danas je dokumentirano puno više oblikovnih obrazaca, ali dvadeset i tri obrasca spomenutih autora pod grupnim nazivom Družba četvorice (engl. *Gang of four*) smatraju se početnim i najpoznatijim obrascima.

Oblikovni obrasci za razvoj programske podrške uglavnom se daju u obliku kataloga. U katalogu su obrasci podijeljeni u tri glavne skupine. Svaki obrazac ima svoje ime koje jasno označava njegovu upotrebu i objašnjenje na koji način obrazac rješava konkretan problem. Katalog tako omogućuje programeru brz pregled svih oblikovnih obrazaca kako bi lako pronašao obrazac koji mu je potreban za rješavanje problema.

2.2. Vrste oblikovnih obrazaca

Oblikovni obrasci podijeljeni su u tri vrste: obrasci stvaranja (engl. *Creational Patterns*), strukturni obrasci (engl. *Structural Patterns*) i obrasci ponašanja (engl. Behavioral Patterns).

2.2.1. Obrasci stvaranja

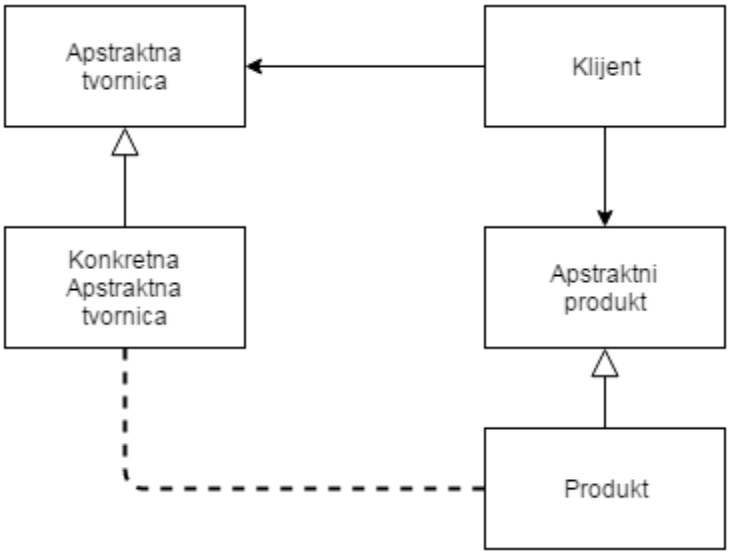
Obrasci stvaranja odvajaju proces instanciranja od ostatka koda [4]. Cilj je pomoću obrazaca stvaranja stvoriti sustav koji je neovisan o načinu na koji se objekti stvaraju. Stvaranjem kompleksnog sustava dolazi se do problema da sustav sve više postaje ovisan o kompoziciji objekata. Pomoću oblikovnih obrazaca problem se rješava na način da se enkapsulira znanje sustava koju konkretno klasu koristi i prikriva se način na koji se objekti stvaraju. Sve što će sustav znati o objektu je njegovo prethodno definirano sučelje. Također, obrasci stvaranja daju fleksibilnost programeru u tome što se stvara, tko stvara, kako i gdje se stvara. Postoji pet

oblikovnih obrazaca, a to su: apstraktna tvornica, graditelj, metoda-tvornica, prototip i jedinstveni objekt.

Apstraktna tvornica

Apstraktna tvornica (engl. *Abstract* Factory) je oblikovni obrazac stvaranja koji omogućava sučelje za stvaranje srodnih objekata bez definiranja konkretnih klasa kojima su opisani. Apstraktna tvornica je prema tome apstraktna klasa od koje korisnik traži instancu objekta. Apstraktna tvornica korisniku predaje objekt sa svojstvima koje je korisnik zatražio bez da korisnika informira o tome prema kojoj konkretno klasi je objekt stvoren. Apstraktna tvornica se stvara samo za jednu obitelj objekata, odnosno nije moguće napraviti apstraktnu tvornicu za sve objekte, već samo za srodne.

Oblikovni obrazac Apstraktna tvornica koristi se kada je potrebno odvojiti stvaranje objekata od ostatka koda, odnosno kada se želi da stvaranje, predstavljanje i komponiranje objekata bude nezavisni sustav. Zatim, koristi se kada u sustavu postoji više srodnih objekata i/ili kada se želi prisiliti zajedničko korištenje nekoliko srodnih objekata. Na slici 2.1, koja je izrađena prema [4], prikazana je struktura Apstraktne tvornice. Klijent (engl. *Client*) zahtjeva određeni objekt od Apstraktne tvornice. Apstraktna tvornica koristi konkretnu klasu tvornice (engl. *Concrete Factory*) kojom stvara objekt, odnosno produkt (engl. *Product*) koji se enkapsulira kao apstraktni produkt (engl. *AbstractProduct*) i nudi klijentu na korištenje.



Slika 2.1. Struktura Apstraktne tvornice

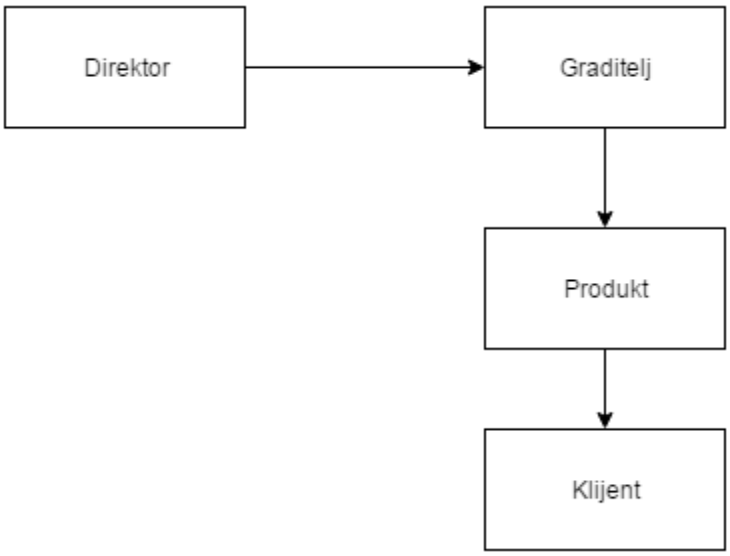
Prednost Apstraktne tvornice je i laka mogućnost promjene konkretne klase. Razlog tomu je što promjena konkretne klase neće utjecati na Apstraktnu tvornicu ukoliko je konkretna klasa već dio tvornice [4]. S druge strane, teže je dodavanje novih objekata koje bi tvornica mogla

stvarati. Zbog toga što je potrebno proširiti sučelje na način da se promjeni i Apstraktna tvornica i sve podklase koje tvornica koristi.

Graditelj

Svrha oblikovnog obrasca stvaranja Graditelja (engl. *Builder*) je odvajanje stvaranja složenog objekta od njegovog prikaza tako da jednaki proces stvaranja može stvoriti objekte s različitim prikazom [6]. Tako Graditelj odvaja algoritme i funkcionalnost objekta od njegovog formata kojim će biti prikazan. Graditelj se koristi kada se mora omogućiti da se pri procesu stvaranja može promijeniti prikaz objekta i/ili kada se želi da algoritam za stvaranje složenih objekata bude odvojen i samostalan od dijelova koje čine objekt.

Slikom 2.2, izrađenom prema [4], prikazani su sudionici uz Graditelja. Klijent stvara objekt Direktor (eng. *Director*) i postavlja ga prema željenim postavkama pomoću Graditelja. Direktor zatim obaviještava Graditelja o postavkama prema kojima Graditelj stvara objekt i predaje ga klijentu.

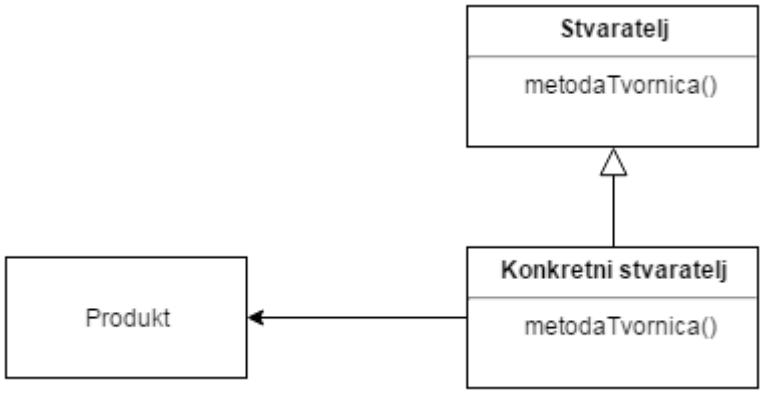


Slika 2.2. Struktura Graditelja

Rezultat korištenja oblikovnog obrasca Graditelja je omogućavanje različitih oblika prikaza objekta. Graditelj enkapsulira kod vezan uz konstrukciju i prikaz objekta, pa tako klijent ne mora znati ništa o unutarnjoj strukturi klase već samo određuje vanjski prikaz. Također je i olakšana konstrukcija procesa jer Graditelj omogućuje bolju kontrolu cijelog procesa. Dodatno, Graditelj rješava problem „teleskopskih“ konstruktora (engl. *Telescoping Constructor*), tj. uklanja potrebu za stvaranjem velikog broja konstruktora s različitim brojem parametara.

Metoda-tvornica

Metoda-tvornica (engl. *Factory* Method) je sučelje koje određuje prema kojoj podklasi će objekt biti stvoren. Ona omogućava da nadređena klasa dozvoli podklasama da odaberu koja je podklasa potrebna i na kraju klijentu predaje objekt koji je zatražio. Oblikovni obrazac stvaranja Metoda-tvornica koristi se kada klasa nije u mogućnosti saznati prema kojoj klasi treba stvoriti objekt ili kada je potrebno da podklase odluče koji tip objekta se točno traži. Prema strukturi prikazanoj na slici 2.3, izrađenoj prema [4], postoji Stvaratelj (eng. *Creator*) koji sadrži Metodu-tvornicu i podklasu Konkretni stvaratelj (eng. *Concrete Creator*). Stvaratelj dozvoljava svojoj podklasi Konkretnom stvaratelju da stvori željeni objekt Produkt.



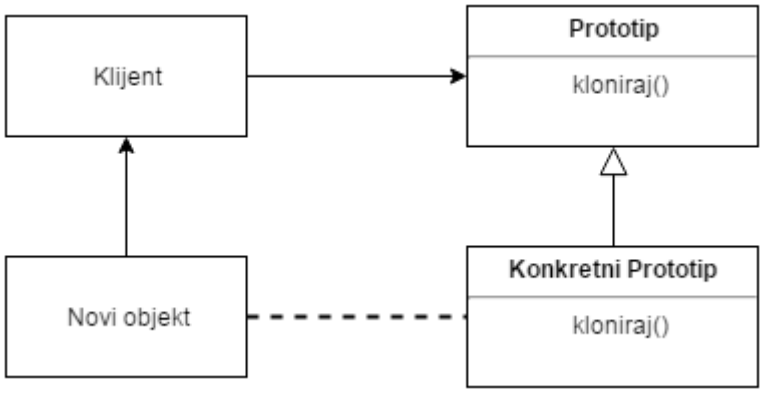
Slika 2.3. Struktura Metode-tvornice

Negativna strana korištenja Metode-tvornice je potreba za stvaranjem podklasa. Ukoliko je aplikacija već dizajnirana tako da postoje podklase, onda Metoda-tvornica postaje prednost jer će omogućiti fleksibilnost pri kreiranju objekata. Također, metoda-tvornica povezuje klase koje su paralelne u hijerarhiji, odnosno u istoj razini.

Prototip

Oblikovni obrazac Prototip (engl. *Prototype*) je gotov objekt koji ima mogućnost kopiranja samog sebe kako bi stvorio nove objekte. Stvaranje Prototipa je slično nasljeđivanju klasa, ali nije potrebno pisati kod za svaku podklasu već se pri stvaranju novog objekta korištenjem Prototipa određuje kako će izgledati taj objekt.

Prema [4], glavni razlog korištenja Prototipa je želja da stvaranje i prikaz objekata budu samostalani i odvojeni činovi. Osim toga, Prototip se može koristiti kako bi se smanjilo pisanje brojnih klasa i tako umanjila hijerarhija klasa na jednoj razini. Rad s Prototipom se odvija tako da klijent zatraži novi objekt, odnosno zatraži od Prototipa da se klonira. Prototip odradi kloniranje i novi objekt vrati klijentu kao što je prikazano na slici 2.4, izrađenoj prema [4].



Slika 2.4. Struktura Prototipa

Prototip, kao i Graditelj i Apstraktna tvornica, sakriva konkretnu klasu prema kojoj je stvoren čime smanjuje količinu informacija kojima klijent raspolaže. Osim toga, omogućuje dodavanje i brisanje stvorenih objekata prema Prototipu dok se program izvršava, smanjuje potrebu za nasljeđivanjem i tako smanjuje hijerarhiju, te omogućuje definiranje novih vrijednosti ili dodavanje novih dijelova objekta bez stvaranja nove klase za takav objekt. Nedostatak ovog oblikovnog obrasca je da svaka podklasa Prototipa mora sadržavati operaciju kloniranja.

Jedinstveni objekt

Cilj posljednjeg oblikovnog obrasca stvaranja, Jedinstvenog objekta (engl. *Singleton*), je osiguravanje da klasa ima mogućnost stvoriti samo jedan objekt i omogućiti globalni pristup tom objektu. Takva klasa će pratiti svoj jedinstveni objekt i neće dozvoliti stvaranje drugih objekata. Jedinstveni objekt se koristi kada je potrebno da postoji samo jedan objekt klase i da mu se može pristupiti sa određenih pristupnih točaka. Struktura Jedinstvenog objekta je jednostavna, sastoji se jedino od same klase koja pruža Jedinstveni objekt i omogućuje drugim klijentima pristup tom objektu.

Osim navedenih mogućnosti, dobra strana Jedinstvenog objekta je i velika kontrola prema objektu. Može se određivati kada i kako će klijent moći pristupiti objektu. Lako je i promijeniti klasu koja stvara Jedinstveni objekt da može stvoriti više od jednog objekta, odnosno moguće je odrediti točan broj stvorenih objekata. Prema [7], negativna strana Jedinstvenog objekta je nemogućnost testiranja izoliranih podataka. Jedinstveni objekt treba koristiti za javne biblioteke kada se želi osigurati stvaranje samo jednog objekta, ali ukoliko projekt ostaje unutar tima bolje ga je izbjegavati i jednostavno stvoriti samo jedan objekt.

2.2.2. Strukturni obrasci

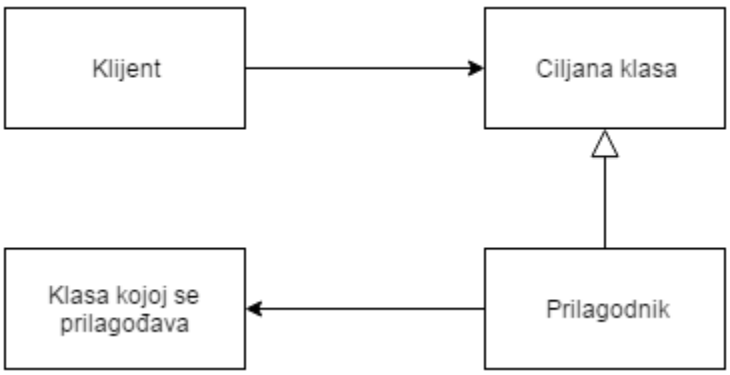
Prema [4], strukturni obrasci opisuju načine sastavljanja objekta i klasa kako bi se stvorile veće strukture i dijele se na Strukturne klasne obrasce i Strukturne objektne obrasce. Strukturni klasni obrasci koriste nasljeđivanje za stvaranje sučelja ili implementacija. Oni omogućavaju povezivanje odvojeno razvijenih klasa kako bi mogle raditi zajedno. Strukturni objektni obrasci imaju mogućnost rada nad objektima i mijenjanja njihove kompozicije dok je program u izvođenju. To svojstvo omogućava komponiranje objekata čime se stvara nova funkcionalnost. Najčešće korišteni strukturni obrasci su: Prilagodnik, Most, Kompozit, Dekorator, Fasada, *Flyweight* i *Proxy*.

Prilagodnik

Strukturni obrazac Prilagodnik (engl. Adapter) izmjenjuje sučelje klase u sučelje koje klijent očekuje. On omogućuje zajednički rad klasa koje u protivnom ne bi mogle zajedno raditi zbog neadekvatnog izgleda sučelja. Postoje Prilagodnici za klase i Prilagodnici za objekte, kako je objašnjeno u [5]. Prilagodnik za klase koristi nasljeđivanje i može prilagođavati samo klase, a ne može se koristiti za sučelja. Prilagodnik za objekte sastavlja željene dijelove u novu cjelinu i tako se može koristiti i za klase i za sučelja.

Prilagodnik se koristi kada se želi koristiti već postojeću klasu, ali njezino sučelje ne odgovara sustavu. Koristi se i za omogućavanje da dvije klase funkcioniraju zajedno kada po definiciji nemaju tu mogućnost. Prilagodnikom se stvara veza među njima kako bi klase mogle komunicirati i raditi zajedno.

Strukture za Prilagodnik za klase i Prilagodnik za objekte su gotovo jednake. Klijent zatražuje povezivanje ciljane klase (eng. *Target*) sa klasom kojoj se treba prilagoditi (eng. *Adaptee*). Prilagodnik prima zahtjev, prilagođava ga i komunicira dalje s drugom klasom (Sl.2.5, slično kao u [4]).



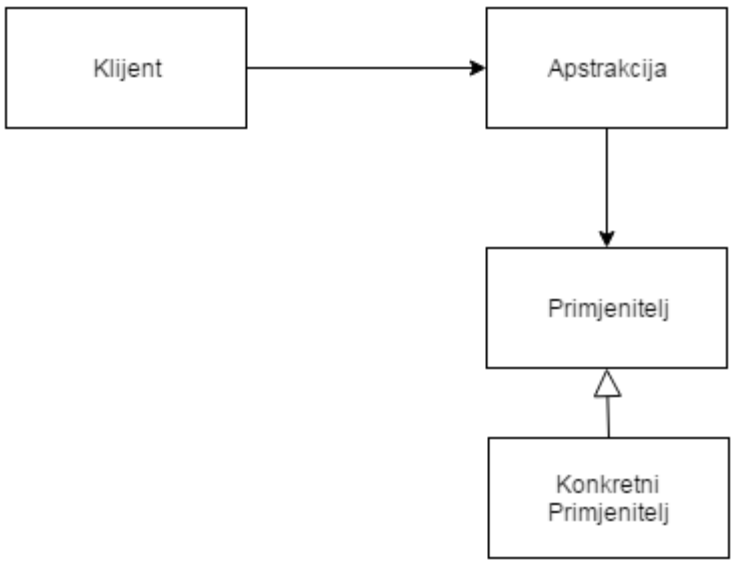
Slika 2.5. Struktura Prilagodnika

Klasni Prilagodnik prilagođava se za konkretno jednu klasu, što znači da on neće raditi i za podređene klase. Pošto je klasni Prilagodnik podklasa klase koja se prilagođava, Prilagodniku je omogućena promjena ponašanja nadređene klase, odnosno klase koju se prilagođava. S druge strane, objektni Prilagodnik može raditi s više klasa. Može raditi s nadređenom klasom i sa svim podklasama. Pri tome je otežana promjena ponašanja nadređene klase.

Most

Most (engl. *Bridge*) je strukturni oblikovni obrazac koji odvaja klasu od već stvorenih objekata kako bi se mogli mijenjati samostalno. Oblikovni obrazac Most osim glavne uporabe koristi se i kada se želi da izmjene nad klasom nemaju utjecaj na stvorene objekte [4]. Tako neće biti potrebno ponovno prevoditi kod već će sljedeći objekt biti stvoren prema izmjenama, a stari objekti ostati jednaki kao i prije. Most se koristi i kako bi klasa i stvoreni objekti mogli biti proširivani nasljeđivanjem, odnosno omogućava spajanje različitih klasa i stvorenih objekata i njihovo samostalno proširivanje.

Struktura obrasca Most prikazana je na slici 2.6, izrađenoj prema [4], i sastoji se od Apstrakcije (engl. *Abstraction*), Primijenitelja (engl. *Implementor*) i Konkretnog primijenitelja (engl. *Concrete Implementor*). Klijent predaje Apstrakciji željenu promjenu, a Apstrakcija predaje taj zahtjev Primijenitelju koji obavlja zahtijevanu izmjenu. Obrazac svojim razdvajanjem apstrakcije od implementacije stvara „štit“ kojim čuva detalje o implementaciji objekata od klijenta.

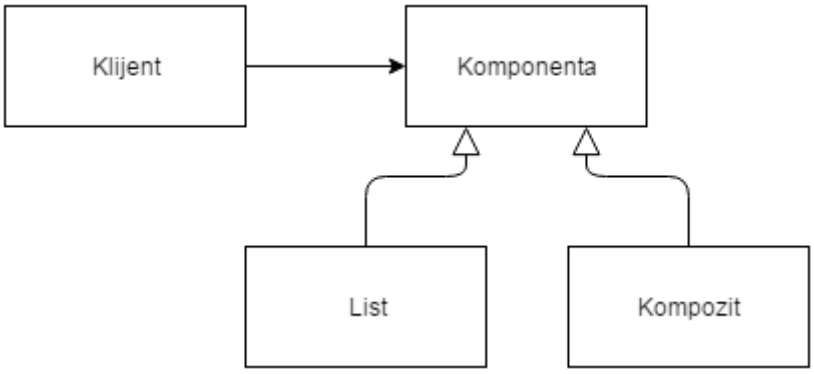


Slika 2.6. Struktura Mosta

Kompozit

Oblikovni obrazac Kompozit (engl. *Composite*) omogućuje predstavljanje hijerarhije objekata u obliku jedne zajedničke strukture. Klijenti se tako mogu odnositi prema hijerarhiji objekata jednako kao i prema pojedinačnom objektu. Korištenjem Kompozita klijent može ignorirati razlike između skupine objekata iz jednog dijela hijerarhije i između samostalnih objekata.

Kompozit se sastoji od Klijenta, Kompozita, Komponente (engl. *Component*) i Lista (engl. *Leaf*) (Sl.2.7, nalik na [4]). List predstavlja objekt u hijerarhiji (stablu) koji nema djece, tj. podređenih objekata, a Kompozit predstavlja objekt koji ima djece, tj. predstavlja roditeljski objekt. Klijent koristi Komponentu kao sučelje za interakciju s objektima iz strukture obrasca Kompozit. Ukoliko Klijent zahtjeva List, on mu se predaje direktno, a ukoliko se zahtjeva Kompozit, zahtjev se predaje do Lista koji obavlja željenu operaciju.



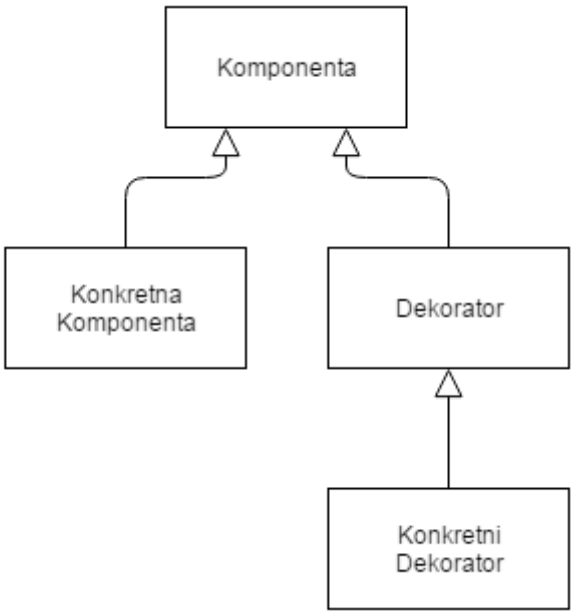
Slika 2.7. Struktura Kompozita

Uz Kompozit klijenti postaju jednostavniji, jer nije potrebno provjeravati radi li se o čvoru u stablu ili o listu jer će obrazac sam rukovati s objektima i predati klijentu objekt koji izvršava zahtijevanu operaciju. Također, pojednostavljuje se i dodavanje novih komponenata u strukturu objekata jer nije potrebna promjena klijenta kako bi ga uskladili s novom komponentom.

Dekorator

Strukturni obrazac Dekorator (engl. *Decorator*) dinamički dodaje objektu nove mogućnosti. Dekorator je jedan od oblikovnih obrazaca koji omogućuje povećavanje funkcionalnosti koje se obično dobiva nasljeđivanjem, odnosno stvaranjem podklasa. Prema [4], Dekorator se koristi za dodavanje novih mogućnosti jednom objektu, bez odraza na ostale objekte. Mogućnosti objektu mogu se dodati i samo privremeno, tj. mogu biti prisutne do određenog događaja. Iako se Dekorator može zamijeniti stvaranjem podklasa, on postaje koristan kada stvaranje podklase nije moguće ili kada rješavanje problema kao posljedicu stvara

nepregledan broj podklasa kojima se pokrivaju sve moguće kombinacije. Obrazac Dekorator sastoji se od Komponente, Konkretne komponente (engl. *Concrete Component*), Dekoratora i Konkretnog Dekoratora (engl. *Concrete Decorator*) u obliku prikazanom na slici 2.8, prema [4]. Dekorator predaje zahtjev Komponenti koja zatim definira novu funkcionalnost koja će biti dodana objektu.

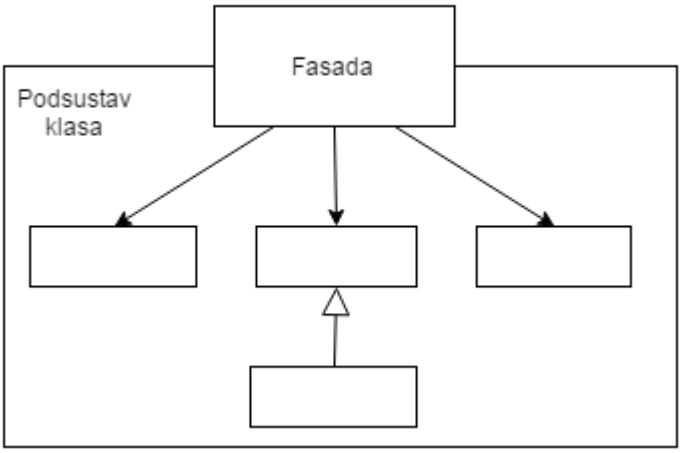


Slika 2.8. Struktura Dekoratora

Korištenjem Dekoratora povećava se fleksibilnost pri dodavanju novih funkcionalnosti objektu jer se funkcionalnosti mogu dodavati dok se program izvršava. Time se dobiva mogućnost da se ne mora točno isplanirati izgled klase već se funkcionalnosti pomoću Dekoratora dodaju kada postanu potrebne. Negativna strana Dekoratora je stvaranje mnoštva sličnih objekata koji postanu teško razumljivi.

Fasada

Fasada (engl. *Facade*) je oblikovni obrazac koji omogućuje opće sučelje sustava za upravljanje sučeljima podsustava, odnosno definira višu razinu sučelja kako bi se olakšalo korištenje podsustava. Korištenje velikog broja oblikovnih obrazaca rezultira velikim brojem manjih klasa. Tada je moguće koristiti Fasadu kako bi se omogućilo jednostavnije sučelje za kompleksne podsustave [5]. Fasada se još koristi i za stvaranje slojeva među podsustavima. Na slici 2.9, izrađenoj prema [4], prikazana je struktura Fasade, a sastoji se od klasa u podsustavu i same klase Fasada (engl. *Facade*).



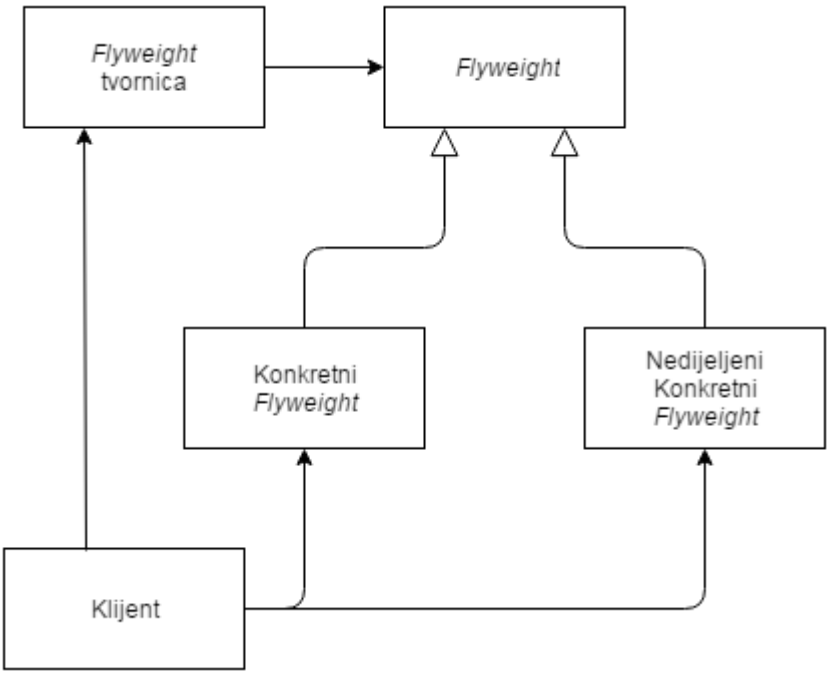
Slika 2.9. Struktura Fasade

Klijent koji želi komunicirati s klasama u podsustavu nije u mogućnosti komunicirati direktno već mora preko Fasade. Klijent šalje zahtjev Fasadi, a ona prije prenošenja zahtjevu klasi u podsustavu izmjenjuje dobiveni zahtjev kako bi bio prilagođen primatelju zahtjeva. Korištenjem Fasade stvara se dodatni sloj između klijenta i klase koju klijent želi koristiti. Prednost uvođenja navedenog sloja je olakšavanje korištenja podsustava zbog smanjenja broja objekata s kojima klijent mora komunicirati.

Flyweight

Strukturni obrazac *Flyweight* je dijeljeni objekt koji se koristi kao potpora velikom broju objekata. Može se koristiti u više konteksta istovremeno, a u svakom se ponaša kao samostalan objekt. Pri korištenju *Flyweight* obrasca nailazi se na pojmove unutarnje i vanjsko stanje. Unutarnje stanje se nalazi u *Flyweight* objektu, a vanjsko stanje klijent predaje *Flyweight*u kada želi koristiti njegove funkcije. Prema [4], *Flyweight* se koristi kada su ove tvrdnje istinite: aplikacija koristi velik broj objekata, zbog velikog broja objekata postoje veliki troškovi memorije, većina objekata sadrži vanjsko stanje za predaju, grupe objekata se mogu zamijeniti sa nekoliko dijeljenih objekata kada im se odbaci vanjsko stanje i aplikacija ne ovisi o identitetima objekata.

Struktura ovog oblikovnog obrasca sastoji se od *Flyweight*a, Konkretnog *Flyweight*a (engl. *Concrete Flyweight*), Nedijeljenog konkretnog *Flyweight*a (engl. *Unshared Concrete Flyweight*), *Flyweight* tvornice (engl. *Flyweight Factory*) i klijenta kao što je prikazano na slici 2.10, izrađenoj prema [4], Tvornica stvara i upravlja *Flyweight* objektima koji sadrže unutarnje stanje. Klijent sadrži vanjsko stanje i traži od tvornice *Flyweight* objekte koji su mu potrebni. Osim standardnog konkretnog *Flyweight* objekta postoje i nedijeljeni objekti koji nisu stvoreni za dijeljenje.



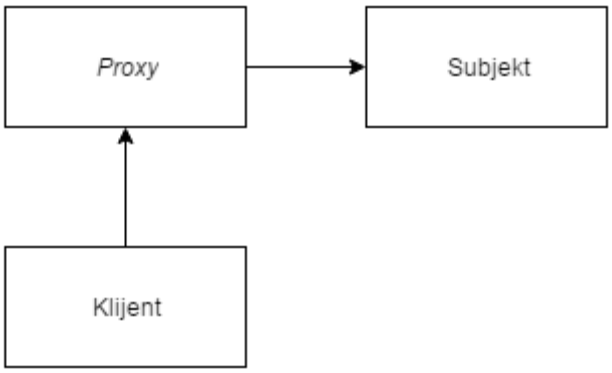
Slika 2.10. Struktura *Flyweight*a

Obrazac *Flyweight* smanjuje troškove memorije jer smanjuje broj potrebnih objekata, ali povećava vrijeme izvođenja zbog potrebnog pretraživanja i premještanja podataka. Kada se koristi za veći broj objekata, njegova mogućnost smanjenja potrebnog mjesta za spremanje podataka postaje daleko veća prednost od negativne strane da usporava izvođenje programa.

Proxy

Proxy omogućuje zamjenika za drugi objekt kako bi se mogao kontrolirati pristup tom objektu. Oblikovni strukturni obrazac *Proxy* koristi se kada je potrebna bolja referenca prema objektu od standardnog pokazivača [6]. Primjeri korištenja su: kontroliranje pristupa objektu (čuvar), omogućavanje predstavnika u različitim adresnim prostorima (udaljeni), stvaranje objekta koji zahtijevaju puno resursa (virtualni), pokazivač koji ima dodatne mogućnosti (pametni), itd.

Struktura obrasca prikazana je na slici 2.11, koja je izrađena prema [4]. Obrazac se sastoji od *Proxy*ja i Subjekta (engl. *Subject*), tj. objekta koji *Proxy* predstavlja. Klijent poziva *Proxy* objekt koji zatim obavlja posao za koji je stvoren (čuvanje, virtualno stvaranje, „pametne“ aktivnosti, itd.) i dalje prenosi zahtjev prema pravom objektu, Subjektu.



Slika 2.11. Struktura *Proxya*

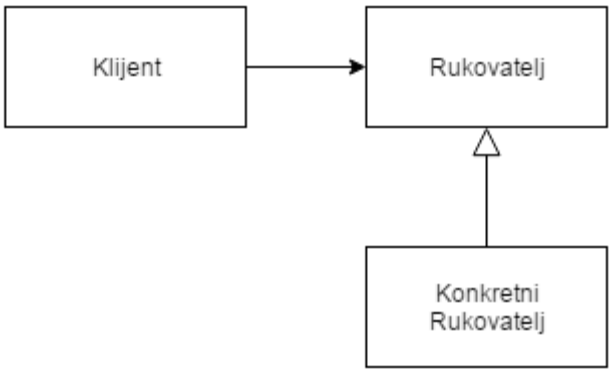
Ovisno o tipu *Proxy*ja, tj. razlogu zbog kojeg je stvoren kao pokazivač na drugi objekt, ovise i njegove prednosti. Udaljeni *Proxy* može sakriti podataka da se pravi objekt nalazi negdje dalje u memoriji. Virtualni *Proxy* stvara objekt na zahtjev, dok ostali mogu pružati dodatne usluge kada se pokušava pristupiti pravom objektu.

2.2.3. Obrasci ponašanja

Prema [4], obrasci ponašanja primjenjuju se za probleme vezane uz algoritme i dodjeljivanje odgovornosti među objektima. Također, opisuju i najbolji način za stvaranje komunikacije među objektima. Među obrascima ponašanja nalaze se, kao i kod strukturnih obrazaca, obrasce vezane uz ponašanje objekata i obrasce koji se tiču ponašanja klasa. Ostali obrasci vezani su uz enkapsuliranje ponašanja unutar objekta. Neki od obrazaca ponašanja su: Lanac odgovornosti, Naredba, Interpreter, Iterator, Posrednik, Uspomena, Promatrač, Stanje, Strategija, Metoda-predložak i Posjetitelj.

Lanac odgovornosti

Obrazac ponašanja Lanac odgovornosti (engl. *Chain of Responsibility*) spaja objekte koji su u mogućnosti izvršiti zahtjev. Tako pošiljatelj neće poslati zahtjev samo jednom određenom primatelju već će više objekata imati mogućnost izvršiti zahtjev. Zahtjev će se prenositi s objekta na objekt dok jedan od objekata ne obradi zahtjev. Lanac odgovornosti se koristi kada postoji više objekata koji mogu obraditi zahtjev, a nije točno određen objekt koji će obraditi zahtjev. Koristi se i kada objekti za obradu zahtjeva trebaju biti određeni dinamički. Struktura obrasca se sastoji od rukovatelja (engl. *Handler*), konkretnog rukovatelja (engl. *Concrete handler*) i klijenta (Sl.2.12, prema [4]). Klijent postavlja zahtjev rukovatelju koji zatim prenosi zahtjev kroz lanac konkretnih rukovatelja dok jedan od konkretnih rukovatelja ne izvrši zahtjev.

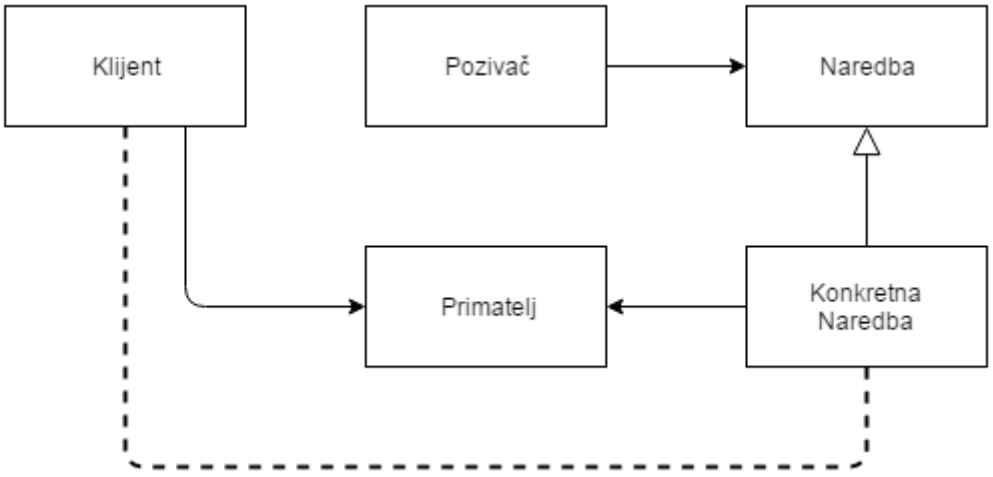


Slika 2.12. Struktura Lanca odgovornosti

Lancem odgovornosti smanjuje se potreba za povezivanjem. Objekt ne mora znati tko točno može obraditi objekt i kome je potrebno poslati zahtjev, već samo zna da će zahtjev biti izvršen na željeni način. Negativna strana Lanca odgovornosti je da upravo to što se ne zna tko će točno obrađivati zahtjev. Može se i dogoditi da će zahtjev doći do kraja lanca i neće biti izvršen.

Naredba

Obrazac Naredba (engl. *Command*) je zahtjev enkapsuliran u objekt. Tako je omogućeno lakše kontroliranje zahtjeva, odnosno mogu se nad njima odrađivati operacije koje inače nisu moguće (npr. stvaranje liste čekanja). Glavna svrha Naredbe je omogućavanje izvršavanja zahtjeva u željenom vremenu. Naredbom je moguće stvoriti listu čekanja prema kojoj se zatim zahtjevi izvršavaju. Naredba donosi i podršku za pohranjivanje promjena u sustavu koje se mogu koristiti za povratak sustava u kvaru u prethodno valjano stanje. Slikom 2.13, izrađenoj prema [4], prikazana je struktura oblikovnog obrasca Naredbe.



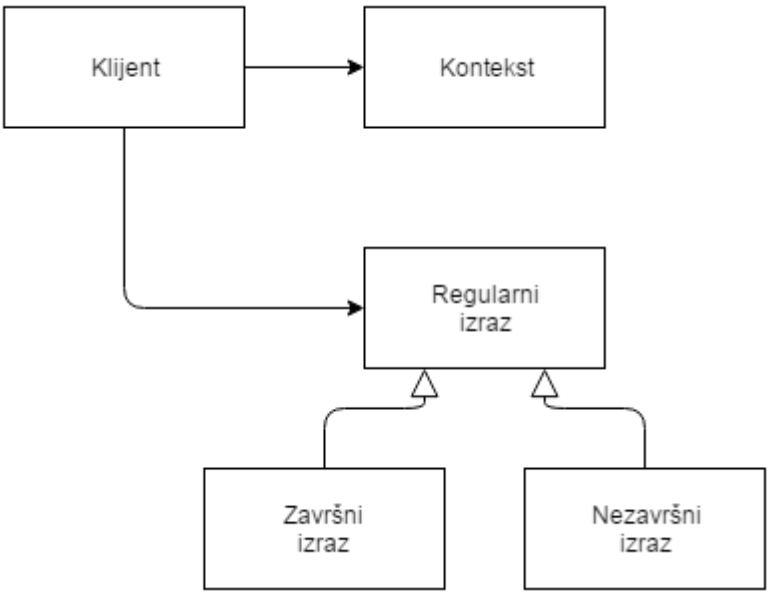
Slika 2.13. Struktura Naredbe

Klijent započinje stvaranjem Konkretne Naredbe (engl. *Concrete Command*) i određuje Primatelja (engl. *Receiver*). Pozivač (engl. *Invoker*) sprema Konkretnu Naredbu i poziva njeno izvršavanje. Dodavanje novih Naredbi u sustav je lagano jer za dodavanje nije potrebna promjena postojećih klasa. S obzirom da je Naredba objekt, moguće ju je nasljeđivati i proširivati. Detaljnije o Naredbi može se pročitati u [5].

Interpreter

Obrazac ponašanja Interpreter služi za prevođenje rečenica nekog jezika korištenjem gramatike tog jezika. Obrazac pomoću klasa predstavlja svako gramatičko pravilo. Sastoji se od apstraktne klase za regularne izraze koju nasljeđuju klase za doslovne izraze, alternacije, nizane izraze i izraze koji se ponavljaju [4]. Interpreter najbolje funkcionira kada je gramatika jednostavna, jer se pri složenijom gramatikom povećava klasna hijerarhija koju je tada teško održavati.

Struktura obrasca Interpretera prikazana je na slici 2.14, izrađenoj prema [4], i sastoji se od klijenta, apstraktne klase koja predstavlja regularne izraze (engl. *Abstract Expression*), klasa koje nasljeđuju apstraktnu klasu: završni izrazi (engl. *Terminal Expression*) i nezavršni izrazi (engl. *Nonterminal Expression*) i konteksta (engl. *Context*). Klijent predaje rečenicu klasi za regularni izraz koja zatim pomoću podklasa prevodi rečenicu i pohranjuje u kontekst.



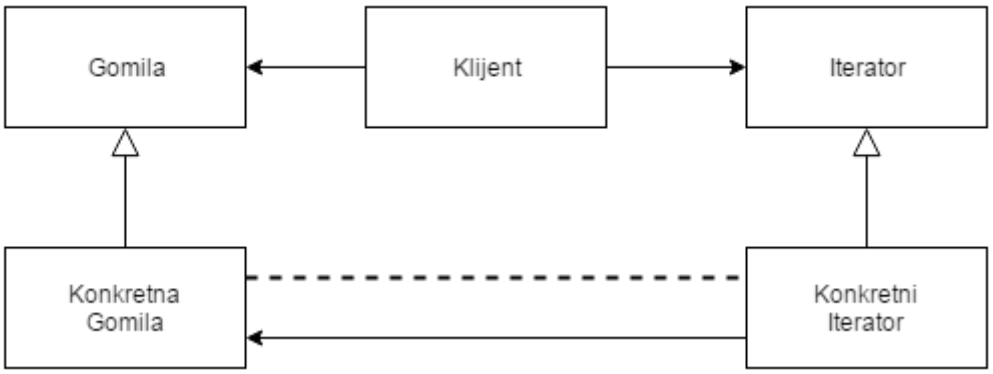
Slika 2.14. Struktura Interpretera

Prednost Interpretera je lagano dodavanje, promjene i proširenja gramatike. Za rad nad gramatikom potrebna je promjena klasa i podklasa koje ju opisuju. Negativna strana Interpretera je održavanje kompleksnih klasa upravo iz razloga što je svako gramatičko

pravilo opisano s jednom klasom. Što znači da je za složenu gramatiku potrebna i složena klasna hijerarhija koja tada postaje zahtjevna za održavanje.

Iterator

Oblikovni obrazac Iterator omogućuje sekvencijalni pristup elementima objekta koji je sastavljen od većeg broja elemenata bez otkrivanja inherentnih dijelova tog objekta. On omogućuje prolazak kroz listu elemenata objekta za one objekte koji inače nemaju to svojstvo. Obrazac se sastoji od Gomile (engl. *Aggregate*), Konkretne gomile (engl. *Concrete Aggregate*), Iteratora (engl. *Iterator*) i Konkretnog iteratora (engl. *Concrete Iterator*) kako je prikazano na slici 2.15, koja je izrađena prema [4].

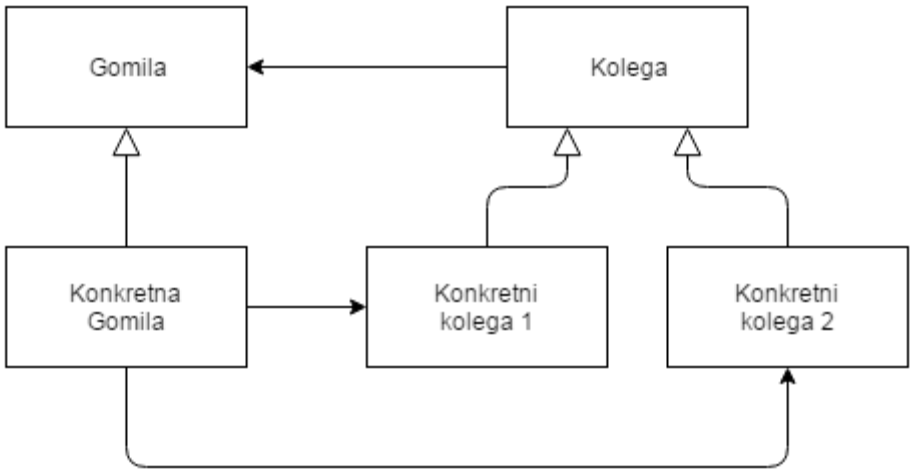


Slika 2.15. Struktura Iteratora

Gomila je objekt u kojem se nalaze elementi kroz koje će prolaziti Iterator. Konkretni iterator stvara sučelje Iteratora i prati poziciju u Gomili. Iterator tako pojednostavljuje potrebno sučelje Gomile (objekta sa elementima) jer tada nije potrebno stvaranje nove sposobnosti za prolazak kroz elemente već se koristi oblikovni obrazac Iterator.

Posrednik

Ponašajni obrazac Posrednik (engl. *Mediator*) opisuje stvaranje dodatnog objekta preko kojeg skupina drugih objekata komunicira. Objekti više neće komunicirati direktno jedan s drugim, već će komunicirati preko Posrednika. U [4] je objašnjeno da se time smanjuje potreba objekata da budu dizajnirani prema objektu s kojim komuniciraju. Oblikovni obrazac Posrednik koristi se kada je ponovno korištenje istog objekta zahtjevno jer je objekt povezan s nekoliko drugih objekata. Drugi primjer korištenja ovog obrasca je za objekte koji su povezani u dobro definiranom sustavu, ali na vrlo kompleksan način. Struktura Posrednika, prikazana na slici 2.16, slično kao u [4], sastoji se od Posrednika, Konkretnog posrednika (engl. *Concrete Mediator*) i klasa koje međusobno komuniciraju, tzv. Klase kolege (engl. *Colleague classes*).



Slika 2.16. Struktura Posrednika

Klase koje međusobno komuniciraju sada će komunicirati preko Posrednika. Zahtjeve šalju njemu i od njega ih primaju natrag. Posrednik sadrži ponašanje kojim usmjerava zahtjeve prema komuncirajućim klasama. Pozitivne strane obrasca Posrednika su pojednostavljenje protokola komunikacije među objektima, centraliziranje kontrole komunikacije i lako unapređenje komunikacije jer je potrebno stvoriti podklasu jedino klase Posrednika.

Uspomena

Oblikovni obrazac ponašanja Uspomena (engl. *Memento*) pamti stanje u kojem se objekt nalazio. Stanje se pamti bez ugrožavanja enkapsulacije objekta. Uspomenom je moguće vratiti objekt u spremljeno prethodno stanje. Obrazac Uspomena se koristi kada bi direktno spremanje trenutnog stanja objekta ugrozilo njegovu implementaciju i uništilo enkapsulaciju objekta. Korištenje oblikovnog obrasca Uspomene pojednostavljuje klasu Originala i omogućuje da podaci Originala ostanu sačuvani. Negativna strana je što kopiranje velikog broja informacija koje sadrži Original znači i dupliranje potrebnog memorijskog prostora.

Obrazac se izvršava pomoću tri klase (Sl.2.17, prema [4]): Originala (engl. *Originator*), tj. klase čije stanje objekta će se spremati, Uspomene koja sprema stanje Originala i Čuvara (engl. *Caretaker*) koji je odgovoran za čuvanje stanja. Čuvar zahtjeva sliku stanja od Originala preko Uspomene i zatim vraća stanje kada bude potrebno vratiti Original u prethodno stanje.

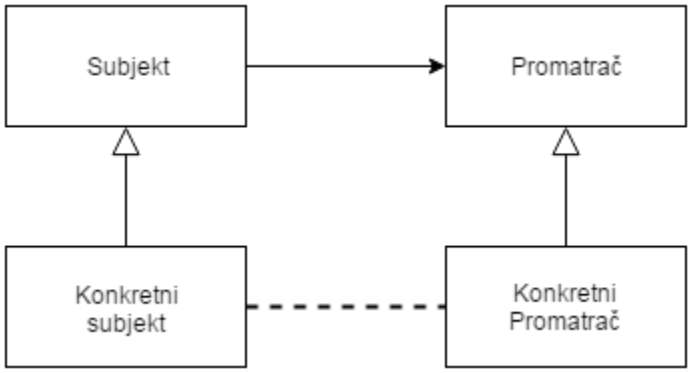
Slika 2.17. Struktura Uspomene

Promatrač

Obrazac Promatrač (engl. *Observer*) stvara vezu jedan-prema-više između objekata [6]. Kada glavni objekt promjeni svoje stanje, pomoću poveznice se obavještavaju svi ostali objekti i izvode svoja ažuriranja. U toj vezi glavni objekt se naziva subjektom, a ostali su promatrači. Subjekt može imati bilo koji broj promatrača.

Promatrač se koristi kada postoje dvije odvojene strane. Enkapsuliranjem svake strane omogućuje se odvojena promjena i ponovno korištenje svake od strana. Koristi se i kada je nakon promjene jednog objekta potrebna promjena i ostalih, iako nije poznat točan broj objekata kojima je potrebna promjena. Promatrač omogućuje i obavještavanje ostalih objekata bez točnog saznanja o kojim objektima je riječ.

Ovaj oblikovni obrazac sastoji se od Subjekta (engl. *Subject*) i podklase Konkretnog subjekta (engl. *Concrete Subject*), te Promatrača i podklase Konkretnog promatrača (engl. *Concrete Observer*) kako je i prikazano na slici 2.18, koja je izrađena prema [4]. Nakon promjene, Konkretni subjekt obavještava svoje promatrače koji zatim mogu zatražiti informacije o promjeni i prema njoj se uskladiti.



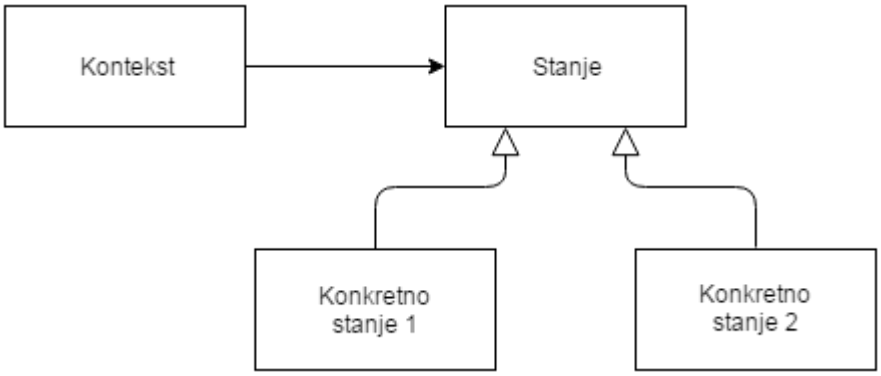
Slika 2.18. Struktura Promatrača

Pozitivna strana oblikovnog obrasca Promatrač je što subjektu nije potrebna informacija kome točno šalje obavijest i koji je broj promatrača čime se postiže sloboda za dodavanje i brisanje promatrača. Negativna strana toga je što programer mora paziti da obavijest ne primi krivi promatrač, jer bi to moglo uzrokovati daljnje probleme.

Stanje

Oblikovni obrazac ponašanja Stanje (engl. *State*) omogućuje objektu promjenu ponašanja nakon što dođe do promjene stanja objekta. Nakon promjene stanja stvorit će se objekt koji će promijeniti klasu glavnog objekta. Obrazac Stanje koristi se kada ponašanje objekta ovisi o njegovom stanju, pa je potrebno promijeniti ponašanje objekta dok se program izvodi. Obično

se stanje objekta opisuje sa nekoliko varijabli. Oblikovni obrazac Stanje razdvaja svaku varijablu u poseban objekt kako bi stanje objekta postalo neovisno. Za stvaranje obrasca Stanje potrebna je klasa za Kontekst (engl. *Context*), klasa za Stanje i podklase za Konkretna Stanja (engl. *Concrete State*). Struktura Stanja prikazana je na slici 2.19, koja je izrađena prema [4]. Kontekst je definiran prema željama klijenta. Nakon promjene u klijentu, Kontekst šalje zahtjev Stanju. Stanje, pomoću svojih podklasa za svakog stanje klijenta (Konkretna stanja), obrađuje zahtjev i mijenja ponašanje klijenta.

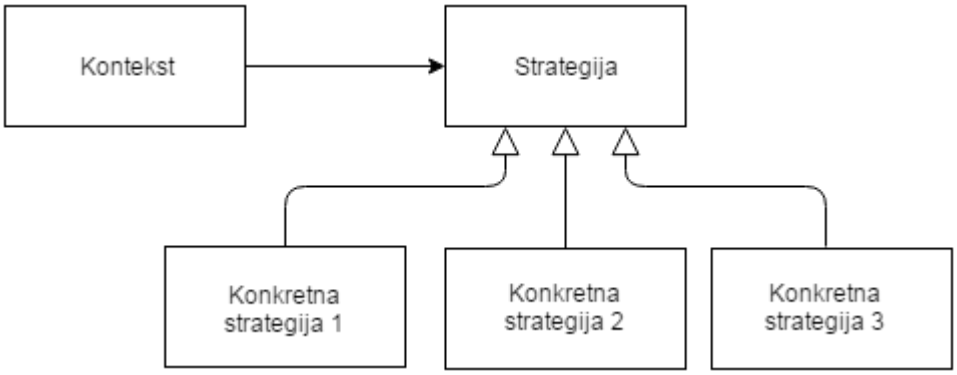


Slika 2.19. Struktura Stanja

Strategija

Oblikovni obrazac Strategija (engl. *Strategy*) omogućava razdvajanje algoritma i klijenta koji ga koristi. Prema [4], to se postiže definiranjem algoritma, njegovom enkapsulacijom i omogućavanjem zamjene za neki drugi algoritam. Obrazac Strategija koristi se kada algoritam koristi podatke za koje klijent ne treba znati i/ili kada će biti potrebno više različitih oblika jednog algoritma. Srodne klase se najčešće razlikuju prema svojim metodama, odnosno ponašanjem. Korištenjem ovog obrasca, moguće je stvaranje samo jedne klase, a pomoću Strategije se omogućuje klasi različita ponašanja.

Struktura obrasca Strategija prikazana je na slici 2.20, izrađenoj prema [4]. Ona sadrži Kontekst (engl. *Context*), Strategiju i Konkretne strategije (engl. *Concrete Strategy*). Strategijom se definira sučelje koje je dizajnirano kao opće za sve korištene algoritme, a Konkretnim strategijama se opisuje svaki algoritam posebno. Kontekst povezuje klijent sa algoritmima, tj. prima zahtjeve od klijenta i preuzima algoritme od Strategije.



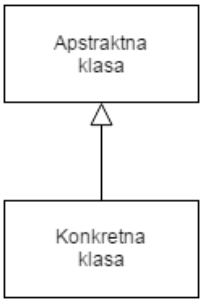
Slika 2.20. Struktura Strategije

Strategijom se omogućava alternativa za stvaranje podklasa jer postaje moguće da jedna klasa uz strategiju može imati različite uloge. Pozitivna stvar je i da klijenti dobivaju mogućnost biranja koji algoritmom će biti izvršen određeni zadatak. Negativne strane su povećanje broja objekata u sustavu i činjenica da klijenti moraju poznavati koje strategije, odnosno algoritmi su im na raspolaganju.

Metoda-predložak

Oblikovni obrazac Metoda-predložak (engl. *Template Method*) omogućuje definiranje glavnog dijela algoritma u nadređenoj klasi kako bi podklase mogle dovršiti algoritam svaka na svoj način. Tako je u klasi definirano opće ponašanje, a u podklasama konkretno ponašanje algoritma. Obrazac Metoda-redložak koristi se za implementaciju dijelova algoritma koji nisu promjenjivi kako bi podklase mogle dovršiti ostatak algoritma. Koristi se i kako ne bi došlo do dupliciranja koda ukoliko postoji metoda koja je jednaka za više klasa. Metoda-predložak smatra se temeljnom tehnikom za pisanje koda koji se koristi na više mjesta u jednakom obliku [4]. Obrazac se najčešće koristi u apstraktnim klasama, kao npr. u bibliotekama.

Izgled strukture Metode-predložak prikazan je na slici 2.21 i sastoji se od Apstraktne klase (engl. *Abstract Class*) i Konkretne klase (engl. *Concrete Class*). Apstraktna klasa sadrži kostur algoritma koji je jednak za svaki oblik takvog algoritma, a Konkretne klase sadrže ostatak algoritma prema kojem se algoritmi razlikuju jedan od drugog.

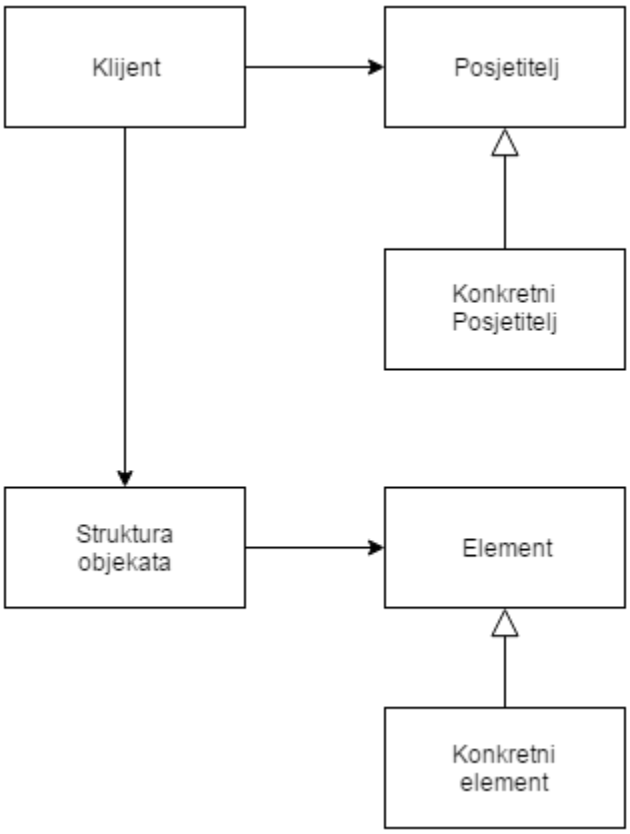


Slika 2.21. Struktura Metode-predložak

Posjetitelj

Obrazac ponašanja Posjetitelj (engl. *Visitor*) predstavlja svaku operaciju (metodu) nad elementima strukture objekata [5]. Omogućuje i definiranje novih operacija bez promjene klasa elemenata koje će koristiti te operacije. Posjetitelj se koristi kada postoji potreba za promjenom objekata čija se struktura sastoji od većeg broja klasa sa različitim sučeljima. Pogotovo kada se struktura rijetko ili nikada ne mijenja, ali su potrebe za promjenama objekata (tj. elemenata) česte. Posjetitelj omogućuje i čuvanje sličnih operacija zajedno unutar jedne klase.

Struktura Posjetitelja prikazana na slici 2.22, izrađenoj prema [4], sastoji se od samog Posjetitelja, Konkretnog posjetitelja (engl. *Concrete Visitor*), Elementa, Konkretnog elementa (engl. *Concrete Element*) i Strukture objekta (engl. *Object Structure*). Kako bi klijent koristio Posjetitelja mora stvoriti Konkretne posjetitelje za prolazak kroz cijelu strukturu objekata. Konkretni posjetitelj zatim posjećuje svoj element, a element mu omogućuje pristup svom stanju.



Slika 2.22. Struktura Posjetitelja

Pozitivna strana oblikovnog obrasca Posjetitelj je lako dodavanje novih operacija za pojedini element neovisno o kompleksnosti strukture, ali je dodavanje novih Konkretnih elemenata

teško. Razlog je potreba za stvaranjem novih apstraktnih metoda unutar Posjetitelja i potrebna implementacija unutar svake klase Konkretnog posjetitelja.

2.3. Načela objektno orijentiranog dizajna

Visoka kohezija sa slabim vezama između elemenata sustava i enkapsulacija podataka su temelji objektno orijentiranog programiranja. Slabe veze označavaju mogućnost sustava za laku promjenu elemenata bez potrebe za novim dizajniranjem, a visoka kohezija simbolizira mogućnost povezivanja zasebnih elemenata u radne cjeline. Posljednji temelj je enkapsulacija podataka elementa zbog potrebe reduciranja pristupa podacima ostalim elementima u sustavu. Temelji se ostvaruju pomoću pet načela objektno orijentiranog dizajna pod nazivom S.O.L.I.D. koje je osmislio Robert C. Martin 1990-ih godina [8].

Spomenuta načela dobila su ime prema prvom slovu naziva pet glavnih načela objektno orijentiranog dizajna, a tako tvore englesku riječ *solid* koja se prevodi kao „čvrsto“ čime se upućuje na čvrstoću temelja arhitekture sustava koja se ostvaruje načelima. Prvo načelo, Načelo jedne dužnosti (engl. *Single Responsibility Principle*) govori da klasa treba imati jedan i samo jedan razlog za promjenu [5]. Načelo objašnjava da ukoliko klasa ima dva gledišta na problem, odnosno dva razloga za prilagođavanje problemu onda je potrebno klasu razdvojiti na dvije klase. To je obrnuto gledanje na izjavu da klasa treba imati samo jednu dužnost. Sljedeće načelo je Otvoreno-zatvoreno načelo (engl. *Open-Closed Principle*) koje kaže da se klasa treba moći proširiti bez mijenjanja. Drugim riječima to znači da klasa treba biti otvorena za proširivanje, ali zatvorena za promjenu već gotovih elemenata. Načelo se smatra glavnim dijelom objektno orijentiranog programiranja jer omogućuje ponovno korištenje klasa i mogućnost održavanja programa. Slovo L u skraćenici S.O.L.I.D. predstavlja Načelo zamjene Liskov (engl. *Liskov Substitution Principle*). Načelo govori da izvedene klase moraju biti zamjena za osnovnu klasu. Ime je dobio prema znanstvenici Barbari Liskov koja je rekla da ukoliko je objekt B izveden od objekta A (naslijeđen) tada objekt B mora imati mogućnost potpuno zamijeniti objekt A bez gubljenja svojstava objekta A. Predzadnje načelo je Načelo odvajanja sučelja (engl. *Interface Segregation Principle*) koje tvrdi da je potrebno stvoriti sučelje koje je razumljivo i lagano za korištenje. Prema [8], klijentu bi trebao biti jasan tok korištenja metoda i atributa i ne bi trebao pristupati informacijama koje nisu potrebne za rad sa sučeljem. Zadnje načelo, Načelo inverzije ovisnosti (engl. *Dependency Inversion Principle*), govori da funkcija koja koristi referencu na osnovnu klasu, mora moći koristiti i

referencu na klasu izvedenu od osnovne (naslijeđenu). Načelima se dobiva sustav koji može savladati promjene, ima mogućnost proširivanja bez posljedica i otporan je na kvarove.

2.4. Anti-obrasci

Oblikovni obrasci postoje zbog problema koji se često pojavljuju u relativno sličnim oblicima, pa su oblikovni obrasci ispravna rješenja za takve probleme. S druge strane, mladi i neiskusni programeri probleme znaju rješavati na krivi način. Zbog toga su nastali anti-obrasci, tj. primjeri krivog načina rješavanja problema na koje se često nailazi. Programerima služe kako bi mogli usporediti svoj način rješavanja problema i saznati dizajniraju li sustav neefikasno sa negativnim posljedicama [9].

Anti-obrasci su nastali nakon dokumentiranja najvažnijih oblikovnih obrazaca i smatra se da su inspirirani oblikovnim obrascima što potvrđuje i njihovo ime. Anti-obrasci nisu detaljno dokumentirani kao oblikovni obrasci, već se smatra da ih je stvorila programerska zajednica. Neki od najčešćih anti-obrazaca su: Klasa Bog, Špageti kod, Duh, Sustav kao cijev za peć, Zatvoren isporučiteljem i Corncob.

Klasa Bog (engl. *God Class*) metaforično predstavlja klasu koja je svemoguća. Klasa Bog sadrži veliki broj atributa i metoda koje uglavnom nemaju veze jedni s drugima. U klasi ne postoji objektno orijentirani dizajn, jako je kompleksna i teško ju je testirati. Stvaranje objekta takve klase zauzima previše memorije jer gotovo sigurno nisu potrebne sve mogućnosti koje klasa posjeduje. Prema [10], rješenje za Klasu Bog je razdvajanje u više klasa gdje će svaka klasa sadržavati metode i atribute koji su usko vezani.

Izraz Špageti kod (engl. *Spaghetti* Code) je najpoznatiji Anti-obrazac i označava kod bez strukture. Dijelovi koda su nepregledno povezani i na prvi pogled nemaju smisla. Takav program je vrlo teško održavati i nastaviti razvijati. Špageti kod je zbunjujuć čak i programeru koji ga je napisao. Zbog toga ga je teško popraviti pa se vrlo često mora napisati ispočetka. Zato je vrlo bitno da se Špageti kod spriječi planiranjem izgleda i toka kretanja sustava.

Duh (engl. *Poltergeist*) je Anti-obrazac koji predstavlja klasu sa ograničenom ulogom koja zbog toga ima malu efektivnost u sustavu. Takve klase su nepotrebne, neefektivne i ne prate smisao objektno orijentiranog dizajna. Rješenje je potpuno izbaciti klasu iz sustava, a njenu ulogu dodijeliti drugoj klasi.

Anti-obrazac Sustav kao cijev za peć (engl. *Stovepipe System*) predstavlja sustav u kojem su elementi nemaju zajedničko sučelje za komunikaciju već su čvrsto vezani jedan za drugi.

Takav sustav je teško promijeniti ili mu dodavati nove elemente. Cijev za peć je potrebno često popravljati i zbog toga se popravlja na najbrži način, bilo kako. Zbog toga ovakav sustav asocira na cijev za peć jer se pri dodavanju novog elementa stvara novo drugačije sučelje samo za taj element. Kako bi se Sustav kao cijev za peć popravio potrebno ga je redizajnirati i stvoriti jedinstveno sučelje pomoću kojeg bi elementi mogli nesmetano komunicirati.

Zatvoren isporučiteljem (engl. *Vendor Lock-In*) je anti-obrazac koji opisuje sustav ovisan o implementaciji isporučitelja. Takav sustav prisvoji tehnologiju isporučitelja kao dio svog sustava, ali nema kontrolu nad tim dijelom sustava. Nakon što isporučitelj promjeni ili poboljša svoju tehnologiju potrebno je promijeniti i ostatak sustava kako bi i dalje funkcionirali sinkronizirano. Time se ograničava cjelokupni sustav i ponekad je potrebno odgađati zamišljeni rok kako bi se popravljala šteta načinjena isporučiteljevom promjenom. Rješenje je stvoriti dodatni sloj između sustava isporučitelja kako promjene ne bi kritično utjecale na ostatak sustava.

Kukuruzni klip (engl. *Corncob*) je anti-obrazac koji ne predstavlja virtualni dio sustava već programera. Takav programer je zahtjevna osoba koja ne reagira dobro pod stresom. Približavanjem krajnjeg roka za isporuku programa, Kukuruzni klip stvara dodatne probleme i dodatni stres u timu. Razlog zbog kojeg se takav programer naziva Kukuruznim klipom je zbog fraze da „pukne kao kokica“. Neka od rješenja su smanjenje pritiska i odgovornosti u timu i stvaranje grupa za potporu kako bi se moglo pomoći programerima s takvim problemom.

3. UPORABA OBLIKOVNIH OBRAZACA NA ANDROID PLATFORMI

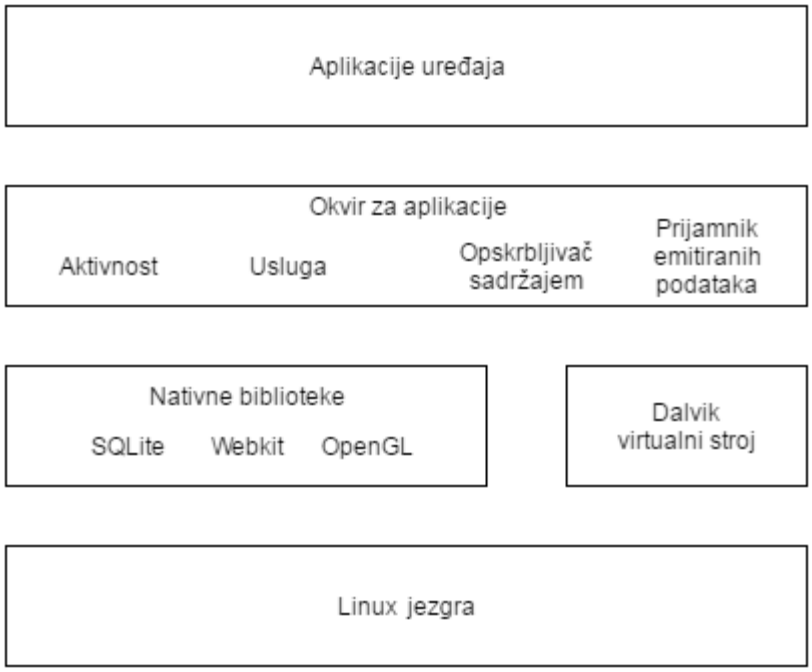
Operacijski sustav Android dizajniran je za mobilne uređaje. Razvio ga je Google, a pod vlasništvom je Open Handset Alliance. Open Handset Alliance je udruženje čiji je cilj stvoriti inovacije u mobilnom svijetu [11]. Android je platforma sa otvorenim kodom, što znači da bilo koji korisnih ima pristup kompletnom Android razvojnom sučelju. Otvoreni kod tako pomaže Androidu zbog stvaranja brojnih biblioteka i *API*-ja koje inače ne bi postojali.

Značajnija povijest Androida počinje 2005. godine nakon što je Google kupio Android. 2007. godine Android postaje platforma s otvorenim kodom i dio Open Handset Alliance. Tijekom 2009. godine izlaze početne verzije Androida od Cupcake (1.5) do Eclair (2.1). Danas, Android je najpopularniji operacijski sustav u velikoj većini država diljem svijeta i izvršava se na približno pet puta više uređaja nego konkurentski iOS.

Prema zadnjim podacima u [12], verzija Lollipop (5.0) nalazi se na najviše Android uređaja, a nakon nje dolaze Kit Kat (4.4), najnovija Marshmallow (6.0) i Jelly Bean (4.1.-4.3.). Te četiri verzije zajedno obuhvaćaju preko 96% svih Android uređaja što znači da se programeri mogu fokusirati samo na stvaranje Aplikacija koje podržavaju verzije od 4.1. i više. Sljedeća verzija, 7.0., izaći će krajem 2016. godine pod kodnim imenom Nougat.

3.1. Android platforma i razvoj programske podrške

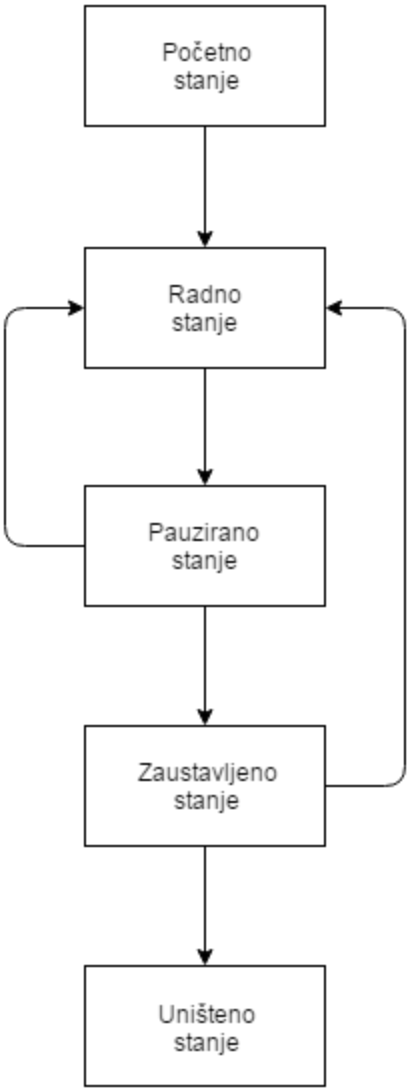
Android operacijski sustav složen je od nekoliko slojeva prikazanih na slici 3.1, izrađenoj prema [13]. Prvi sloj je Linux jezgra (engl. *kernel*) iz razloga što operacijski sustav Linux ima brojne mogućnosti, veliku sigurnosti i lako je prenosiv. Sljedeći sloj su nativne biblioteke Androida. Neke od biblioteka su: stroj za brzo prikazivanje web stranica Webkit, baza podataka SQLite, biblioteke za 3D grafiku OpenGL i implementacija Jave Apache Harmony. Uz nativne biblioteke na ovom sloju nalazi se i Dalvik virtualni stroj. Dalvik je izrađen kao zamjena za Java virtualni stroj, posebno stvoren samo za Android. Iako se Aplikacije pišu u Java programskom jeziku, nakon prevođenja u Javi ponovno se kod prevodi u Dalvik kod koji se zatim izvršava u Dalvik virtualnom stroju. U posljednjem sloju nalazi se okvir (engl. *framework*) za aplikacije. Prema [13], aplikacije imaju četiri temeljne komponente: Aktivnosti (engl. *Activity*), Usluge (engl. *Service*), Opskrbljivač sadržajem (engl. *Content Provider*) i Prijamnik emitiranih podataka (engl. *Broadcast Receiver*).



Slika 3.1. Slojevi Android operacijskog sustava

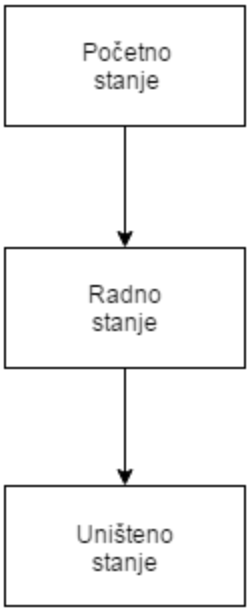
Aktivnost predstavlja zaslon na mobilnom uređaju. Stvaranje nove aplikacije započinje upravo stvaranjem podklase Aktivnosti i prema tome se može smatrati glavnim dijelom Android mobilne aplikacije. Aplikacije mogu sadržavati više od jedne aktivnosti, a za prijelaz iz jedne aktivnosti u drugu koriste se namjere (engl. *Intents*). Namjere su poruke koje se prenose između temeljnih sastojaka aplikacije. Mogu biti implicitne i eksplicitne. U eksplicitnim namjerama pošiljatelj točno određuje primatelja, dok u implicitnim pošiljatelj određuje tip primatelja i zatim sustav korisniku predlaže aplikacije koje mogu izvršiti zahtjev.

Stvaranje nove aktivnosti znači i stvaranje novog Linux procesa, zauzimanje memorije za grafičko sučelje i pripremu zaslona za korisnika za što je potrebna određena količina resursa. Aktivnosti zbog toga imaju životni ciklus (Sl.3.2, prema [13]). Životni ciklus se sastoji od pet stanja: početnog, radnog, pauziranog, zaustavljenog i uništenog stanja. U početnom stanju Aktivnost se stvara i obavljaju se prethodno navedene radnje. Radno stanje Aktivnosti označava vrijeme u kojem je Aktivnost na zaslonu i tada korisnik djeluje na nju. Kada Aktivnost prestane biti u fokusu, ona prelazi u pauzirano stanje. U pauziranom stanju Aktivnost i dalje troši resurse i postoji u memoriji. Sustav može brzo postaviti Aktivnost iz pauziranog u radno stanje. Aktivnost u zaustavljenom stanju i dalje postoji u memoriju, ali pri ponovnom pozivanju mora izvršiti metode koje se izvršavaju pri stvaranju Aktivnosti. Ukoliko se Aktivnost više neće vraćati u radno stanje ono se uništava i briše iz memorije, tj. prelazi u uništeno stanje.



Slika 3.2. Životni ciklus Aktivnosti

Sljedeći temeljni dio aplikacije su Usluge. Usluge nemaju korisničko sučelje i mogu se postaviti za izvršavanje zadataka u pozadini. Korisne su za izvršavanje zahtjeva dok je korisniku na zaslonu prikazan neki drugi sadržaj. Također sadrže životni ciklus, ali puno jednostavniji od životnog ciklusa Aktivnosti (Sl.3.3, prema [13]). Usluga može biti u početnom stanju, radnom stanju i uništenom stanju. Pri alokaciji memorije i zauzimanju ostalih resursa Usluga se nalazi u početnom stanju. Dok izvršava svoj zadatak nalazi se u radnom stanju, a nakon što zadatak izvrši prelazi u uništeno stanje u kojem se briše iz memorije.



Slika 3.3. Životni ciklus Usluge

Primjer korištenja Usluge je uspoređivanje trenutne lokacije korisnika s nekom prethodno odabranom lokacijom i obavještavanje korisnika ukoliko se lokacije podudaraju. Dok se Usluga izvršava, korisnik može neometano koristiti uređaj i obavljati neki drugi zadatak.

Aplikacije standardno onemogućavaju drugim aplikacijama pristup svojim podacima. Zbog toga postoji Opskrbljivač sadržajem. Opskrbljivač sadržajem je sučelje za izmjenu podataka između aplikacija [13]. On omogućuje izmjenu velikih podataka koji nisu mogući pomoću Namjera. Opskrbljivač sadržajem sadrži jednostavno sučelje sa metodama za upis, ažuriranje, brisanje podataka i metodom za postavljanje upita.

Posljednji temeljni dio Aplikacije je Prijamnik emitiranih podataka. On služi za osluškivanje promjena i slanje zahtjeva za odrađivanje zadataka nakon što se promjene dogode. Primjer mogućih promjena koje se osluškuju su paljenje uređaja, primanje SMS poruke ili dolaznog poziva. Nakon promjene izvršava se kod vezan uz promjenu, npr. paljenje bežičnog interneta nakon paljenja uređaja.

Razvojno okruženje i alati

Za izradu Android mobilne aplikacije potrebno je razvojno okruženje. Najčešće korišteno okruženje je Android Studio. Android Studio sadrži i Android alate za razvoj programske podrške (engl. *Software Development Kit*) i emulator koji predstavlja virtualni Android uređaj. Uz Android Studio potrebni su i Java alati za razvoj programske podrške. Programski kod za Android mobilne aplikacije piše se u objektno orijentiranom programskom jeziku Java

i označnom jeziku XML. Jezikom XML opisuje se statično korisničko sučelje, a Javom programski dio vezan uz opisivanje rada korisničkog sučelja, tj. njegovu interakciju.

3.2. Primjeri najčešće korištenih obrazaca u Android aplikacijama

Zbog toga što se u programiranju za Android koriste i Java alati za razvoj programske podrške, u Androidu se koriste oblikovni obrasci koji su implementirani unutar jednog ili oba razvojna alata. Korištenje oblikovnih obrasca nije točno definirano i dokumentirano, već se njihovo korištenje može prepoznati u kodu pomoću znanja o oblikovnim obrascima. Neki od obrazaca stvaranja koji se nalaze u razvojnim alatima su Jedinstveni objekt, Apstraktna tvornica i Graditelj [14]. Strukturni oblikovni obrasci prepoznati u Android i Java razvojnim alatima su Prilagodnik, Most, Kompozit, Dekorator i *Proxy*, a obrasci ponašanja prepoznati u razvojnim alatima su Iterator, Uspomena i Promatrač [15].

Klasa Aplikacija (engl. Application Class) je klasa koja održava glavno stanje aplikacije. Ona se poziva prva, prije pozivanja bilo koje druge klase unutar projekta. Klasa Aplikacija koristi obrazac Jedinstveni objekt jer je potrebno da postoji samo jedan objekt takve klase. Obrazac stvaranja Apstraktna tvornica korišten je pri stvaranju novih Aktivnosti, Servisa i Opskrbljivača sadržajem. Koristi se i u povratnim pozivima (engl. *Callbacks*) od prethodno navedenih temelja Android aplikacije. U razvojnim alatima često se pojavljuje Graditelj. Koristi se za stvaranje složenijih objekata koji imaju gotovo uvijek različiti izgled. Kao primjer su Graditelji za klasu Notifikaciju (engl. *Notification*) i Dijalog za upozorenje (engl. *Alert Dialog*), ali i Java klasu *StringBuilder*. Graditeljem za Notifikaciju može se postaviti naslov sadržaja, tekst sadržaja, ikona, boja, zvuk notifikacije i sl. *StringBuilder* koristi se za spajanje *stringova* jer standardna klasa za povezani niz znakova nema mogućnost dodavanja novog niza u postojeći. Jedan od Pogleda (engl. *View*) koji su definirani u Androidu je ListView. *ListView* sadrži listu članaka (engl. *Item*) čiji je oblik prethodno definiran. Kako bi pohranjene podatke mogli prikazati unutar članaka potrebno je koristiti Prilagodnik. U Android razvojnim alatima nalazi se osnovni prilagodnik koji se nasljeđuje i prilagođava prema potrebi. Zadatak Prilagodnika za *ListView* je popuniti svaki članak odabranim podacima. Pomoću obrasca Kompozit omogućuje se programeru da ignorira razlike između Pogleda i da ih može označavati na jednak način. Što znači da ukoliko je potrebno metodi predati Pogled, nije potrebno naznačiti točno koji Pogled će se predati. Ta metoda može primiti i Pogled koji predstavlja sliku i Pogled koji predstavlja tekst ili bilo koji drugi Pogled. Objektni obrazac Dekorator je korišten u Java klasama za ulaz i izlaz. Primjer takve klase je

InputStream, klasa koje se koristi za primanje ulaznog toka podataka. Ovisno o vrsti ulaznog podataka postoji i Dekorator koji će omogućiti klasi *InputStream* njegovo obrađivanje. U Javi postoji mogućnost stvaranje dva virtualna stroja korištenjem Java RMI-a (*Remote Method Invocation*)[16]. Objekt u prvom virtualnom stroju može pozvati metodu na objektu u drugom virtualnom stroju korištenjem udaljenog *Proxy*-ja. Obrazac *Proxy* omogućuje komunikaciju među objektima i poziva metodu na objektu u drugom virtualnom stroju. Obrazac ponašanja Iterator postoji kao gotova klasa u Java razvojnim alatima i prolazi kroz skupinu elemenata istog tipa. Sadrži implementirane metode: *hasNext()* koja vraća boolean vrijednost postoji li sljedeći element, *next()* vraća sljedeći element u iteraciji i *remove()* koji briše iz memorije zadnji element na kojem se Iterator nalazio. Često se koristi za prolazak kroz liste (engl. *Array List*) i za dohvaćanje elemenata u bazama podataka. Potrebno je zapamtiti stanje u kojem se trenutno nalazi Aktivnost. Tako da prilikom ponovnog pozivanja Aktivnosti i vraćanja na zaslon korisnika se točno zna stanje u kojem se Aktivnost nalazi. Zbog toga se koristi oblikovni obrazac ponašanja Uspomena u koji se sprema stanje Aktivnosti bez ugrožavanja njene enkapsulacije. Oblikovni obrazac Promatrač obavještava objekte o promjeni stanja objekta kojeg se promatra. Prijamnik emitiranih podataka funkcionira upravo prema obrascu Promatrač. Drugi primjer Promatrača u Androidu je i metoda Baznog prilagodnika *notifyDataSetChanged()* koja obavještava promatrače da su se promijenili podaci koji Prilagodnik prenosi i da je potrebno osvježiti Poglede sa novim podacima.

U Androidu se često koriste i oblikovni obrasci Model Pogled Predstavljač (engl. *Model View Presenter*) i Držač Pogleda (engl. *ViewHolder*) koje Družba četvorice nije opisala u svojoj dokumentaciji. Model Pogled Predstavljač omogućava odvajanje prezentacijskog sloja od logike [17]. Model predstavlja sloj u kojem se odvija logika, Pogled je sloj za prikaz, a Predstavljač je središnji dio koji povezuje Model i Pogled. Kada se pravila obrasca ne poštuju, stvaraju se Aktivnosti u kojima su pomiješani svi slojevi i koje je kao takve teško održavati i proširivati. Korištenjem obrasca postiže se smanjivanje odgovornosti Aktivnosti razdvajanjem na manje klase što rezultira povećanjem organiziranosti sustava. Oblikovni obrazac Držač Pogleda najčešće se koristi sa Prilagodnicima gdje ubrzava rad lista. Korisnik pomicanjem lista stvara potrebu za brzim pretraživanjem Pogleda i umetanjem u članak liste. Držač Pogleda omogućuje zaobilazak čestog korištenja skupih metoda pri pomicanju lista na način da čuva Poglede kako se ne bi morali ponovno pozivati i trošiti resurse. Nakon što Pogled postane potreban, jednostavno se preuzme od Držača Pogleda i postavi za korištenje.

4. PROGRAMSKO RJEŠENJE ZA KORIŠTENJE OBLIKOVNIH OBRAZACA PRI IZRADI ANDROID MOBILNE APLIKACIJE

4.1. Specifikacija mobilne aplikacije i zahtjevi korisnika

Zadatak diplomskog rada je izraditi Android mobilnu aplikaciju koja prikazuje primjenu oblikovnih obrazaca iz Android i Java alata za razvoj programske podrške. Svrha aplikacije je omogućiti korisniku pregled lokacija u gradu koje imaju povijesnu priču. Zahtjev aplikacije je da se pomoću Google Karte prikazuje trenutna lokacija korisnika i ostale lokacije u gradu s povijesnom pričom. Prilaskom lokaciji korisnik treba biti obaviješten da se nalazi blizu lokacije i trebaju mu biti pružene detaljnije informacije o mjestu na kojem se nalazi. Obavijest treba biti prikazana iako korisnik trenutno ne koristi aplikaciju. Prije otvaranja lokacija na karti korisniku treba biti prikazan popis svih lokacija. Lokacije se moraju moći filtrirati prema gradovima i nužno je omogućiti pristup detaljnijim informacijama o lokaciji. Nakon što korisnik pristupi lokaciji, potrebno ju je spremiti u korisnikovu povijest posjećenih lokacija. Aplikacija treba sadržavati i mogućnost prijavljivanja više korisnika i posebnu povijest o posjećenim lokacijama za svakog korisnika. Osim Android aplikacije potrebno je i web sučelje i baza podataka koja se nalazi na poslužitelju preko kojeg se treba odvijati dodavanje novih i pristupanje postojećim lokacijama. Popis svih zahtjeva korisnika i korisnički slučajevi koji prikazuju njihovo korištenje nalaze se u tablici 1.

Tablica 1. – Popis zahtjeva korisnika

ID	Prioritet	Opis	KS
1	2	Korisnik se treba prijaviti sa svojim korisničkim računom	KS1
2	2	Korisnik može napraviti novi korisnički račun	KS2
3	1	Korisnik može otvoriti kartu sa odabranim lokacijama	KS1
4	1	Korisnik može filtrirati lokacije prema gradu	KS2
5	2	Korisnik može osvježiti lokacije	KS3
6	2	Korisnik može otvoriti detalje o lokaciji	KS4
7	3	Baza podataka s lokacijama se treba nalaziti na web poslužitelju	-
8	2	Korisnik može otvoriti samo jednu lokaciju na karti	KS5
9	3	Korisnik može upaliti osluškivanje odabranih lokacija	KS6
10	3	Korisnik može ugasiti osluškivanje odabranih lokacija	KS7
11	2	Pri ponovnom filtriranju, gradovi trebaju biti označeni prema prethodnim postavkama	-
12	2	Moguće je imati više korisnika jedne aplikacije	-
13	1	Korisnik treba dobiti obavijest kada se približi lokaciji s karte	KS8
14	2	Korisnik može pregledati svoju povijest prethodno posjećenih lokacija	KS9
15	1	Izraditi mobilnu aplikaciju za Android uređaje	-
16	1	Izraditi web sučelje za dodavanje novih lokacija	-

Korisnički slučajevi

Nakon otvaranja aplikacije korisnik se treba prijaviti sa svojim korisničkim računom. Potrebno je unijeti svoje korisničko ime i ispravnu lozinku. Korisnički slučaj 1 opisuje tu radnju i prikazan je u tablici 2.

Tablica 2. – Korisnički slučaj 1

ID slučaja	KS1
Ime	Prijava sa korisničkim računom
Opis	Korisnik se treba prijaviti sa svojim korisničkim računom
Preduvjet	Nema
Glavni scenarij	1. Korisnik upisuje korisničko ime i lozinku 2. Sustav pregledava ispravnost 3. Korisnik se uspješno prijavljuje
Alternativni scenarij	2. Neispravno korisničko ime i/ili lozinka 1. Sustav nudi novi upis korisničkog imena i lozinke

Potrebno je i omogućiti korisniku pravljenje novog računa ukoliko nema postojeći ili ukoliko aplikaciju koristi drugi korisnik. Korisnički slučaj pravljenja novog računa prikazan je u tablici 3.

Tablica 3. – Korisnički slučaj 2

ID slučaja	KS2
Ime	Pravljenje novog korisničkog računa
Opis	Korisnik može napraviti novi korisnički račun
Preduvjet	Nema
Glavni scenarij	1. Korisnik odabire stvaranje novog korisničkog računa 2. Korisnik upisuje novo korisničko ime i lozinku 3. Sustav pregledava ispravnost 4. Sustav dodaje novog korisnika
Alternativni scenarij	3. Neispravno korisničko ime i/ili lozinka 1. Povratak na 2. 3. Korisnik već postoji u sustavu 1. Povratak na 2.

Nakon prijave potrebno je omogućiti korisniku otvaranje karte na kojoj će se nalaziti lokacije sa popisa. Otvaranje karte prikazano je korisničkim slučajem 3 u tablici 4.

Tablica 4. – Korisnički slučaj 3

ID slučaja	KS3
Ime	Otvaranje karte
Opis	Korisnik može otvoriti kartu sa prethodno odabranim lokacijama
Preduvjet	Korisnik je prijavljen
Glavni scenarij	1. Sustav dohvaća lokacije s poslužitelja 2. Korisnik zahtjeva otvaranje karte s lokacijama 3. Sustav otvara kartu 4. Sustav prikazuje lokacije na karti
Alternativni scenarij	1. Sustav nije povezan s poslužiteljem 1. Potrebno upaliti internet i ponoviti dohvaćanje lokacija

Filtriranje prema gradovima je korisnički slučaj 4 (tablica 5.). Potrebno je omogućiti korisniku prikazivanje lokacija samo iz gradova koje je korisnik odabrao.

Tablica 5. – Korisnički slučaj 4

ID slučaja	KS4
Ime	Filtriranje prema gradovima
Opis	Korisnik može filtrirati lokacije prema gradu
Preduvjet	Korisnik je prijavljen, dohvaćene lokacije
Glavni scenarij	1. Sustav prikazuje korisniku popis gradova 2. Korisnik odabire gradove 3. Sustav dohvaća s poslužitelja lokacije iz odabranih gradova 4. Sustav prikazuje popis lokacija
Alternativni scenarij	1. Korisnik ne odabire gradove 1. Sustav zahtjeva odabir barem jednog grada 3. Sustav nije povezan s poslužiteljem 1. Potrebno upaliti internet i ponoviti dohvaćanje lokacija

Potrebno je omogućiti korisniku da osvježi lokacije, odnosno preuzme nove lokacije s web poslužitelja. Slučaj je opisan u tablici 6. korisničkim slučajem 5.

Tablica 6. – Korisnički slučaj 5

ID slučaja	KS5
Ime	Osvježavanje lokacije
Opis	Korisnik može osvježiti lokacije
Preduvjet	Korisnik je prijavljen
Glavni scenarij	1. Korisnik zahtjeva osvježavanje lokacija 2. Sustav dohvaća s poslužitelja lokacije 3. Sustav prikazuje dohvaćene lokacije
Alternativni scenarij	1. Sustav nije povezan s poslužiteljem 1. Potrebno upaliti internet, povratak na 1.

Korisničkim slučajem 6 opisuje se slučaj u kojem korisnik otvara detalje o lokaciji i prikazan je u tablici 7.

Tablica 7. – Korisnički slučaj 6

ID slučaja	KS6
Ime	Detalji o lokaciji
Opis	Korisnik može otvoriti detalje o lokaciji
Preduvjet	Korisnik je prijavljen, lokacija je dohvaćena s poslužitelja
Glavni scenarij	1. Korisnik odabire lokaciju 2. Sustav prikazuje detaljnije informacije o lokaciji

Korisniku je potrebno omogućiti otvaranje samo jedne lokacije na karti (npr. nakon otvaranja detalja treba prikazati gdje se lokacija nalazi na karti). Takav slučaj opisan je korisničkim slučajem 7 (tablica 8.).

Tablica 8. – Korisnički slučaj 7

ID slučaja	KS7
Ime	Jedna lokacija na karti
Opis	Korisnik može otvoriti samo jednu lokaciju na karti
Preduvjet	Korisnik je prijavljen, lokacija je dohvaćena s poslužitelja
Glavni scenarij	1. Korisnik odabire lokaciju 2. Sustav prikazuje detaljnije informacije o lokaciji 3. Korisnik odabire prikaz lokacije na karti 4. Sustav otvara kartu i prikazuje lokaciju

Zahtjev je da ukoliko korisnik ugasi aplikaciju potrebno je i dalje osluškivati lokaciju na kojoj se korisnik nalazi. Primjer paljenja osluškivanja lokacije prikazan je u tablici 9.

Tablica 9. – Korisnički slučaj 8

ID slučaja	KS8
Ime	Paljenje osluškivanja lokacija
Opis	Korisnik može upaliti osluškivanje odabranih lokacija
Preduvjet	Korisnik je prijavljen, lokacije su dohvaćene s poslužitelja
Glavni scenarij	1. Korisnik odabire otvaranje karte 2. Sustav otvara kartu 3. Sustav prikazuje lokacije 4. Sustav pali osluškivanje lokacija
Alternativni scenarij	4. Ugašen GPS 1. Sustav traži od korisnika paljenje GPS-a

Primjer gašenja osluškivanja lokacije prikazan je u tablici 10.

Tablica 10. – Korisnički slučaj 9

ID slučaja	KS9
Ime	Gašenje osluškivanja lokacija
Opis	Korisnik može ugasiti osluškivanje odabranih lokacija
Preduvjet	Korisnik je prijavljen, upaljeno osluškivanje lokacija
Glavni scenarij	1. Korisnik u obavijestima odabire gašenje osluškivanja lokacije 2. Sustav otvara aplikaciju 3. Sustav zahtjeva potvrdu korisnika o gašenju 4. Korisnik potvrđuje 5. Osluškivanje lokacija se gasi
Alternativni scenarij	4. Korisnik ne potvrđuje 1. Sustav nastavlja sa osluškivanjem lokacija

Ukoliko korisnik priđe jednoj od odabranih lokacija, potrebno mu je poslati obavijest. Primjer rada slučaja za primanje obavijesti opisan je Korisničkim slučajem 10 (tablica 11.)

Tablica 11. – Korisnički slučaj 10

ID slučaja	KS10
Ime	Obavijest o približavanju lokaciji
Opis	Korisnik treba dobiti obavijest kada se približi lokaciji s karte
Preduvjet	Korisnik je prijavljen, upaljeno osluškivanje lokacija
Glavni scenarij	1. Sustav uspoređuje lokacije 2. Sustav pronalazi da se korisnik približio lokaciji 3. Korisnik prima obavijest

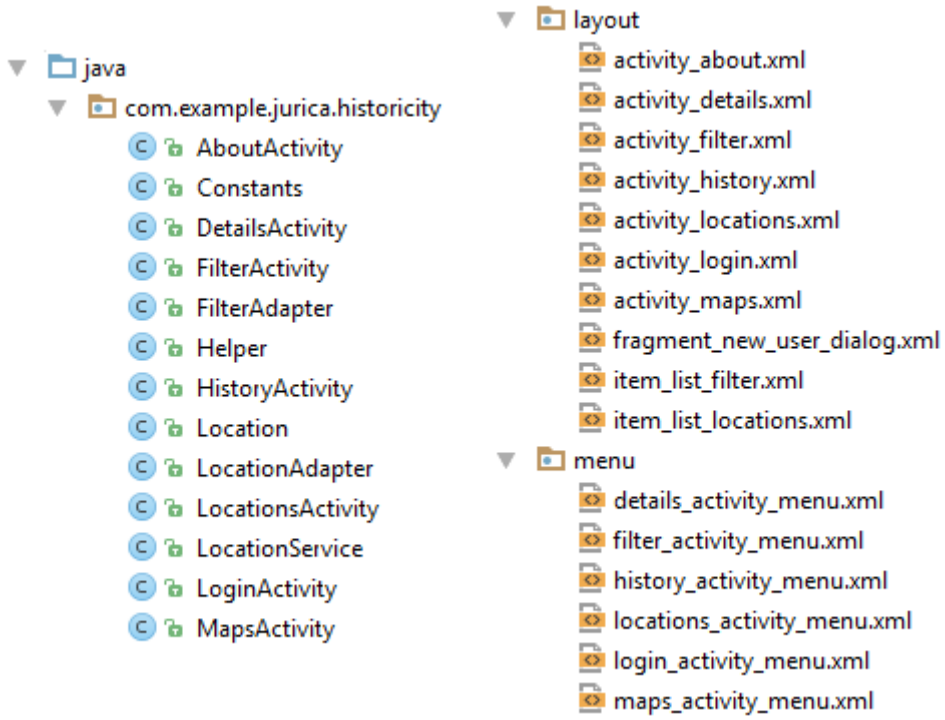
U tablici 12. prikazan je Korisnički slučaj 11 koji opisuje scenarij za prikazivanje povijesti prethodno posjećenih lokacija.

Tablica 12. – Korisnički slučaj 11

ID slučaja	KS11
Ime	Povijest prethodno posjećenih lokacija
Opis	Korisnik može pregledati svoju povijest prethodno posjećenih lokacija
Preduvjet	Korisnik je prijavljen
Glavni scenarij	1. Korisnik prima obavijest o približavanju lokaciji 2. Sustav dodaje lokaciju u povijest prethodno posjećenih lokacija 3. Korisnik otvara povijest 4. Sustav prikazuje povijest prethodno posjećenih lokacija

4.2. Dizajn i funkcionalnost programskog rješenja *HistoriCity*

Izrađena Android mobilna aplikacija koja obuhvaća sve zahtjeve korisnika sastavljena je od sedam Aktivnosti, dva prilagodnika, Servisa, klase koja opisuje lokaciju, klase u kojoj se nalaze sve konstante i klase sa statičnim metodama koje se koriste u više od jedne Aktivnosti. Svaka Aktivnost ima datoteku kojom je opisan njen izgled (engl. *layout*) i uz to su stvoreni meniji kojima je opisan izgled radne trake (engl. *action bar*). Na slici 4.1 prikazani su svi modeli i datoteke za izgled.



Slika 4.1. Modeli i datoteke za izgled

Sedam aktivnosti označavaju sedam mogućih izgleda zaslona na kojima se odvija interakcija s korisnikom, a to su: Aktivnost za prijavu, Aktivnost za popis lokacija, Aktivnost za filtriranje

lokacija, Aktivnost za prikaz detalja o lokaciji, Aktivnost sa Google kartom, Aktivnost za prikaz povijesti korisnika i Aktivnost u kojoj su prikazane informacije o programeru.

Aktivnost za prijavu je prva aktivnost koja se pojavljuje nakon otvaranje mobilne aplikacije. Svrha Aktivnost je prijavljivanje korisnika na njegov korisnički račun. Izgled Aktivnosti za prijavu se sastoji od dva pogleda u kojima korisnik unosi korisničko ime i lozinku, tipka za potvrdu prijave i tipka za stvaranje novog korisničkog računa. Za stvaranje novog korisničkog računa otvara se novi prozorčić čiji je izgled stvoren pomoću oblikovnog obrasca Graditelja.

U Aktivnosti za popis lokacija nalazi se lista lokacija. Svaka lokacija je prikazana u jednom članku pomoću jednog od prilagodnika aplikacije. Prilagodnik je oblikovni obrazac koji povezuje sučelje klase koja predstavlja lokaciju i sučelje liste lokacija, točnije jednog članka liste. Prilagodnik preuzima podatke o lokaciji iz objekta i prema određenom izgledu predstavlja podatke unutar članka. U članak se tako postavlja slika lokacije, njeno ime, opis i grad u kojem se nalazi. Osim liste lokacija, Aktivnost sadrži i radnu traku na kojoj se nalaze tipke za prelazak u druge Aktivnosti. Aktivnost za popis lokacija ima i dva asinkrona zadatka. Asinkronim zadatkom se omogućava izvršavanje koda sa posebnom niti, odnosno kod neće izvršavati glavna nit za korisničko sučelje. Na taj način se odvija povezivanje aplikacije s poslužiteljem gdje se koriste oblikovni obrasci Dekorator i Graditelj. Obrazac Dekorator koristi se kod ulaznog toka podataka s poslužitelja. Dekorator omogućava klasi *InputStream* čitanje bilo kojeg tipa podataka, u ovom slučaju niza znakova (Sl.4.2). Pročitani niz znakova se zatim spaja u cjelinu pomoću oblikovnog obrasca Graditelja. Asinkronim zadacima preuzimaju se lokacije i popis svih gradova koji se prikazuje u Aktivnosti za filtriranje lokacija.

```
DataOutputStream wr = new DataOutputStream(
        connection.getOutputStream());
wr.writeBytes(parameters);
wr.flush();
wr.close();

//Read line by line and save in JSONString
InputStream in = connection.getInputStream();
StringBuilder sBuilder = new StringBuilder();
BufferedReader bReader = new BufferedReader(new InputStreamReader(in, Charset.forName("ISO8859-2")));
String line;
while((line = bReader.readLine()) != null){
    sBuilder.append(line);
}
String JSONString = sBuilder.toString();
```

Slika 4.2. Prikaz izvornog koda vezanog uz Dekorator i Graditelj

Aktivnost za filtriranje lokacija sadrži gradove koje korisnik može označiti. Pri povratku u Aktivnost za popis lokacija bit će prikazane samo lokacije koje se nalaze u označenim gradovima. Gradovi u Aktivnosti za filtriranje lokacija prikazuju se pomoću obrasca Prilagodnika. U ovom slučaju Prilagodnik omogućuje komunikaciju između sučelja liste gradova i niza gradova koji su preuzeti s poslužitelja. Osim liste gradova, Aktivnost sadrži i radnu traku s kojom je moguće označiti ili odznačiti sve gradove i potvrditi kraj označavanja i povratak u Aktivnost za popis lokacija.

Prikaz detalja o lokaciji odvija se u Aktivnosti za prikaz detalja. Aktivnost se sastoji od imena lokacije, grada i adrese na kojoj se lokacija nalazi, kratkog opisa lokacije i slike. Iz radne trake moguće je prikazati lokaciju na Google karti bez pokretanja Servisa za osluškivanje lokacija.

Aktivnost sa Google kartom sadrži kartu koja je povezana sa Google API-jem. API omogućuje pristup Googleovim podacima koji se prikazuju na karti u Aktivnosti. Nakon otvaranja Aktivnosti sa Google kartom iz Aktivnosti za popis lokacija pokreće se Servis za oskuškivanje lokacija. Servis je pokrenut u pozadini i nastavlja sa radom iako aplikacija prestane biti u korisnikovom fokusu. Servis uspoređuje trenutnu lokaciju korisnika sa lokacijama sa popisa i ukoliko udaljenost između lokacija bude manja od sto metara, Servis će obavijestiti korisnika o približavanju lokaciji. Za stvaranje izgleda obavijesti i upozorenja koristi se oblikovni obrazac Graditelj. Prema slici 4.3 pomoću Graditelja postavlja se naslov i poruka upozorenja, te postupci nakon pritiska pozitivne i negativne tipke. Graditelj zatim stvara upozorenje prema uputama i tada se upozorenje može prikazati korisniku.

```
AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(this);
alertDialogBuilder
        .setTitle(Constants.SERVICE_DIALOG_TITLE)
        .setMessage(Constants.SERVICE_DIALOG_MESSAGE)
        .setPositiveButton(Constants.SERVICE_DIALOG_POSITIVE, (dialog, which) -> {
                stopService(new Intent(LocationsActivity.this, LocationService.class));
                dialog.dismiss();
        })
        .setNegativeButton(Constants.SERVICE_DIALOG_NEGATIVE, (dialog, which) -> {
                dialog.cancel();
        });

AlertDialog stopServiceDialog = alertDialogBuilder.create();
stopServiceDialog.show();
```

Slika 4.3. Prikaz izvornog koda vezanog uz Graditelja

Aktivnost za prikaz povijesti sadrži listu u kojoj se prikazuju lokacije koje je korisnik posjetio. Popis lokacija preuzima se iz Dijeljenih postavki (engl. *Shared Preferences*) i

prikazuju se u listi pomoću jednostavnog Prilagodnika koji je implementiran u Android alate za razvoj programske podrške. Na slici 4.4 prikazano je povezivanje Prilagodnika sa listom lokacija. Potrebno je stvoriti objekte liste i Prilagodnika i zatim predati listi Prilagodnik koji će popunjavati članke sa podacima. Pri stvaranju Prilagodnika potrebno je odrediti izgled jednog članka liste i predati Prilagodniku podatke koji će se prikazivati. Osim oblikovnog obrasca Prilagodnika, u ovoj Aktivnosti ponovno se koriste i obrasci Dekorator i Graditelj. Korisnik može odabrati lokaciju iz liste i otvoriti detalje o lokaciji. Podaci o lokaciji se preuzimaju s poslužitelja na identičan način kao i u Aktivnosti za popis lokacija.

```
listView = (ListView) findViewById(R.id.listView);
adapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, historyArray);
listView.setAdapter(adapter);
adapter.notifyDataSetChanged();
```

Slika 4.4. Prikaz izvornog koda vezanog uz Prilagodnik

Klasa Lokacija predstavlja lokaciju koja ima povijesnu priču. Atributi klase su ID, ime, adresa, grad, opis i URL na kojem se nalazi slika lokacije (Tablica 13.). Klasa sadrži metode za dohvaćanje i promjenu atributa. Lokacija se na karti prikazuje pomoću metode *getFromLocationName()* klase Geocoder koja dohvaća geografsku širinu i duljinu prema adresi lokacije. Slika lokacije nalazi se na URL-u i dohvaća se pomoću biblioteke Universal Image Loader. Biblioteka ima implementirane metode koje olakšavaju preuzimanje slika s Interneta i prikazivanje slika u Pogledima u Android aplikaciji.

Tablica 13. – Atributi klase Lokacija

Atribut	Tip
ID	Cijeli broj
Ime	Niz znakova
Adresa	Niz znakova
Grad	Niz znakova
Opis	Niz znakova
Slika (URL)	Niz znakova

Osim spomenutih oblikovnih obrazaca za izradu aplikacije koristio se i Jedinstveni objekt. Obrazac Jedinstveni objekt koristi se u svakoj Android mobilnoj aplikaciji na klasi Application. Klasa Application je bazna klasa za održavanje globalnog stanja aplikacije i instancira se prije bilo koje druge klase u aplikaciji. Oblikovni obrazac omogućava da postoji samo jedan objekt takve klase, jer stvaranje više od jednog objekta dovodi do problema. Aktivnosti i Servisi stvaraju se pomoću oblikovnog obrasca Apstraktne tvornice, a stanje u kojom se svaka Aktivnost nalazi se pamti pomoću obrasca Uspomene. Unutar svih prethodno navedenih Prilagodnika koriste se oblikovni obrasci Iterator i Promatrač. Iterator omogućuje

prolazak kroz listu elemenata koje Prilagodnik prilagođava za drugo sučelje. Oblikovni obrazac Promatrač koristi se pri pozivanju metode Prilagodnika *notifyDataSetChanged()* koja obavješava sve promatrače da je došlo do promjene u podacima koje Prilagodnik prenosi. Posljednji korišteni oblikovni obrazac je Kompozit čije je korištenje prikazano kodom na slici 4.5. Nakon pritiska na bilo koji pogled u aplikaciji poziva se metoda *onClick()*. Metoda *onClick()* prima samo jedan parametar i to Pogled. Zbog Kompozita je u mogućnosti primiti bilo koji Pogled, odnosno ignorirati razlike među Pogledima i funkcionirati jednako za svaki.

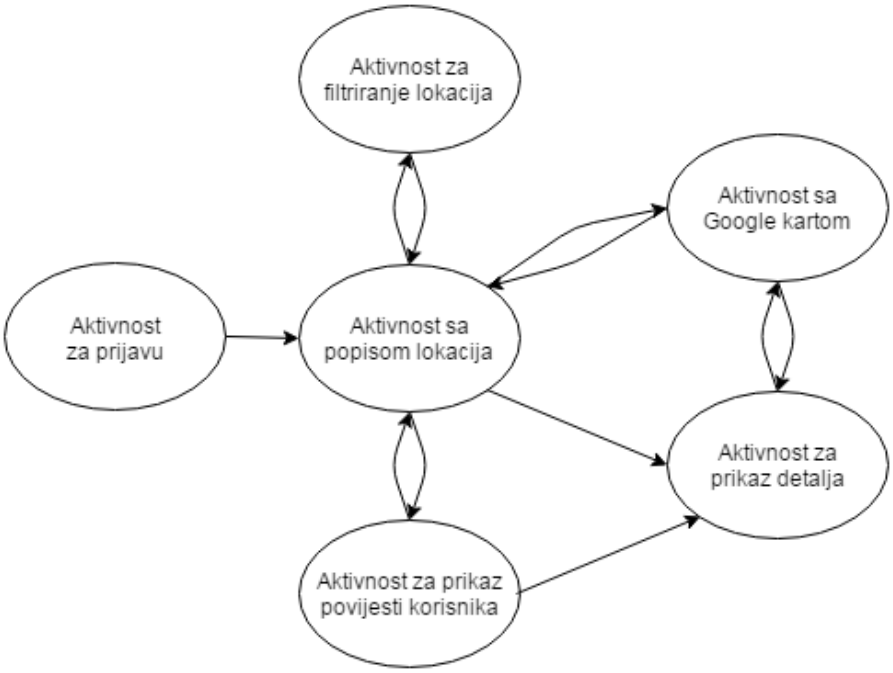
```
bCancel.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        login.cancel();
    }
});
```

Slika 4.5. Prikaz izvornog koda vezanog uz Kompozit

Uz Android mobilnu aplikaciju izrađeno je i web sučelje za dodavanje novih lokacija, baza podataka u koju se lokacije spremaju i API za komuniciranje aplikacije sa bazom podataka.

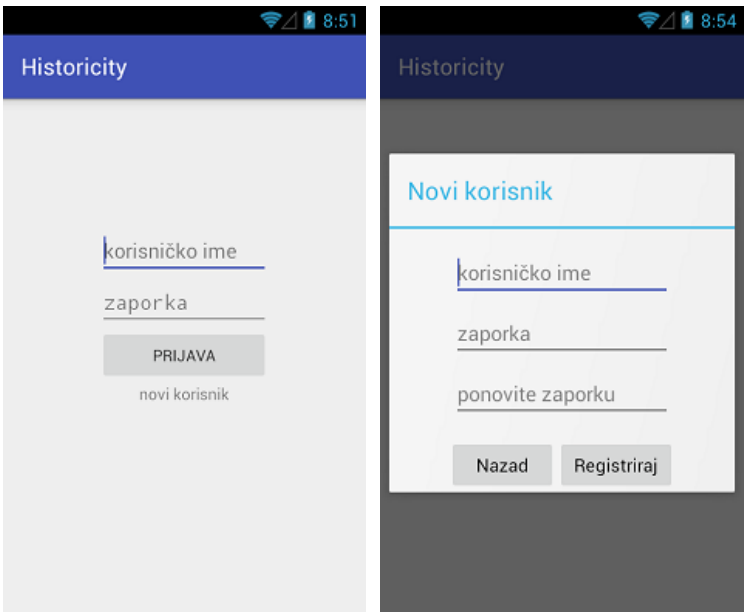
4.3. Opis izrađenog programskog rješenja *HistoriCity*

Na slici 4.6 prikazan je dijagram toka Aktivnosti, odnosno mogućnosti prelaska iz jedne aktivnosti u drugu. Na dijagramu nije prikazana mogućnost odjavljivanja iz korisničkog računa koja je moguća iz svake Aktivnosti i mogućnost korištenja hardverske tipke na Android uređajima koja se koristi za povratak. Rad u aplikaciji započinje u Aktivnosti za prijavu iz koje se prelazi u Aktivnost sa popisom lokacija. Aktivnost sa popisom lokacija je centralna aktivnost i iz nje je moguće pokreniti bilo koju drugu Aktivnost. U Aktivnost za prikaz detalja moguće je doći iz Aktivnosti sa popisom lokacija, za prikaz povijesti korisnika ili iz Aktivnosti sa Google kartom, a iz nje je moguće se vratiti u Aktivnost sa Google kartom kada se otvara samo ta jedna lokacija čiji se detalji prikazuju. Iz Aktivnosti sa popisom lokacija moguće je pokrenuti Aktivnost za filtriranje lokacija, za prikaz povijesti korisnika i Aktivnost sa Google kartom. Isto tako je moguć povratak iz spomenutih Aktivnosti u Aktivnost sa popisom lokacija.



Slika 4.6. Dijagram toka Aktivnosti

Nakon što korisnik otvori mobilnu aplikaciju prikazat će mu se Aktivnost za prijavu u kojoj je potrebno upisati korisničko ime i zaporku (Sl.4.7.a). Ukoliko korisnik krivo unese korisničko ime ili zaporku sustav će ga obavijestiti i bit će potrebno ponoviti upis. Ako korisnik nema korisnički račun ili želi izraditi novi potrebno je pritisnuti „novi korisnik“ i tada će se pojaviti sučelje za izradu novog korisničkog računa (Sl.4.7.b). Za izradu novog računa potrebno je upisati korisničko ime koje se već ne koristi i dva puta upisati identičnu zaporku.

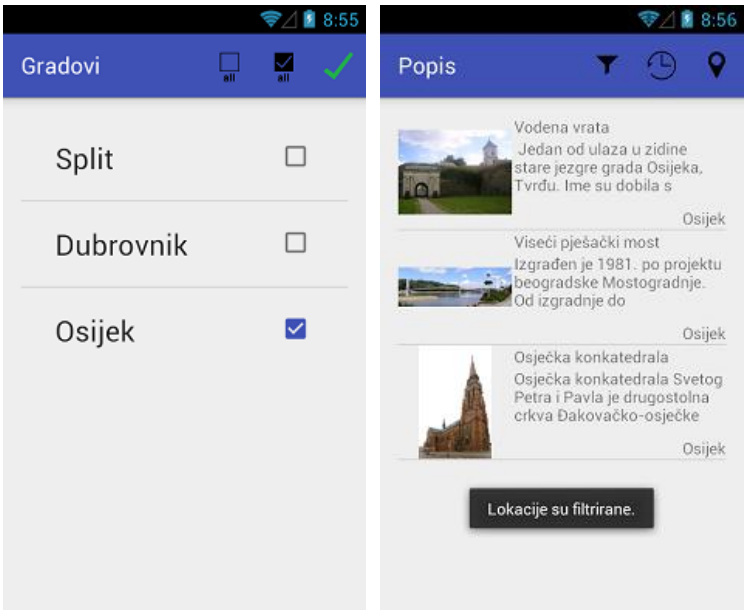


Slika 4.7. Početni zaslon aplikacije; a) Prijava postojećeg korisnika, b) Sučelje za izradu novog korisničkog računa

Nakon prijave, korisnik ulazi u centralnu Aktivnost gdje je prikazan popis lokacija (Sl.4.8). Ako se prvi puta ulazi u aktivnost, sve lokacije se prikupljaju s poslužitelja i učitavaju u listu. Korisnik može osvježiti listu povlačenjem prema dolje i ponovno prikupiti podatke s poslužitelja. Ukoliko korisnik pritisne na jednu od lokacija, otvorit će se nova Aktivnost u kojoj će biti prikazane detaljnije informacije o pritisnutoj lokaciji. Prikazane lokacije u listi moguće je filtrirati prema gradovima odabiranjem filtriranja u radnoj traci. Također, moguće je i prikazati povijest prethodno posjećenih lokacija ili otvoriti Aktivnost sa Google Kartom na kojoj će biti prikazane lokacije sa liste.

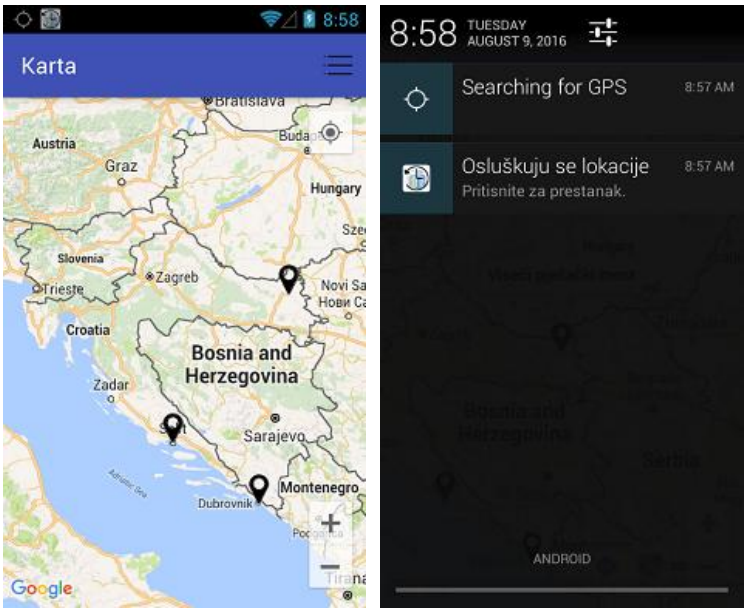
Slika 4.8. Aktivnost s popisom lokacija

Na slici 4.9.a) prikazan je izgled Aktivnosti za filtriranje. Korisnik može označiti gradove u kojima se nalaze lokacije kako bi prikazao u listi samo lokacije iz označenih gradova. Putem radne trake moguće je odabrati da se jednim pritiskom označe ili odznače svi gradovi. Nakon odabira gradova korisnik potvrđuje filtriranje i vraća se u Aktivnost sa popisom lokacija gdje će tada biti prikazani samo lokacije iz odabranih gradova (Sl.4.9.b).



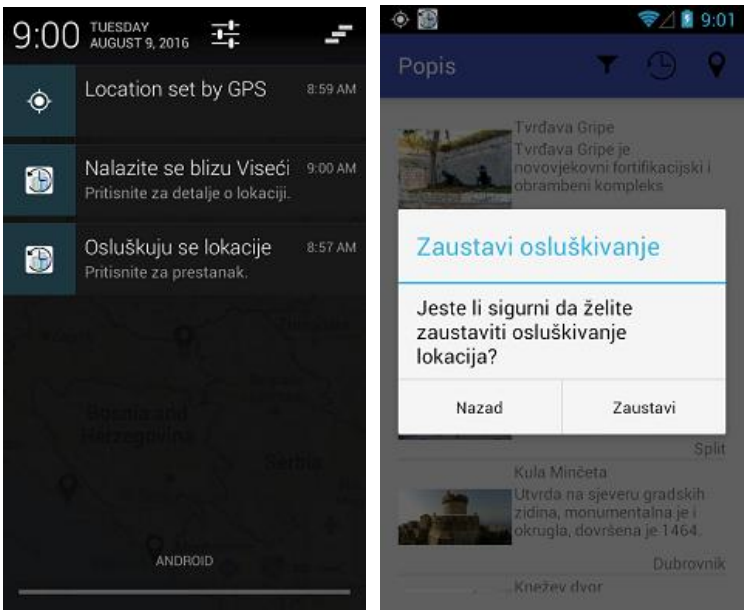
Slika 4.9. Filtriranje lokacija; a) Aktivnost za filtriranje lokacija, b) Povratak na popis lokacija gdje su prikazane lokacije iz označenih gradova

Nakon što korisnik odabere listu lokacija, pritiskom na sliku za kartu u radnoj traci Aktivnost sa popisom lokacija otvaraju se lokacije u novoj Aktivnosti i prikazuju na Google Karti (Sl.4.10.a). Google Kartu je moguće povećati ili smanjiti pomoću predviđenih tipki. Korisnik može i pritisnuti na simbol koji predstavlja lokaciju. Tada će se prikazati prozorčić u kojem će pisati ime lokacije. Ponovnim pritiskom na simbol otvorit će se Aktivnost za prikaz detalja u kojoj će biti prikazane detaljnije informacije o lokaciji. Pri otvaranju Aktivnosti sa Google Kartom iz Aktivnosti sa popisom lokacija pokreće se i Servis koji osluškuje lokacije (Sl.4.10.b).



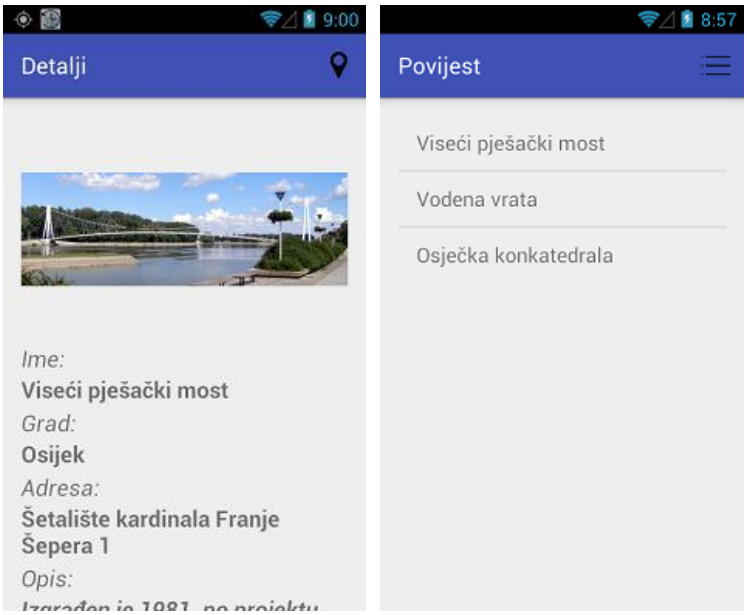
Slika 4.10. Prikaz lokacija na Google Karti; a) Lokacije su označene na karti, b) Servis za osluškivanje lokacija

Servis za osluškivanje lokacije nakon promjene lokacije korisnika uspoređuje korisničku lokaciju sa lokacijama prikazanim na Google Karti. Ukoliko je udaljenost između dvije lokacije manja od sto metara, korisnik će primiti obavijest (Sl.4.11.a). Pritiskom na nalijepljenu obavijest (engl. *sticky notification*) korisnik može zaustaviti osluškivanje lokacija (Sl.4.11.b). Nakon toga korisnika će se odvesti u Aktivnost sa popisom lokacija gdje će moći ponovno izabrati lokacije koje želi prikazati na Google Karti i osluškivati pomoću novog Servisa.



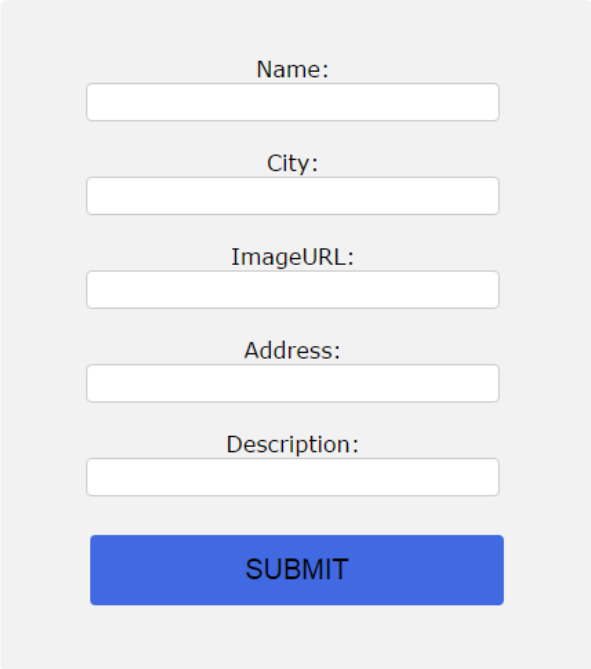
Slika 4.11. Osluškivanje lokacija korisnika; a) Obavijest o lokaciji u blizini, b) Prekid osluškivanja lokacija

Već spomenuta Aktivnost sa detaljima o lokaciji prikazana je na slici 4.12.a). Korisniku se prikazuje slika lokacije, njeno ime, grad i adresa na kojoj se nalazi i kratak opis u kojem je opisana povijest lokacije. Pritiskom na simbol na radnoj traci otvorit će se Aktivnost sa Google Kartom gdje će biti prikazana samo lokacija čiji su detalji bili prikazani. Na slici 4.12.b) prikazana je Aktivnost u kojoj su ispisane prethodno posjećene lokacije. Svaki korisnik ima svoju listu posjećenih lokacija. Lokacije se dodaju u povijest nakon što Servis pronađe novu lokaciju u blizini korisnika.



Slika 4.12. Detalji i povijest lokacija; a) Aktivnost za prikaz detalja, b) Aktivnost za prikaz povijesti korisnika

Na slici 4.13 je prikazan dio web sučelja pomoću kojeg se dodavaju nove lokacije. Korisnik bi trebao ispuniti formu i spremiti. Lokacija će tada biti spremljena u bazu podataka i svi korisnici mobilne aplikacije moći će ju vidjeti u svojoj aplikaciji.



Slika 4.13. Web sučelje za dodavanje novih lokacija

4.4. Testiranje mobilne aplikacije *HistoriCity*

Izrađena Android mobilna aplikacija tokom razvoja testirana je na stvarnom Android mobilnom uređaju Samsung Galaxy S2 koji sadrži zaslon razlučivosti 480x800 piksela i Android verziju 4.1.1.. Uz stvarni uređaj korišten je i virtualni uređaj sa zaslonom razlučivosti 768x1280 piksela sa Android verzijom 4.1.1. kako bi se aplikacija mogla testirati na različitim veličinama zaslona. Zbog nedostatka računalne opreme i uređaja nije bilo moguće testiranje aplikacije na novijim verzijama Androida. Povezivanjem uređaja sa razvojnim okruženjem Android Studio omogućuje prepoznavanje kvarova prilikom korištenja aplikacije. Nakon događaja koji prethodi kvaru, Android Studio prikazuje linije programa koji su se izvršavale prilikom izvođenja aplikacije i olakšavaju programeru popravak kvara.

Tokom testiranja korisničkih slučajeva tijekom izrade mobilne aplikacije uočeni su problemi koji su zatim ispravljeni. Popis ključnih problema, njihov opis i rješenje problema prikazani su u tablici 14. Posljednja verzija mobilne aplikacije HistoriCity testirana je kroz sve korisničke slučajeve i zaključeno je da se glavni i alternativni scenariji korisničkih slučajeva podudaraju sa izvođenjem aplikacije na uređajima.

Tablica 14. – Ključni problemi pri izradi aplikacije

Problem	Opis	Rješenje
Recikliranje članaka sa Prilagodnikom	Ukoliko korisnik prilikom filtriranja lokacija označi gradove i pomakne popis gradova, gradovi će se u prikazu odznačiti, ali u pozadini ostati označene	Unutar Prilagodnika potrebno je provjeriti je li grad već označen
Tipka za povratak ignorira prijavljivanje korisnika	Nakon što se korisnik odjavi, pomoću tipke za povratak se može vratiti u Aktivnost u kojoj je potrebno biti prijavljen	Korištenjem zastavica potrebno je zatvoriti sve prethodne Aktivnosti
Osluškivanje lokacija samo dok korisnik ima otvorenu Google Kartu	Lokacije se osluškuju samo dok se korisnik nalazi u Aktivnosti sa Google Kartom	Stvaranje Servisa koji u pozadini osluškuje lokacije
Otvaranje Google Karte iz detalja započinje osluškivanje lokacija	Nakon što korisnik otvori samo jednu lokaciju iz Aktivnosti sa detaljima, aplikacija započinje sa osluškivanjem lokacija	Provjera iz koje Aktivnosti je otvorena Aktivnost sa Google Kartom i prema informaciji odlučiti o početku osluškivanja lokacija

Testiranje aplikacije se može poboljšati korištenjem većeg broja korisnika sa različitim Android uređajima i prikupljanjem podataka o kvarovima na koje su korisnici naišli. Iz perspektive oblikovnih obrazaca poboljšanje je moguće dizajniranjem kompleksnije aplikacije u kojoj će se samostalno razvijati dijelovi programske podrške korištenjem oblikovnih obrazaca koji se sada nisu koristili. Mobilnu aplikaciju je moguće nastaviti proširivati vlastitim dizajniranjem Pogleda, dodavanjem animacija, omogućavanjem fotografiranja lokacija i pregledavanja fotografija drugih korisnika.

5. ZAKLJUČAK

Izradom diplomskog rada opisani su najvažniji oblikovni obrasci koje je dokumentirala Družba četvorice. Opisana je uloga pojedinog obrasca, njegova struktura, osobine i predstavljeni su primjeri korištenja. Uz oblikovne obrasce, opisana su i načela objektno orijentiranog dizajna i anti-obrasci. Prikazani su i često korišteni obrasci pri stvaranju Android aplikacija i izrađena je mobilna aplikacija pri čijoj izradi su korišteni neki od opisanih oblikovnih obrazaca.

Android mobilna aplikacija napisana je korištenjem programskog jezika Java i označnog jezika XML u razvojnom okruženju Android Studio, a testirana je sa virtualnim i stvarnim uređajima. Mobilna aplikacija omogućuje korisniku popis lokacija u odabranom gradu koje imaju povijesnu priču. Korisnik samostalno odabire lokacije koje se prikazuju na Google Karti, a sustav će obavijestiti korisnika ukoliko se korisnik nalazi u blizini jedne od odabranih lokacija. Aplikacija preuzima podatke o lokacijama sa izrađenog web poslužitelja što je ujedno bio i najzahtjevniji zadatak pri izradi aplikacije.

Unaprjeđenje rada moguće je proširivanjem broja opisanih oblikovnih obrazaca jer su u sadašnjem radu opisani samo najvažniji. Također, moguće je i proširivanje sa prikazom koda za svaki obrazac i načinom implementacije u sustav. Android mobilna aplikacija pronalazi mjesta za proširivanje u dodatnim funkcijama i animacijama Pogleda.

LITERATURA

[1] I. Horton, Beginning Java, Wiley, Indianapolis, USA, 2011.

[2] J. McC. Smith, Why programmers need design patterns to communicate effectively, http://www.informit.com/articles/article.aspx?p=2044336, (pristupljeno 13.5.2016.)

[3] J. Maioriello, What are design patterns and do I need them?, http://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm, (pristupljeno 14.6.2016.)

[4] E. Gamma, R. Helm, R. Johnson i J.M. Vlissides, Design patterns – elements of reusable object-oriented software, Addison-Wesley, USA, 1995.

[5] R.C. Martin, Agile software development, principles, patterns, and practices, Pearson, Upper Saddle River, USA, 2002.

[6] Sourcemaking, Design patterns, https://sourcemaking.com/design_patterns, (pristupljeno 14.6.2016.)

[7] R.C. Martin, The little singleton, https://8thlight.com/blog/uncle-bob/2015/06/30/the-little-singleton.html, (pristupljeno 28.8.2016.)

[8] D. Bailey, S.O.L.I.D. software development, one step at a time, http://www.codemag.com/article/1001061, (pristupljeno 29.7.2016.)

[9] Sourcemaking, AntiPatterns, https://sourcemaking.com/antipatterns, (pristupljeno 29.7.2016.)

[10] Antipatterns, AntiPatterns, a brief tutorial, http://antipatterns.com/briefing/index.htm, (pristupljeno 31.7.2016.)

[11] Openhandsetalliance, FAQ, http://www.openhandsetalliance.com/oha_faq.html, (pristupljeno 6.8.2016.)

[12] Google, Dashboards, https://developer.android.com/about/dashboards/index.html, (pristupljeno 6.8.2016.)

[13] M. Gargenta, Learning Android, O'Reilly Media, Sebastopol, USA, 2011.

[14] P.V. Gomez Sanchez, Software design patterns on Android, http://www.slideshare.net/PedroVicenteGmezSnch/software-design-patterns-on-android, (pristupljeno 3.8.2016.)

[15] Vardhan Blog, Overview & mapping of GoF design patterns with Android API's, http://vardhan-justlikethat.blogspot.hr/2013/10/mapping-gof-design-patterns-with.html, (pristupljeno 3.8.2016.)

[16] Oodesign, Proxy pattern, http://www.oodesign.com/proxy-pattern.html, (pristupljeno 5.8.2016.)

[17] A. Leiva, MVP for Android: how to organize the presentation layer, http://antonioleiva.com/mvp-android/, (pristupljeno 28.8.2016.)

SAŽETAK

U ovom su radu opisani glavni oblikovni obrasci. Riječ je o nacrtima rješenja za probleme koji se često pojavljuju pri objektno orijentiranom programiranju. Svaki oblikovni obrazac opisuje svoju ulogu, primjere i način korištenja za rješavanje jednog problema pri čemu se osigurava kvalitetno i dugoročno ispravno rješenje. Uz oblikovne obrasca, prikazana su načela objektno orijentiranog dizajna te anti-obrasci. Načela su temeljne odredbe koje treba imati na umu pri pisanju koda, a anti-obrasci, suprotno od oblikovnih obrazaca, opisuju česte krive načine dizajniranja programske podrške. Kao dio diplomskog rada izrađena je Android mobilna aplikacije čija je svrha omogućiti korisniku aplikacije pregled lokacija u pojedinom gradu koje imaju povijesnu priču. Kroz izradu aplikacije prikazano je korištenje najčešćih oblikovnih obrazaca na Android sustavu kao i prednosti koje oni donose.

Ključne riječi: Android, anti-obrasci, načela, oblikovni obrasci, OOP

ABSTRACT

Design patterns and their application in Android mobile applications development

This thesis describes main design patterns. They represent blueprints of solutions for frequently occurring problems in object oriented programming. Each design pattern describes its role, examples and usage for solving one problem which ensures quality and long-term correct solution. Accompanied by design patterns, principles of object oriented design and anti-patterns are presented. Principles are core terms which should be considered while programming and anti-patterns, contrary to design patterns, describe frequent bad examples of software development. Android mobile application has been developed as the part of thesis with the intent to give user a list of locations in each city that have historical relevance. Development of the application showed usages of most often used design patterns on the Android system and their advantages.

Keywords: Android, anti-patterns, principles, design patterns, OOP

ŽIVOTOPIS

Jurica Pleša rođen je 20. studenog 1992. godine u Osijeku. Nakon završene osnovne škole, 2007. godine upisuje III. gimnaziju u Osijeku. Godine 2011. polaže državnu maturu i upisuje preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku. Obranom završnog rada 2014. godine uspješno završava preddiplomski studij i upisuje diplomski studij Procesno računarstvo na istom fakultetu.

PRILOZI

- Oblikovni obrasci i primjena u razvoju mobilnih aplikacija na Android platformi.docx
- Oblikovni obrasci i primjena u razvoju mobilnih aplikacija na Android platformi.pdf
- Programsko rješenje *HistoriCity*