UNIVERZA V LJUBLJANI
EKONOMSKA FAKULTETA

MAGISTRSKO DELO

RAZVOJ PROGRAMSKE REŠITVE ZA ELEKTRONSKI ŠTEVEC ELEKTRIČNE ENERGIJE S POMOČJO JEZIKA ZA MODELIRANJE UML

Ljubljana, junij 2004 MATEJ FETIH

IZJAVA:

Študent MATEJ FETIH izjavljam, da sem avtor tega magistrskega dela, ki sem ga napisal pod mentorstvom dr. MOJCE INDIHAR ŠTEMBERGER in skladno s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovolim objavo magistrskega dela na fakultetnih spletnih straneh.

V Ljubljani, dne 7.6.2004

Podpis: _____

1 UVOD

Današnja družba je družba hitrih sprememb, ki jih doživljajo organizacije in njihove poglavitne dejavnosti. Če želimo na takšnem trgu prodreti z novo zamislijo ali izdelkom, moramo biti sposobni nov izdelek izdelati v čim krajšem času. Zaradi tega potrebujemo takšna programska orodja, ki nam pomagajo pri skrajšanju razvojnega cikla novega izdelka.

Svet postaja vse bolj kompleksen in uporabniki potrebujejo vedno bolj zahtevne programske rešitve. Zaradi tega potrebujemo tudi močnejša orodja oziroma metodologije, ki so sposobna te probleme reševati. S pojavom objektno orientiranih metodologij smo dobili zelo močno orodje za ustvarjanje zapletenih programskih rešitev. Osnovni problem vseh starejših metodologij je, da ne omogočajo skupne obravnave podatkov, ki predstavljajo statični del, in postopkov, ki predstavljajo dinamični del analiziranega sistema. To povzroča težave z doslednostjo razvoja in preprečuje tipizacijo osnovnih gradbenih blokov sistema, kar povečuje čas in stroške razvoja. Ideja objektnega pristopa je torej povezana z željo, da bi programsko opremo lahko razvijali podobno kot strojno opremo, ki je sestavljena iz standardiziranih sestavnih delov. Tako je možno posamezni sestavni del uporabiti večkrat v različne namene (Kovačič, 1994, str. 48).

Iz pristopa tipiziranih gradbenih blokov je prišla ideja o programski rešitvi sestavljeni iz standardiziranih sestavnih delov, ki bi bili povezani s pomočjo UML[1] poenotenega jezika za modeliranje. V podjetju Iskraemeco d. d. bi pristop uporabili zaradi hitrejšega sledenja zahtevam trga.

Cilj magistrske naloge je bil prikazati celotni cikel razvoja programske rešitve za števec električne energije s pomočjo UML jezika. Pri tem sem dosedanje delo v okviru podjetja razširil na analizo informacijskih potreb in zahtev kupcev in celovito načrtovanje programske rešitve. Pri razvoju programske rešitve so bili uporabljeni nekateri že razviti programskimi gradniki, ki sem jih med razvojem uporabil v programski rešitvi in so omogočili precejšnje skrajšanje razvojnega cikla. Uporabljal sem CASE[2] programsko orodje Rhapsody[3] in C, ki omogoča razvoj in testiranje vgrajenih računalniških sistemov delujočih v realnem času. V nalogi so razvidne vse faze razvoja

[1] ang. Unified Modeling Language
[2] ang. Computer Aided Software Engineering
[3] Rhapsody je izdelek proizvajalca I-Logic®.

programske rešitve od postavljanja zahtev, analize, načrtovanja in na koncu implementacije, kjer dodamo še standardizirane programske gradnike.

Rezultat pristopa je 30 odstotkov krajši čas razvoja izdelka. Največja prednost UML modela pa je neodvisnost programske rešitve od ciljne programske platforme, kar omogoča veliko možnost prenosljivost razvite programske rešitve na kasnejše izpeljanke. Poleg tega pa je mogoče preverjanje obnašanja sistema na Windows programski platformi pred dejansko implementacijo v ciljno programsko platformo. To omogoča možnost iskanja napak že na nivoju načrtovanja in ne šele na nivoju implementacije sistema. Iz UML modela je mogoča avtomatična generacija programske kode[4], ki je pripravljena za uporabo na ciljnem sistemu. Zaradi vpeljave neodvisnih programskih gradnikov programske rešitve je mogoče skupinsko razvijanje, podpora in vzdrževanje izdelka v podjetju. Zahteve za razvoj teh programskih gradnikov morajo biti dobro definirane, razvijalec programske rešitve, ki te gradnike uporablja, ne sme biti tisti, ki njihovo funkcionalnost prilagaja svojemu sistemu. Na takšen način se lahko izgubi neodvisnost programskih gradnikov.

V prvem poglavju je opisana kratka primerjava metodologije življenjskega cikla in objektnih metodologij. V nadaljevanju so navedena dejstva o osnovnih konceptih objektne metodologije, kaj so njeni osnovni gradniki, kako jih uporabljamo, in kakšne so njene prednosti. Drugo poglavje opisuje zgodovinsko ozadje UML poenotenega jezika, njegov namen in kako ga uporabljamo. S konceptualnim modelom prikažemo sestavne dele UML, elemente, povezave in diagrame. Opisan je tudi poenoteni proces razvoja programske opreme. Tretje poglavje je namenjeno procesu razvoja programske rešitve. V okviru procesa razvoja je opisana definicija zahtev, njihova analiza, načrtovanje in implementacija programske rešitve. V fazi analize iz zahtev identificiramo štiri glavne funkcionalnosti elektronskega števca električne energije in priredimo štiri objektne tipe analize. V fazi načrtovanja objektne tipe dodatno dodelamo, tako da dobijo končno obliko. Faza implementacije pa je namenjena samo še implementaciji programske kode, ki bo objektne tipe načrtovanja realizirala. V četrtem poglavju sem opravil SWOT analizo klasičnega pristopa razvoja programske rešitve za vgrajene računalniške sisteme in razvoja programske rešitve s pomočjo UML jezika, ter podal primerjavo obeh razvojnih procesov. Na koncu sledi zaključek, ter navedba literature in virov uporabljenih v magistrskem delu.

[4] Generacija programske kode v različnih programskih jezikih C, C++ in Java.

2 OSNOVNI KONCEPTI OBJEKTNO ORIENTIRANE METODOLOGIJE

V začetku poglavja bom govoril o metodologijah za načrtovanje in gradnjo informacijskega sistema, gre predvsem za primerjavo metodologije življenjskega cikla in objektno orientiranih metodologij. Nato pa bom skušal predstaviti osnovne koncepte objektno orientiranega programiranja in metodologije brez katerih objektni pristop ne bi mogel obstajati.

2.1 METODOLOGIJE RAZVOJA PROGRAMSKE REŠITVE

Metodologija formalno določa proces, ki vključuje zbiranje, analizo zahtev in načrtovanje programske rešitve, ki ustreza tem zahtevam. Obstaja veliko število različnih metodologij, katere se med seboj razlikujejo v načinu, kako pristopimo problemu. Metodologije za razvoj velikih in obširnih programskih rešitev v velikih korporacijah se razlikujejo od tistih, ki se uporabljajo za razvoj vgrajenih računalniških sistemov. Same metodologije pa se razlikujejo tudi od samega pristopa k projektu. Razvojni proces, pri katerem sodeluje veliko število načrtovalcev in programerjev, zagotovo uporablja drugačno metodologijo kot proces pri katerem je vključen samo en razvijalec programske opreme.

2.1.1 METODOLOGIJA ŽIVLJENJSKEGA CIKLA

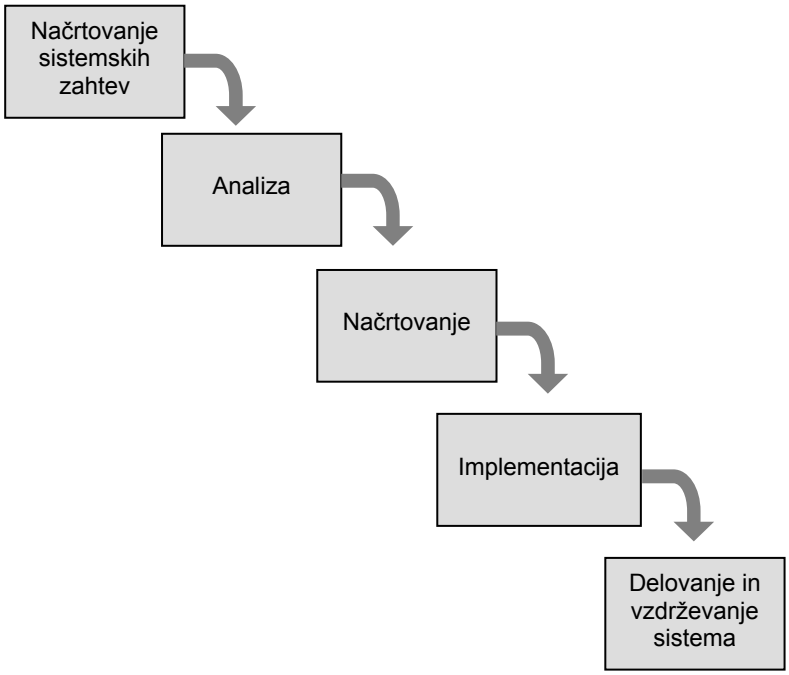
Metodologijo življenjskega cikla sestavlja niz zaporednih faz, za katere je značilna zaporednost njihovega izvajanja. Po koncu vsake faze je potrebno izdelati poročilo in šele nato se lahko začne nova faza (Kovačič, Vintar, 1994, str. 44). Dobre lastnosti klasičnega razvojnega cikla so enostavnost, splošno poznavanje procesa med razvijalci programske opreme in preverjanje aktivnosti na prehodu v naslednjo aktivnost. Ena največjih slabosti pa je težavno planiranje zahtev pri bolj kompleksnih sistemih. Zaporedno izvajanje aktivnosti temelji na zaključenih predhodnih aktivnostih, kar pa je velikokrat nemogoče, saj se v razvojnih projektih zahteve velikokrat spreminjajo. Ovrednotenje delovanja celotnega sistema nastopi šele proti koncu njegovega življenjskega cikla, kar lahko precej podaljša njegov razvojni cikel.

Modeliranja v metodologiji življenjskega cikla skorajda ni, problematika je opisana z analitičnimi in dokumentacijskimi tehnikami, ali pa imamo opravka z verbalnimi opisi.

Metodologija vključuje naslednje faze (Shelly et al., 1998, str. 17), ki so prikazane na sliki 1:

- Načrtovanje zahtev za informacijski sistem se začne s pisanjem sistemskih zahtev, ki opisujejo želeno delovanje informacijskega sistema ali opisujejo potrebne spremembe že obstoječega informacijskega sistema. Rezultat te faze je preliminarno poročilo, ki je zelo pomembno, saj so od njega odvisne vse naslednje faze razvojnega procesa. Preliminarno poročilo[5] imenujemo tudi študij izvedljivosti[6], ki ga sestavljajo ekonomski, tehnični in operacijski faktorji.

Slika 1: Model življenjskega cikla



Vir: Jacobson et al., 1994, str.71

- Analiza sistema se začne s procesom določanja zahtev, ki temelji na preučevanju funkcionalnosti obstoječega informacijskega sistema in identifikaciji novih funkcionalnosti. Proces zbiranja zahteve, njihove analize in določitve načrta reševanja problemov za dejanski informacijski sistem, imenujemo analiza zahtev. Rezultat faze je dokument sistemskih zahtev[7], ki vsebuje vodstvene, uporabniške zahteve in tudi načrt stroškov projekta.
- Namen faze načrtovanja sistema je razvoj informacijskega sistema, ki ustreza vsem dokumentiranim zahtevam. Med načrtovanjem določimo, kaj mora sistem početi in kako bo to počel. Potrebno je določiti vhode in izhode sistema, datoteke, s katerimi bo sistem

[5] ang. Preliminary investigation report
[6] ang. Feasibility study
[7] ang. System requirements study

upravljal, aplikacijske programe in postopke rokovanja z njim. Postopke načrtovanja dokumentiramo v opisu načrtovanja sistema[8].

- V fazi implementacije sistema pišemo, testiramo in dokumentiramo programske rešitve. Največkrat je na koncu faze implementacije potrebno narediti določene prilagoditve in konfiguracije sistema, saj je cilj faze dostava popolnoma delujočega in dokumentiranega informacijskega sistema. Implementacija sistema vključuje še ovrednotenje sistemske implementacije, ki ugotavlja pravilnost operativnega in funkcionalnega delovanja sistema.
- Po implementaciji začne podjetje uporabljati informacijski sistem v poslovne namene. Med fazo delovanja in podpore sistema so nenehno prisotne tudi vzdrževalne spremembe in izboljšave sistema.

2.1.2 OBJEKTNO ORIENTIRANE METODOLOGIJE

Največji problem metodologije življenjskega cikla je počasna prilagodljivost spremembam okolja. Potrebno se je zavedati dejstva, da je sodobna informacijska tehnologija bolj zmogljiva od tiste pred tridesetimi leti, ko je nastajala metodologija življenjskega cikla.

Zgodovinsko gledano so se objektno orientirane metodologije začele pojavljati vzporedno z objektno orientiranim programiranjem, izraz programiranje pa predstavlja generacijo programske kode. Razvoj se je nadaljeval objektno z orientiranim načrtovanjem, nato pa se je začela pojavljati tudi objektno orientirana analiza. Izraz objektno orientirana metodologija združuje vse tri dejavnosti in vključuje celotno filozofijo razvoja sistema od analize zahtev, načrtovanje sistema ali baze podatkov, programiranja do testiranja delovanja sistema. Glavne prednosti objektnega pristopa sta ponovna uporabnost in nadgradnja programske kode. Objektno orientiran sistem lahko hitro sestavimo s pomočjo že napisanih programskih komponent, ali pa s temi komponentami nadgradimo že obstoječ sistem. Ponovna uporabnost se ne nanaša samo na programsko kodo, ampak tudi na dokumente, v katerih se nahaja opis načrtovanja in analize sistema (Graham, 1994, str. 3).

Razvoj programskih rešitev lahko razdelimo na tri glavne aktivnosti:

- *analiza* zajema pisanje specifikacij in logično modeliranje,
- *načrtovanje* vključuje arhitekturno modeliranje,
- med *implementacijo* štejemo kodiranje in testiranje.

Kljub takšni delitvi[9] pride v realnem svetu velikokrat do prepletanja specifikacije in implementacije. Objektno orientirana analiza, načrtovanje in celo implementacija lahko z uvedbo dela na konceptualnem modelu skozi celotni življenjski čas sistema pridobijo na sledljivosti napak povzročenih pri razvoju sistema.

[8] ang. System design specification
[9] Med implementacijo večkrat pride do dodatnih zahtev za programsko rešitev, zaradi česar potrebujemo ponovno analizo problematike.

Objektno orientirana metodologija mora vsebovati sledeče komponente (Graham, 1994, str. 9):

- celotni proces razvoja za poslovne in za tehnološke programske rešitve,
- nabor konceptov in neodvisnih modelov,
- zbirko pravil in navodil,
- uporabno notacijo,
- podporo zunanjih orodij za animacijo,
- nabor že preizkušenih tehnik, uporabnih standardov in testnih strategij,
- navodila za projektni management in zagotavljanje kvalitete,
- nasveti za upravljanje s programskimi knjižnicami in za ponovno uporabo kode.

Sodoben razvoj programske opreme je vedno bolj objektno orientiran. Temelj metodologije sta objekt in razred. Objekt si lahko predstavljamo kot stvar vzeto iz zbirke rešitev, razred pa je nabor vseh rešitev. Vsak objekt opisuje njegovo ime, ki ga ločuje od ostalih objektov, stanje v katerem se objekt nahaja in njegovo obnašanje. Objektno orientirane metodologije poizkušajo objekte iz realnega sveta preslikati v model programske rešitve ali model informacijskega sistema. Shematičen opis modela naredimo s pomočjo naslednjih pojmov: objekt, razred, ograjevanje, sporočilo, mnogoličnost in dedovanje.

2.1.2.1 PREDNOSTI OBJEKTNO ORIENTIRANE METODOLOGIJE

Prednosti objektno orientiranih metodologij so podobne prednostim metod strukturiranega programiranja in načrtovanja, le da so tukaj nekatere prednosti sistema še bolj poudarjene (Graham, 1994, str. 17):

- Dobro načrtovani objekti lahko omogočajo načrtovanje sistemov iz ponovno uporabljivih modulov, kar nas vodi k večji produktivnosti.
- Prehodi med fazami razvoja sistema so lažji, na primer metodologija omogoča lažjo nadgradnjo razredov analize v razrede načrtovanja.
- Ponovna uporaba že obstoječih razredov, ki so že bili testirani v preteklih projektih, vodi k povečanju kakovosti sistema in zmanjšanju števila hroščev.
- Pošiljanje sporočil med objekti v precejšnji meri poenostavi komunikacijo med moduli in zunanjim sistemom. Ta lastnost omogoča tudi lažje postavitve grafičnega vmesnika.
- Skrivanje informacij, podatkov omogoča graditev varnejših sistemov.
- Delitev sistemov na bazo ograjenih objektih tipov pripomore k hitrejšemu izvajanju programja. Hitrost izvajanja ne pada eksponentno z velikostjo projekta in njegovo kompleksnostjo.
- Objektno orientiran pristop je orodje za premagovanje kompleksnosti.

2.2 OBJEKT

Objekt je osnovni gradnik pri zasnovi, načrtovanju ali pri programiranju. Instance so primerki objektov, ki imajo iste lastnosti in jih lahko organiziramo v razrede. Objekt lahko označimo tudi kot entiteto (Grad, Jaklič, 1996, str. 76), ki je sposobna shraniti stanje (informacije) in ima na razpolago operacije (obnašanje), s katerimi lahko vplivamo na to stanje. Opišemo ga s poljubnim številom operacij in s stanjem, ki pomni vpliv teh operacij. Objektno orientiran model je model sestavljen iz več objektov, ki so med seboj popolnoma ločeni (Jacobson, 1994, str. 49).

2.3 RAZRED IN INSTANCE

Razred predstavlja vzorec za opis več objektov in opisuje notranjo sestavo teh objektov. Objekti istega razreda imajo enake definicije operacij in podatkovnih struktur. Nekateri avtorji razred imenujejo podatkovni tip določenega objekta, vendar pa je izraz razred širši, saj poleg njegove strukture opisuje tudi njegov odziv.

V objektno orientiranih sistemih vsak objekt pripada nekemu razredu. Objekt, ki pripada določenemu razredu, imenujemo instanca. Razred opisuje strukturo instance (obnašanje in informacijo o stanju), trenutno stanje instance pa je določeno z operacijami izvedenimi na instanci.

2.4 OGRAJEVANJE

Ograjevanje je lastnost objekta, da lahko vidimo, dosegamo in spreminjamo njegove podatke in lastnosti le skozi njegove operacije (Grad, Jaklič, 1996, str. 121). Ograjevanje podatkov o objektu temelji na tem, da vsak objekt vsebuje vse podatke, ki jih potrebuje za svoje delovanje, poleg tega pa nobenemu objektu ni potrebno vedeti za notranjo strukturo podatkov drugega objekta. Skriti ali ograjeni deli objekta so njegova implementacija ter se imenujejo privatni podatki, vidni podatki in operacije pa so javni podatki objekta.

2.5 SPOROČILA

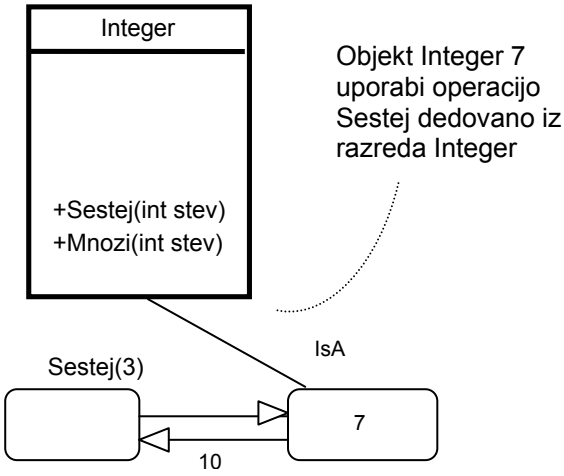
Objekti, razredi in njihove instance komunicirajo s pomočjo sporočil. Torej, če želimo dobiti podatke od nekega objekta v objektno orientiranem sistemu, mu moramo poslati sporočilo. Samo sporočilo je sestavljeno iz treh delov:

- Naslov objekta ali objektov kamor mora biti sporočilo poslano.
- Ime operacije ali izbirnik, ki naj se izvede nad objektom.
- Podatki ali parametri, ki so potrebni pri nekaterih operacijah.

2.6 MNOGOLIČNOST

Mnogoličnost ali polimorfizem pomeni, da se objekti različnih razredov na isto zahtevo odzovejo različno (Grad, Jaklič, 1996, str. 121). Pošiljatelju zahteve ni potrebno poznati, kateremu razredu pripada objekt, ki zahtevo sprejme, vedeti mora samo to, ali je drug objekt zahtevo sposoben sprejeti. Mnogoličnost omogoča povezave različnih objektov med seboj, ki lahko pripadajo različnim razredom.

Slika 2: Način uporabe mnogoličnosti



Vir: Booch et. al.,1999, str.106

2.7 DEDOVANJE

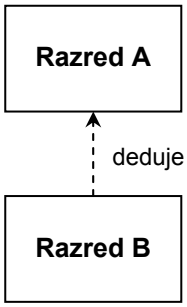
Objekti dedujejo lastnosti razredov katerim pripadajo, v objektno orientiranih sistemih pa lahko tudi razredi pridobijo lastnosti nadrazredov. Kadar ima veliko razredov številne medsebojne podobnosti, lahko skupne lastnosti razredov zberemo v nek skupni nadrazred. Razredi postanejo dedovani iz skupnega nadrazreda. Zaradi prenašanja lastnosti med razredi je dedovanje bistvo filozofije ponovne uporabljivosti kode v industriji programske opreme (Jacobson, 1994, str. 58).

Kadar dodajamo nove lastnosti k programski rešitvi, te lastnosti dodajamo v nadrazred, saj se tako zaradi dedovanja spremenijo lastnosti v celotni programski rešitvi. S takšnim načinom dela se lahko izognemo velikemu številu napak, ki nastanejo zaradi nedoslednega spreminjanja lastnosti razredov.

Iz več razredov lahko s postopkom **generalizacije** izločimo in povežemo njihove skupne lastnosti v novem razredu, ter ga postavimo višje v hierarhiji dedovanja. Nasprotni proces pa je **specializacija**, ko iz razreda vzamemo njegove lastnosti, katerim v novem razredu dodamo še nekaj njegovih posebnih lastnosti. Takšen razred je po hierarhiji nižje od prvotnega razreda.

Razrede, ki ležijo pod višjim razredom v hierarhiji dedovanja, imenujemo **potomci**. Razrede po hierarhiji ležeče nad nižjimi razredi pa imenujemo **predniki**. Uporaba izraza podrazred je ekvivalentna izrazu potomec, nadrazred pa predniku.

Slika 3: Razred B je potomec razreda A in razred A je prednik razreda B



Ker lahko postopek dedovanja poteka preko večih razredov, je smiselno poudariti neposredno povezavo med dvema razredoma. Če se razred deduje direktno iz predhodnega razreda, ta razred imenujemo neposredni potomec, neposrednega prednika pa imenujemo starš.

2.7.1 UPORABA DEDOVANJA

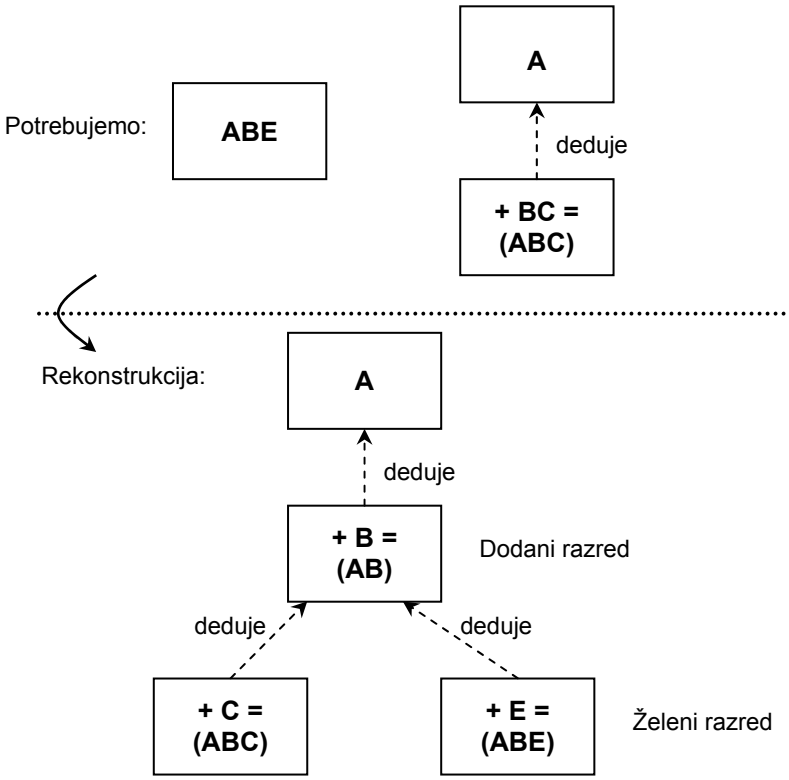
Namen uporabe dedovanja:

- Ponovna uporaba programske kode je najpogostejši razlog uporabe dedovanja razredov.
- Uporaba potomcev namesto prednika na vseh mestih, na katerih je bil uporabljen prednik. Govorimo o kompatibilnosti obnašanja med dedovanimi razredi.
- Specializacija pomeni izpeljavo oziroma spremembo razreda tako, da njegovo obnašanje ni več enako obnašanju prednika.
- Konceptualna uporaba dedovanja se ujema s intuitivno semantiko. Gre za zamenjavo pomenskih objektov z dedovanimi pomenskimi objekti. Npr.: Hiša je zgradba. Lahko zamenjamo zgradbo s hišo, saj hiša vsebuje vse karakteristike zgradbe.

Kako najdemo pravi razred v okviru hierarhije dedovanja:

- Iščemo navzgor po hierarhiji dokler ne najdemo najprimernejšega razreda.
- Uvedemo nov razred neodvisen od hierarhične strukture.
- Rekonstruiramo hierarhične strukture dedovanja, tako da dobimo razred, ki ustreza našim željam.
- Ponovno definiranje lastnosti razreda je zelo enostavno in uporabno, vendar pa lahko porušimo razredno hierarhijo, saj imajo lahko operacije z istim imenom v različnih razredih drugačen pomen.

Slika 4: Rekonstrukcija hierarhije dedovanja je ena od možnosti kako dobiti želene lastnosti razreda, vendar pa velikokrat zahteva veliko preurejanja hierarhične strukture.



Kadar želimo uporabiti lastnosti dveh ali več razredov, da lahko iz njih izpeljemo nov razred, govorimo o **večkratnem dedovanju**. Tako izpeljan razred ima več prednikov. Uporaba večkratnega dedovanja je sporna zaradi zmanjševanja preglednosti modela in možnosti pojavitve **ponavljajočega dedovanje**. Paziti moramo na situacije, ko se ista operacija pojavlja na dveh mestih.

3 POENOTEN JEZIK ZA MODELIRANJE – UML IN POENOTEN PROCES

Poenoten jezik za modeliranje UML je grafični jezik za vizualizacijo, specifikacijo, načrtovanje in dokumentacijo programske opreme. UML ponuja vzorce za načrtovanje sistema, vključujoč konceptualne stvari kot so poslovni procesi in funkcije sistema, kakor tudi bolj konkretne stvari, kot so vgrajeni stavki programskega jezika, sheme baze podatkov in ponovno uporabne programske komponente.

UML predstavlja zbir najboljših postopkov v praktičnem objektno orientiranem modeliranju. Je rezultat večletnega dela, katerega cilj je bil poenotenje metod, ki so najpogosteje v uporabi pri razvoju industrijskih rešitev. Največji trud pa je bil vložen v poenostavitev stvari in povečanju uporabnosti (UML 1.4, 2001, str. xix). UML ni metodologija za razvoj informacijskega sistema, ampak jezik, skupek tehnik, ki jih lahko uporabljamo skupaj s poenotenim procesom ali katero drugo metodologijo.

Začetek poglavja je namenjen opisu UML orodja, ki smo ga uporabljali pri razvoju programske rešitve. Potem bom predstavil konceptualni model UML jezika, ki ga sestavljajo trije osnovni gradniki: elementi, povezave in diagrami. Na koncu pa bom opisal poenoteni proces, katerega smernic sem se skušal držati pri razvoju svoje programske rešitve.

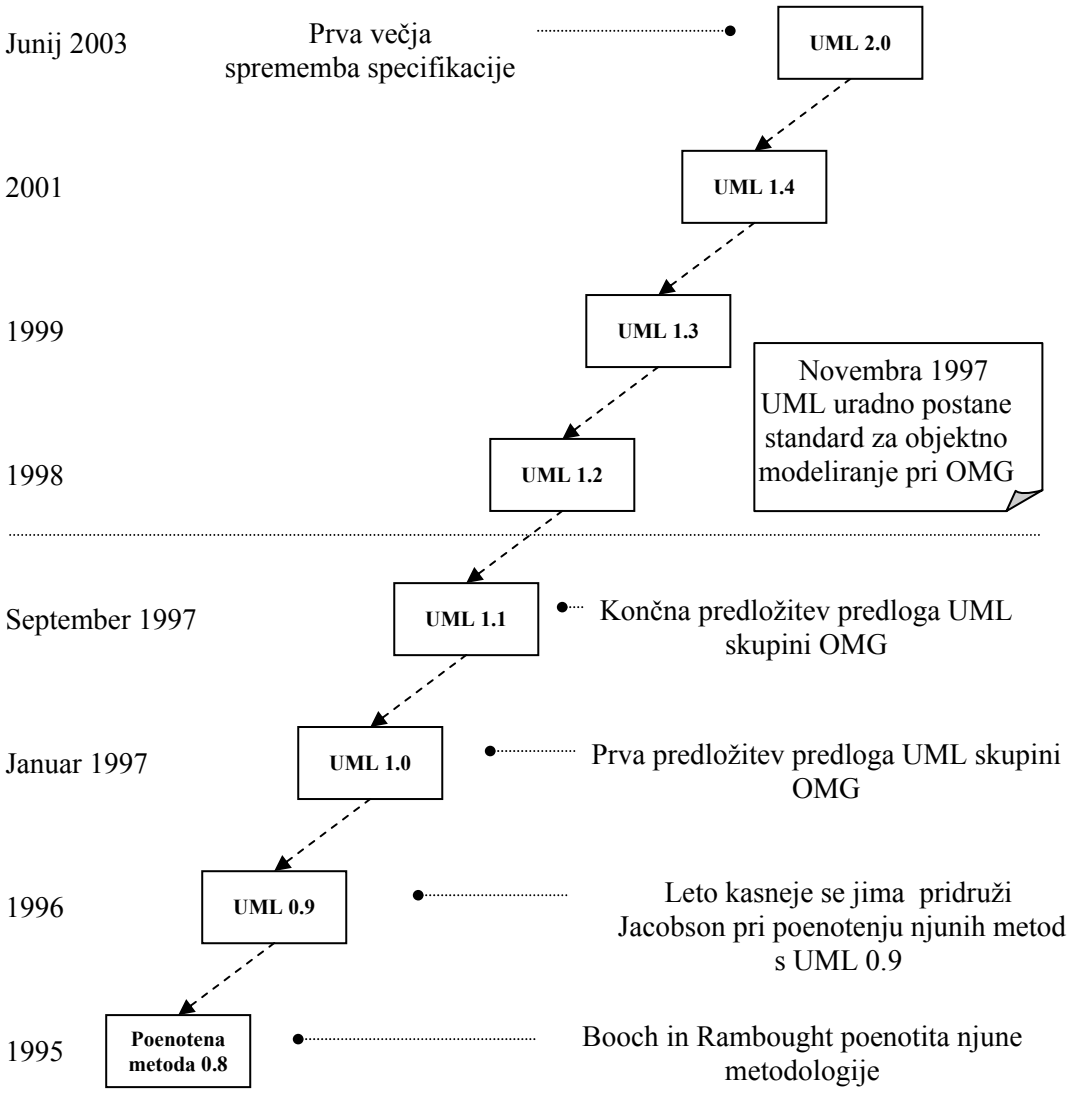
3.1 ZGODOVINA

V relativno kratkem časovnem obdobju je poenoten jezik za modeliranje postal glavni modelni jezik v računalniški industriji. Poenoten jezik za modeliranje je nastal z združitvijo več objektno orientiranih metodologij. V začetku 90-tih let število objektno orientiranih metod naraste do števila 50. Številne izmed teh metod težko najdejo ustrezen modelni jezik, ki bi ustrezal njihovim specifičnim zahtevam. Med temi novimi metodami so izstopale Boochova metoda, Jacobsonova metoda OOSE[10] in Rambaughtova metoda OMT[11]. Boochova metoda je bila učinkovita v fazi načrtovanja in konstrukcije, OOSE je omogočala odlično podporo za diagrame razredov, analizo in visokonivojsko načrtovanje, OMT – 2 pa je bila najuporabnejša pri analizi in

[10] ang. Object-Oriented Software Engineering
[11] ang. Object Modeling Technique

pri sistemih z veliko podatki (Booch et.al., 1999, xviii). V srednjih 90-tih letih se avtorji vseh treh metod Boochove, OOSE in OMT odločijo za snovanje novega modelnega jezika, ki bi bil podprt v vseh treh metodah – poenotenega jezika za modeliranje (Booch et. al., 1999, str. xviii).

Slika 5: Zgodovinski razvoj specifikacije poenotenega modelnega jezika UML



Vir: UML 2001 A Standardization Odyssey, 1999

3.2 NAMEN MODELIRANJA Z JEZIKOM UML

Modele gradimo zaradi boljšega razumevanja delovanja sistema, ki ga načrtujemo (Booch et. al.,1999, str. 6). Namen modeliranja je doseči:

1. Boljšo vizualizacijo delovanja delujočega sistema ali sistema v razvoju. Konceptualni model omogoča lažje razumevanje sistema tako za tistega, ki ta sistem razvija, kakor tudi za nekoga drugega, ki poizkuša delovanje le-tega razumeti. Model ne omogoča samo lažjega razumevanja sistema kot takega, ampak tudi razumevanje koncepta njegovega načrtovanja.
2. Prikaz strukture in obnašanja sistema. Zgrajen model mora biti natančna kopija modeliranega sistema in mora biti jasno definiran.
3. Izdelava šablone, ki nas vodi pri načrtovanju sistema. UML modelni jezik omogoča generacijo kode v številnih programskih jezikih. Mogoča je pretvorba modela v programske jezike kot so C, C++, Java, Visual Basic ali pa celo v relacijsko bazo podatkov.
4. Dokumentiranje procesa načrtovanja. UML omogoča dokumentacijo celotne arhitekture sistema, pisanje zahtev in testnih procedur. UML pa vsebuje tudi jezik za modeliranje aktivnosti pri načrtovanju projekta in upravljanju z izdanimi verzijami sistema.

3.3 KONCEPTUALNI MODEL

Za razumevanje UML-ja je potrebno poznati pomen treh osnovnih gradnikov:

1. elementov,
2. povezav,
3. diagramov.

3.3.1 ELEMENTI

V UML se pojavljajo štiri vrste elementov:

1. strukturni elementi,
2. elementi obnašanja,
3. elementi združevanja,
4. elementi opomb.

3.3.1.1 STRUKTURNI ELEMENTI

Največkrat so to statični deli modela, ki predstavljajo element shematično ali fizično:

1. razredi,
2. vmesniki,
3. sodelovanja,
4. primeri uporabe,
5. komponente,
6. vozlišča.

Razred je opis nabora objektov, ki delijo enake atribute, operacije, povezave in semantiko.

Vmesnik definira povezavo med specifikacijo, ki nam pove, kaj abstrakcija počne in med implementacijo, ki pove, kako to počne. Vmesnik je zbirka operacij, ki opisujejo, kako razred ali

komponenta deluje. Lahko prikazuje celotno ali pa le delno obnašanje razreda ali komponente. Največkrat so vmesniki povezani z razredom ali komponento, ki realizira vmesnik. Dobro strukturiran vmesnik omogoča jasno delitev med pogledom od zunaj in notranjim delovanjem abstrakcije, s tem omogoča razumevanje delovanja brez detajlnega poznavanja delovanja abstrakcije.

Sodelovanje je skupina razredov, vmesnikov in drugih elementov, ki delujejo skupaj v kooperativnem obnašanju, katerega rezultat je izboljšano delovanje sistema. Zapis v UML prikazuje sodelovanje navzven kot en sam kos, vendar pa sodelovanje v sebi skriva različne diagrame, največkrat so to razredni diagrami (strukturni del sodelovanja) in diagram sodelovanja (vedenjski del sodelovanja).

Primer uporabe je opis niza zaporednih akcij, ki jih izvaja sistem z namenom, da določenemu akterju prinese rezultat, ki ga lahko opazujemo.
Aktivni razred je razred, katerega objektom pripadajo eden ali več programskih procesov,[12] ali programskih niti[13] in lahko sam nadzoruje svojo aktivnost. Aktivni razred je razred, katerega instance so aktivni objekti.

Komponenta je fizični in zamenljivi del sistema, ki omogoča izvedbo niza vmesnikov. Komponente uporabljamo za modeliranje fizičnih stvari, ki jih vključimo s pomočjo izvršljivih programov, knjižnic, tabel, datotek in dokumentov. Ponavadi komponenta vsebuje fizični paket logičnih elementov, kot so razredi, vmesniki in sodelovanja.

Vozlišče[14] je fizični element, ki obstaja v času izvajanja programa in predstavlja programski vir, z lastno rezervacijo pomnilnika in sposobnostjo procesorskega dostopa. Vozlišča se uporabljajo za modeliranje topologije strojne opreme, na kateri teče nas sistem. Izraz vozlišče predstavlja procesor ali napravo, na kateri se komponente uporabljajo.

3.3.1.2 ELEMENTI OBNAŠANJA

Elementi obnašanja so dinamičen del UML modela in predstavljajo časovno in krajevno obnašanje modela.

Ločimo dva tipa elementov obnašanja:

- interakcije,
- diagram stanj.

[12] Programski proces je potek programa, ki se lahko izvaja sočasno z drugimi procesi.
ang. process

[13] Programska nit pa je potek programa, ki se lahko izvaja sočasno z drugimi programskimi nitmi znotraj enega programskega procesa.
ang. thread

[14] ang. node

Interakcija je obnašanje, ki vključuje nabor sporočil, ki se izmenjujejo med objekti v okviru določenega konteksta za doseganje določenega cilja. Z uporabo interakcije modeliramo dinamični aspekt sodelovanja, predstavimo funkcijo delovanja skupine objektov, ki je z vzajemnim delovanjem boljše kot delovanje vsakega objekta posebej. Funkcije delovanja pa so predstavitev prototipov instanc razredov, vmesnikov, komponent, vozlišč in primerov uporabe. Vsako interakcijo lahko modeliramo s poudarkom na časovnem poteku sporočil ali pa s poudarkom na zaporedju prehajanja sporočil.

Diagram stanj je obnašanje, ki opisuje zaporedje stanj objekta ali interakcij med objekti, kot odgovor na dogodke v sistemu. Z njim lahko zelo dobro modeliramo življenjsko dobo objekta, s čimer zelo dobro vidimo, kaj se v sistemu dogaja. Diagram stanj vključuje številne elemente stanja, prehode med posameznimi objekti, dogodki (z njimi lahko sprožimo prehode med objekti) in aktivnosti (odziv na prehod).

3.3.1.3 ELEMENTI ZDRUŽEVANJA

Element združevanja je organizacijski del UML, glavni mehanizem tega procesa pa imenujemo paket, ki omogoča združevanje elementov. Za razliko od komponent, ki obstajajo v času izvajanja, je paket konceptualnega značaja. Paket nam služi za združevanje modelnih elementov na nivoju arhitekturnega načrtovanja v večje skupine, s katerimi je lažje upravljati.

3.3.1.4 ELEMENTI OPOMB

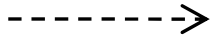
Elemente opomb uporabljamo za opisovanje in pojasnjevanje delovanja posameznih elementov v modelu. Ta element imenujemo zapisek, ki ga lahko pripnemo na element ali na skupino elementov.

3.3.1.5 POVEZAVE

Povezava je spoj dveh elementov. V UML se pojavljajo štiri vrste povezav:
1. odvisnosti,
2. asociacije,
3. generalizacije,
4. realizacije.

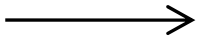
Odvisnost je semantična povezava dveh elementov, pri kateri sprememba neodvisnega elementa lahko povzroči spremembo odvisnega elementa. Odvisnost se uporablja takrat, ko poizkušamo pokazati, da ena stvar uporablja drugo.

Slika 6: Odvisnost

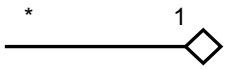


Asociacija je strukturna povezava, ki pojasnjuje nabor vezi, pri čemer so vezi povezave med objekti. Poseben primer asociacije je agregacija ali združevanje, ki predstavlja povezavo med celoto in deli te celote.

Slika 7: Asociacija

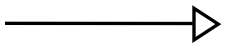


Slika 8: Agregacija



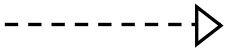
Generalizacija predstavlja povezavo specializacije in generalizacije, v kateri so objekti specializiranega elementa nadomestljivi z elementi generaliziranega elementa. Kot sem že omenil, specializiran element ima isto strukturo in obnašanje kot generaliziran element.

Slika 9: Generalizacija



Realizacija je semantična povezava med elementoma, pri kateri eden element določi dogovor, ki ga mora drugi izvršiti. Realizacija se pojavlja med vmesniki v razredih ali med komponentami, ki jih realizirajo.

Slika 10: Realizacija



3.3.2 DIAGRAMI

Diagram je grafična predstavitev skupine elementov, njegov smisel pa je predstavitev sistema iz različnih vidikov. Z diagramom dosežemo ločeno obravnavo posameznih problemov v sistemu in poenostavimo modeliranje sistema. V UML ločimo strukturalne diagrame, ki predstavljajo statično strukturo sistema in diagrame, ki predstavljajo dinamično strukturo sistema.

Diagrami v UML:

1. Razredni diagram (*ang. class diagram*).
2. Objektni diagram (*ang. object diagram*).
3. Diagram komponent (*ang. component diagram*).
4. Diagram porazdelitve (*ang. deployment diagram*).
5. Diagram primerov uporabe (*ang. use case diagram*).
6. Diagram stanj (*ang. statechart diagram*).
7. Diagram zaporedja (*ang. sequence diagram*).
8. Diagram sodelovanja (*ang. collaboration diagram*).
9. Diagram aktivnosti (*ang. activity diagram*).

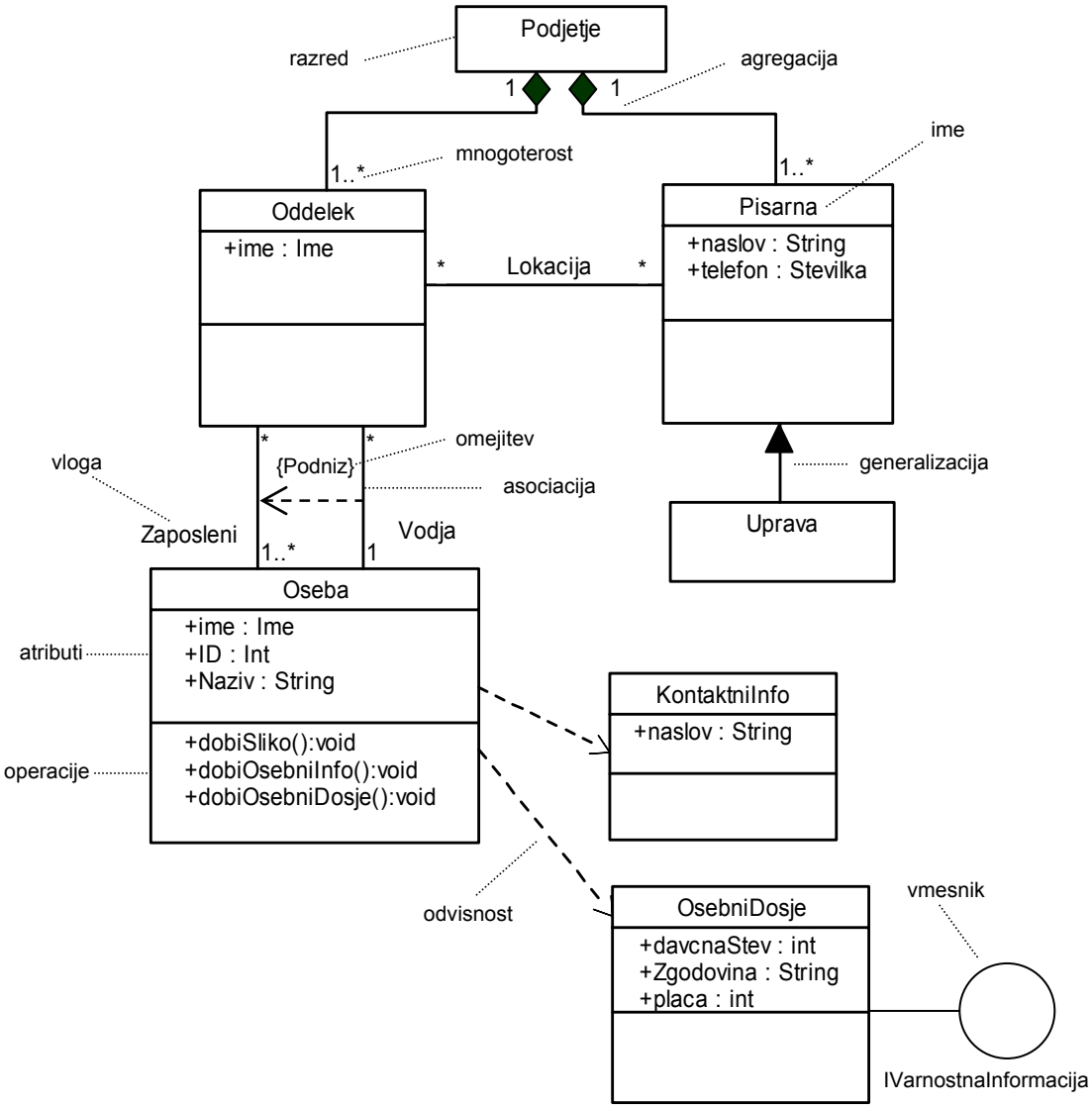
Statični pogled

Dinamični pogled

Strukturni diagrami so diagrami za vizualizacijo, opisovanje, načrtovanje in dokumentiranje statičnega pogleda na sistem. Statični pogled na sistem predstavlja njegovo relativno stabilno ogrodje in zgradbo. Statični pogled opisuje funkcijo in postavitev razredov, vmesnikov, sodelovanj, komponent in vozlišč v modelu sistema.

Diagram obnašanja uporabljamo za vizualizacijo, opisovanje, načrtovanje in dokumentiranje dinamičnega pogleda na sistem. Dinamičen pogled predstavlja dele sistema, ki se spreminjajo. V sistemu dinamični pogled predstavlja tok sporočil v nekem časovnem obdobju in fizično premikanje komponent.

Slika 11: Razredni diagram



Vir: Booch et. al.,1999, str.106

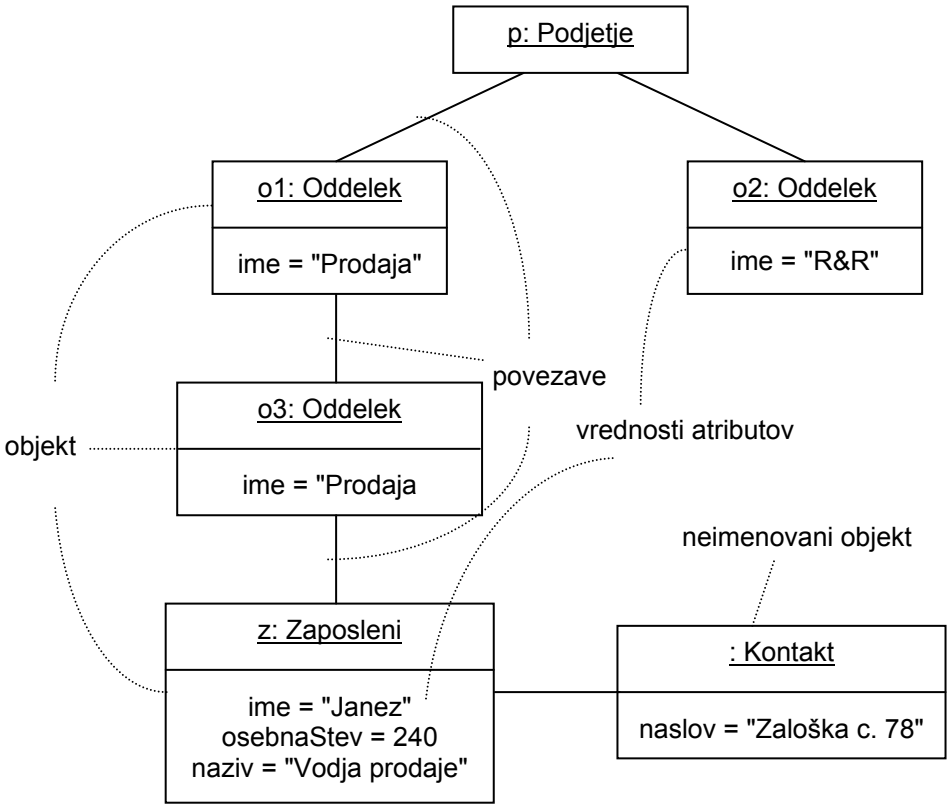
3.3.2.1 RAZREDNI DIAGRAM

Razredni diagram, na sliki 11, prikazuje razrede, vmesnike, sodelovanja in njihove povezave. Povezave med njimi so lahko odvisnosti, generalizacije, asociacije in agregacije. Diagram lahko vsebuje tudi pakete in podsisteme, ki so namenjeni razvrstitvi elementov sistema v večje skupine. Razredni diagram je najpogosteje uporabljen diagram pri modeliranju objektno orientiranega sistema. Razredni diagrami opisujejo statični vidik načrtovanja sistema in so osnova za nekatere druge diagrame: diagram komponent in diagram porazdelitve. Niso pomembni samo za vizualizacijo, opisovanje in dokumentiranje, temveč tudi za načrtovanje izvršljivih sistemov prek vnaprejšnjega načrtovanja, generacija programske kode neposredno iz UML modela , in vzvratnega načrtovanja, prenos informacij iz programske kode v UML model.

3.3.2.2 OBJEKTNI DIAGRAM

Objektni diagram prikazuje nabor objektov in njihovih povezav. Uporabljamo jih za prikazovanje struktur podatkov in trenutnih posnetkov instanc elementov razrednega diagrama. Objektni diagrami so uporabljeni za prikaz statičnega vidika načrtovanja sistema, podobno kot razredni diagrami, z vidika realnih ali prototipnih instanc.

Slika 12: Objektni diagram

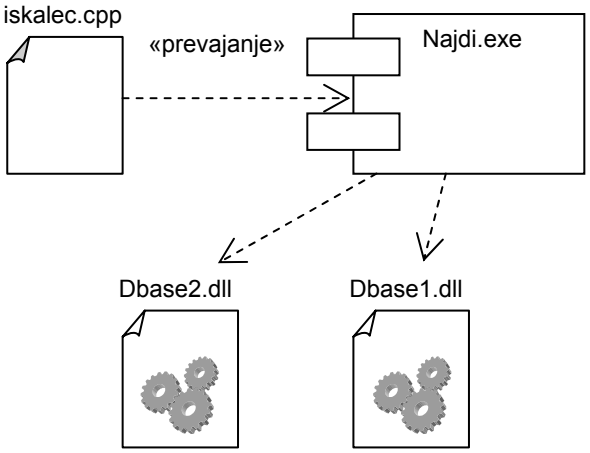


Vir: Booch et. al.,1999, str.196

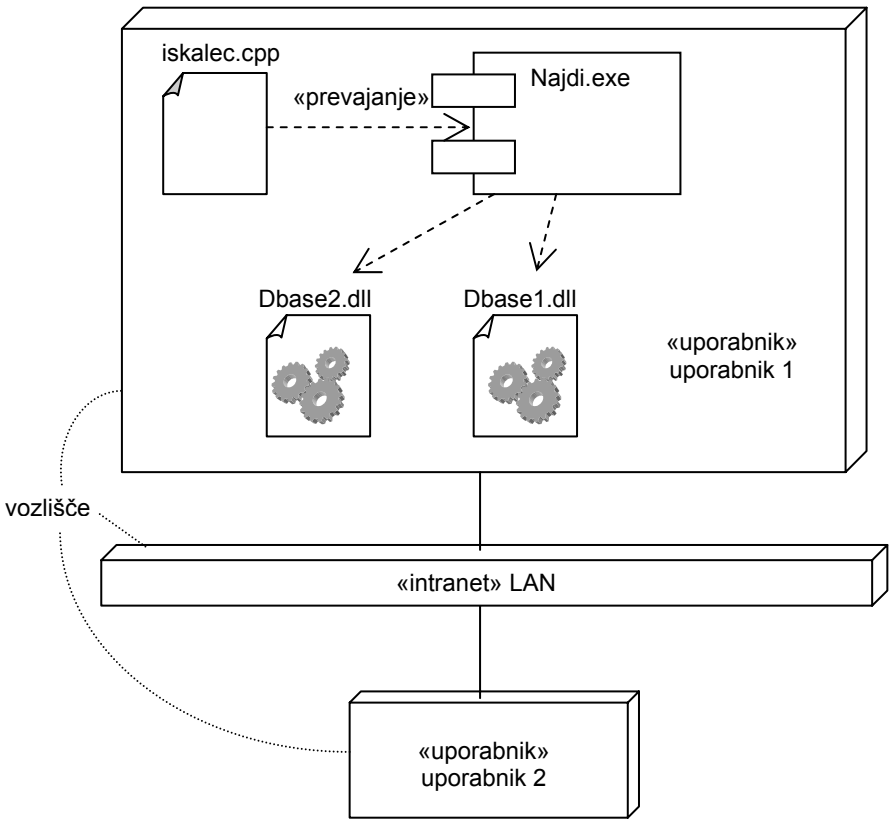
3.3.2.3 DIAGRAM KOMPONENT

Diagram komponent na sliki 13 prikazuje organizacijo in odvisnosti med naborom komponent. Z njim je prikazan statični izvedbeni vidik sistema. Diagram komponent je tesno povezan z razrednim diagramom, saj komponenta predstavlja enega ali več razredov, vmesnikov ali sodelovanj. Uporabljamo ga za modeliranje in upravljanje z izvršljivimi elementi, knjižnicami, tabelami, datotekami in dokumenti.

Slika 13: Iskanje v bazi podatkov, prikazano z diagramom komponent



Slika 14: Iskanje v bazi podatkov preko intraneta, prikazano z diagramom porazdelitve



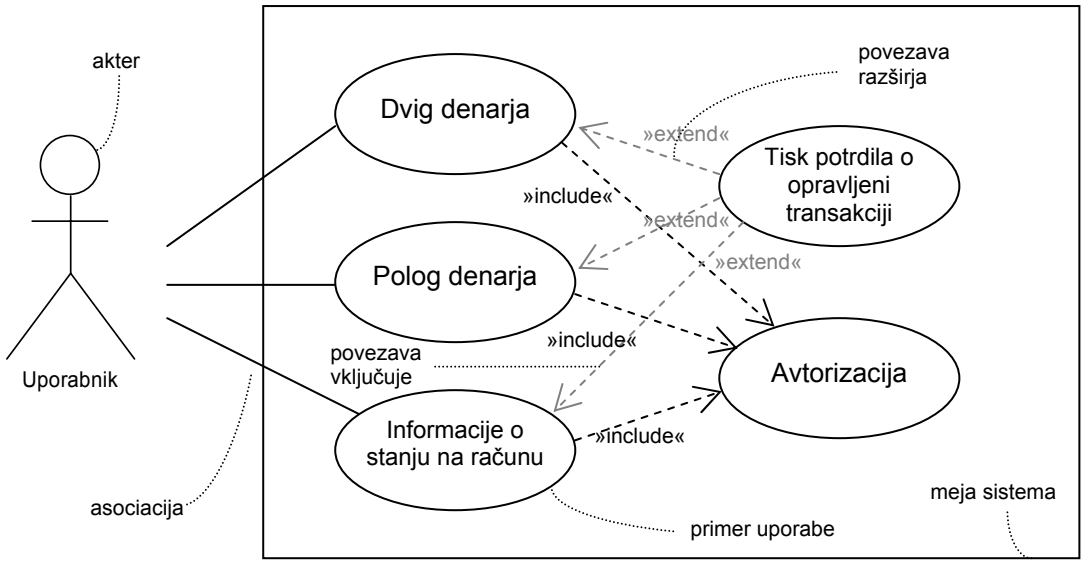
3.3.2.4 DIAGRAM PRIMEROV UPORABE

Diagram primerov uporabe prikazuje obnašanje sistema. Diagram primerov uporabe sestavljajo primeri uporabe, akterji in njihove povezave. Njihov osnovni namen je lažje in hitrejše zajemanje, ter predstavitev uporabnikovih zahtev. Diagrami primerov uporabe povežejo akterje – uporabnike sistema in procese, ki v sistemu tečejo. S tem omogočajo modeliranje z uporabnikovim sodelovanjem skozi celoten razvojni cikel. Služijo kot sredstvo za zajemanje in sledenje uporabnikovim zahtevam o izgledu in uporabnosti sistema (Vpeljava poslovnega modeliranja, 2002).

Akterji ponazarjajo okolje. Okolje prestavlja posameznike, naprave ali druge sisteme zunaj modeliranega sistema, ki z njim komunicirajo in s tem vplivajo na njegovo delovanje. Akter tako predstavlja abstrakcijo vlog uporabnika sistema. *Primer uporabe* predstavlja zaporedje transakcij v sistemu in praviloma komunicira z akterjem. *Povezave* v diagramu primerov uporabe:

- Asociacija sproži ali zgolj sodeluje s primerom uporabe.
- Razširja: Pomeni odvisnost med primeri uporabe. Neki primer uporabe razširja drugega, kadar se le ta izvede le v določenih primerih.
- Vključuje: Kadar je obnašanje primera uporabe skupno več kot enemu primeru uporabe, tedaj rečemo, da le ta vključuje sledeče primere uporabe.
- Generalizacija: Povezava med dvema akterjema ali primeroma uporabe. Če je primerek A generaliziran iz primerka B, to pomeni, da je primer B poseben primer primerka A.

Slika 15: Primer realizacije primera uporabe za bančni sistem



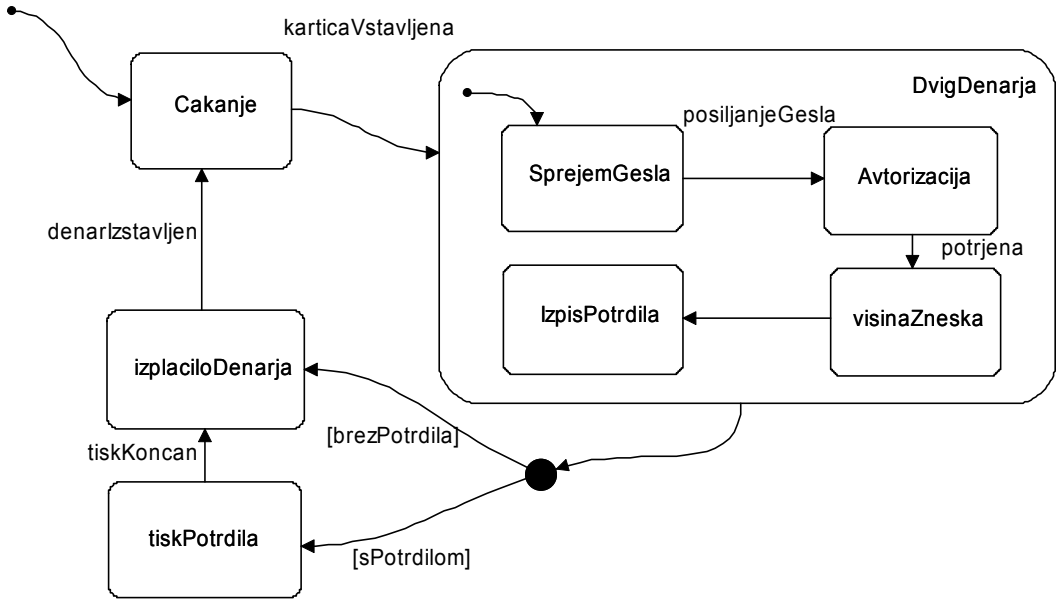
Vir: Vpeljava poslovnega modeliranja, 2002

3.3.2.5 DIAGRAM STANJ

Diagrami stanj omogočajo opisovanje obnašanja sistema ali posameznega razreda. Največkrat z njim modeliramo obnašanje reaktivnih objektov. *Reaktivni objekti*[15] so objekti, katerih obnašanje je odvisno od dogodkov zunaj njihovega konteksta.

Diagrami stanj prikazujejo zaporedje stanj in dogodke, ki sprožijo prehode med stanji. *Stanje* je okoliščina v življenjski dobi nekega objekta, v katerem le-ta izpolnjuje nek pogoj. Dogodek pa je pripetljaj nekega dražljaja, ki lahko sproži prehod med stanji. *Prehod* je povezava med dvema stanjema.

Slika 16: Procedura za dvig denarja preko bankomata realiziran s pomočjo diagrama stanj

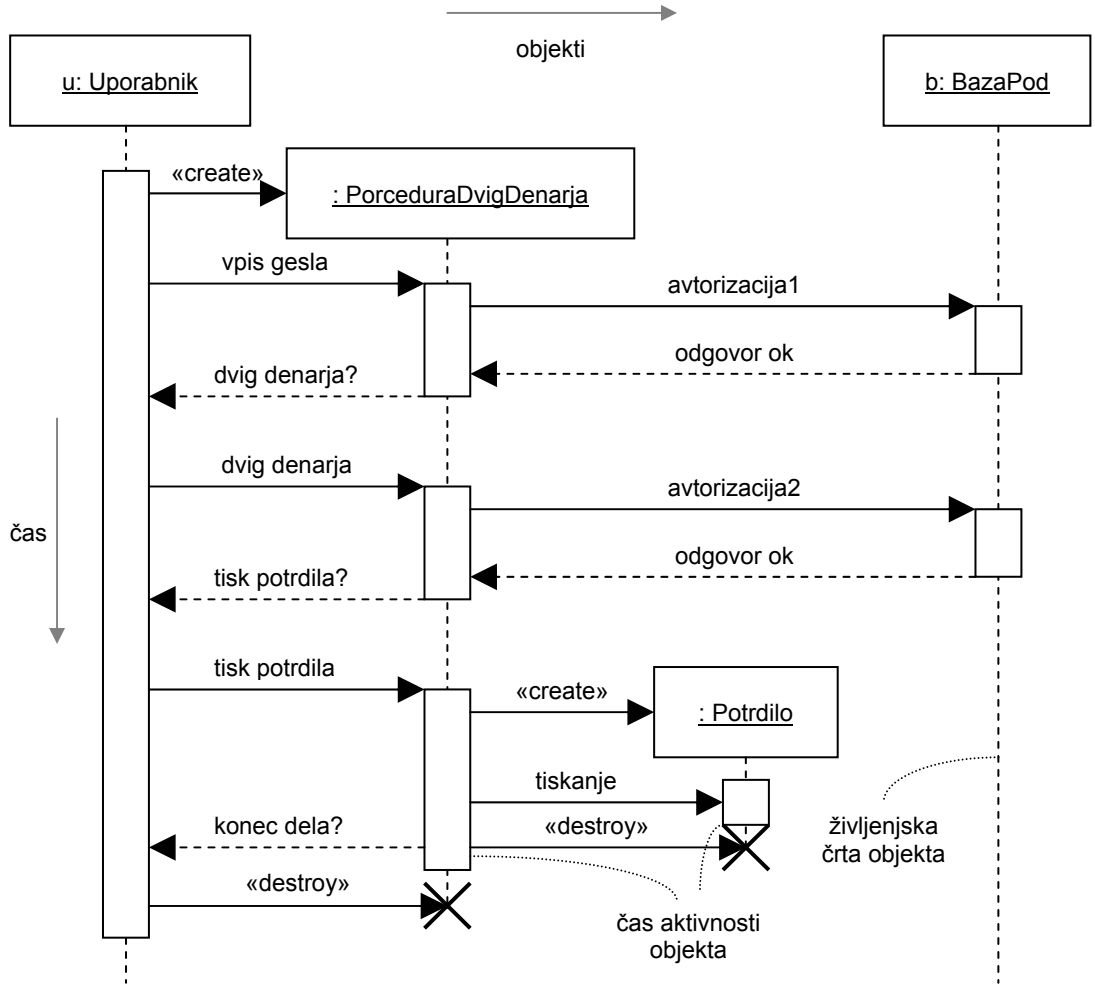


3.3.2.6 DIAGRAM ZAPOREDJA

Diagram zaporedja prikazuje časovni potek izmenjave sporočil med objekti v sistemu. Elementi diagrama zaporedja so objekti, njihove povezave in sporočila med njimi. Uporabljamo jih za ustvarjanje scenarijev v diagramu primerov uporabe. Diagram zaporedja nam pomaga razumeti povezave in medsebojne vplive med objekti s prikazom sporočil izmenjanih v nekem časovnem intervalu.

[15] Npr.: Stanja v diagramu stanj ne morejo vplivati na to, kdaj bodo dosežena, zato so stanja reaktivni objekti.

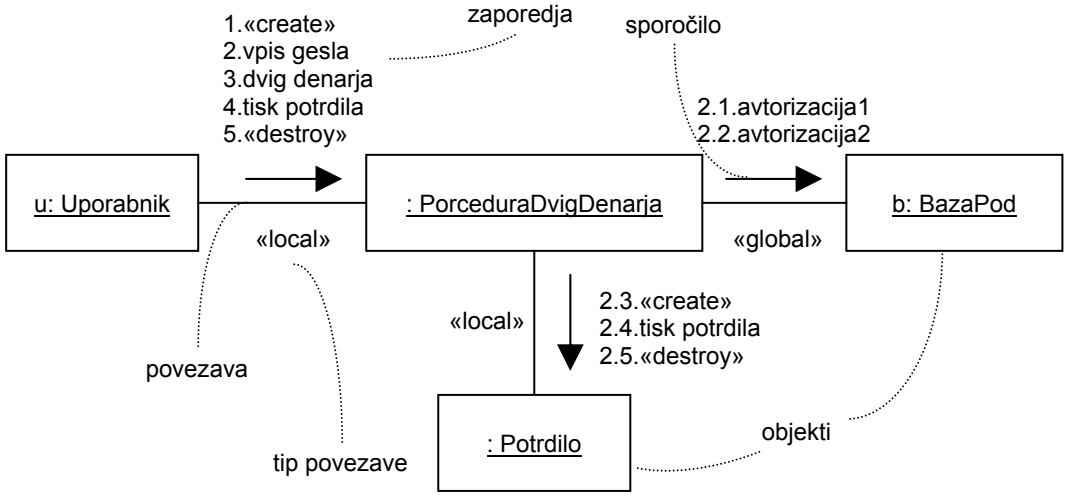
Slika 17: Procedura za dvig denarja preko bankomata z diagramom zaporedja



3.3.2.7 DIAGRAM SODELOVANJA

Diagram sodelovanja je zelo podoben diagramu zaporedja z bolj poudarjeno strukturno organizacijo objektov, ki pošiljajo in sprejemajo sporočila. Diagram zaporedja bolj poudari zaporedje poslanih sporočil med objekti, diagram sodelovanja pa povezave med objekti. Načrtovanje diagrama sodelovanja začnemo s postavitvijo objektov, ki sodelujejo v interakciji. Objekte nato povežemo s povezavami, katerim potem dodamo še sporočila, ki jih objekt sprejema in pošilja. Takšen diagram omogoči preglednost vseh poslanih sporočil med povezanimi objekti.

Slika 18: Procedura za dvig denarja preko bankomata z diagramom sodelovanja



3.3.2.8 DIAGRAM AKTIVNOSTI

Če diagram sodelovanja prikazuje prehode iz objekta na objekt, nam diagram aktivnosti prikazuje prehode iz aktivnosti na aktivnost.

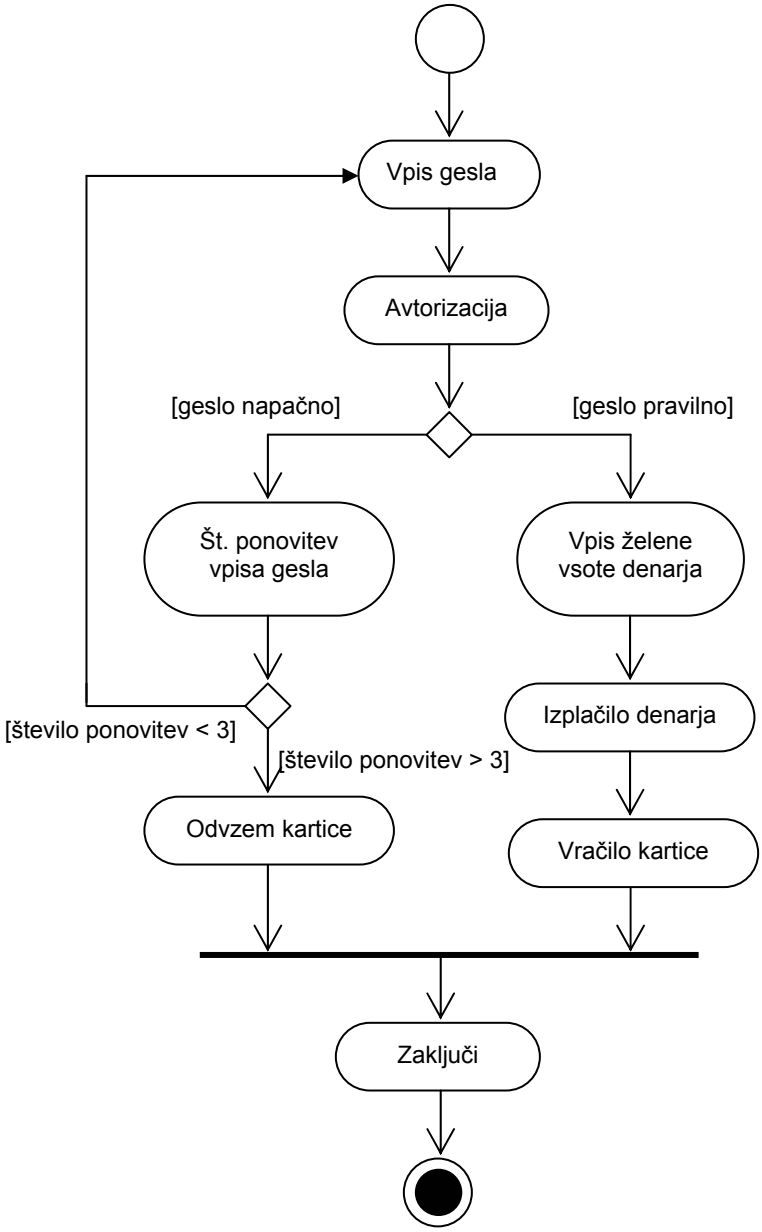
Elementi diagrama aktivnosti:

- aktivnosti,
- prehodi med aktivnostmi,
- razvejitve, sočasne razvejitve[16],
- sočasno stičišče[17],
- linija toka dogodkov[18],
- objektni tok.

Rezultat aktivnost je akcija, katere posledica je prehod v drugo aktivnost, ali pa le vračanje vrednosti. Diagram omogoča navadne razvejitve ali pa sočasne razvejitve. Le-te omogočajo sočasno izvajanje dveh različnih procesov, katerih sočasno izvajanje se konča v sočasnem stičišču. Linija toka dogodkov v praksi omogoča razdeljevanje toka delovanja poslovnih procesov po posameznih poslovnih enotah ali skupinah; iz diagrama je jasno razvidna odgovornost za posamezne aktivnosti. Če so v toku dogodkov vpleteni tudi pomembni objekti, jih lahko s pomočjo objektnega toka dodamo v diagram aktivnosti.

[16] ang. Concurrent fork
[17] ang. Concurrent join
[18] ang. Swimlane

Slika 19: Procedura za dvig denarja preko bankomata z diagramom aktivnosti



3.3.2.9 MODELIRANJE SISTEMA Z RAZLIČNIH VIDIKOV

Modeliranje sistema iz različnih vidikov prinese sočasno načrtovanje z večimi dimenzijami hkrati. Vendar pa je potrebno pri takšnem vidiku paziti, kajti v mislih moramo imeti celo sliko načrtovanega sistema in se ne smemo osredotočiti le na en vidik načrtovanja sistema. Potrebno je raziskati, kateri od petih vidikov najbolje prikazuje arhitekturne lastnosti modeliranega sistema.

Za modeliranje enostavnega sistema ne potrebujemo celotnega nabora diagramov, ampak je za potrebe modeliranja dovolj uporabiti le diagram primerov uporabe, razredni diagram in interakcijski diagram.

Pri kompleksnejših sistemih pa smo prisiljeni poseči po celotnem naboru diagramov. Nekateri diagrami nam postrežejo z boljšo predstavitvijo enega vidika, drugi pa so boljši za predstavitev drugega vidika.(Booch et. al.,1999, str.99):

- Vidik primerov uporabe: → Diagrami primerov uporabe
  → Diagrami aktivnosti (dinamični pogled)
- Načrtovalni vidik: → Razredni diagrami (statični pogled)
  → Interakcijski diagrami (dinamični pogled)
  → Diagram stanj (dinamični pogled)
- Procesni vidik: → Razredni diagrami (statični pogled)
  → Interakcijski diagrami (dinamični pogled)
- Izvedbeni vidik: → Diagram komponent
- Porazdelitveni vidik: → Diagram porazdelitve

Sistem lahko modeliramo z različnih vidikov, poleg tega pa lahko sistem modeliramo na različnih nivojih razumljivosti. Potreba po tem nastane, če želita isti model sistema uporabljati dva različna uporabnika npr. analitik in programer. Analitik uporablja scenarije primerov uporabe, pri katerih je podrobno delovanje samega sistema zanj nepomembno in samo zamegli njegov pogled. Povsem drugače pa je s programerjem, ko potrebuje dostop do operacij in atributov. Zato lahko rečem, da uporabnika potrebujeta dva različna nivoja razumljivosti modela sistema.

3.4 METODOLOGIJA POENOTENEGA PROCESA RAZVOJA PROGRAMSKE OPREME

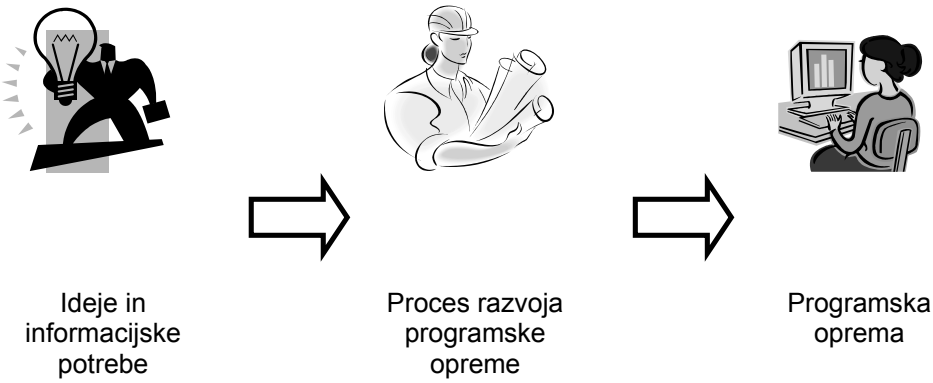
Proces razvoja programske opreme[19], ki ga imenujemo tudi proces inženiringa programske opreme[20], nam pove kdo, kaj, kdaj in kako razvija programsko opremo (Arlow, Neustadt, 2002, str.22).

Poenoten proces razvoja programske opreme[21] leta 1999 postane industrijski standard za proces razvoja programske opreme. Temu procesu rečemo tudi poenoten proces. UML je postal jezik za vizualizacijo razvoja programske opreme, poenoten proces pa predstavlja proces razvoja

[19] ang. Software development process – SDP
[20] ang. Software engineering process - SEP
[21] ang. Unified Software Development Process – USDP

programske opreme. Poenoten procesa razvoja programske opreme izvira iz podjetja Ericsson leta 1967 in iz podjetja Rational od leta 1996 do 1997. Na trgu je dostopnih kar nekaj komercialnih različic poenotenega procesa. Pomembno pa je, da vse te variante vsebujejo vse lastnosti poenotenega procesa, ki ga po funkcionalnosti celo presegajo. Najbolj pogosto uporabljena različica poenotenega procesa je Rational poenoten proces. Proizvod podpira tudi standarde in orodja, ki sicer niso vključeni v poenoten proces, vendar so pomembni pri razvoju programske opreme. Omogoča tudi generacijo dokumentacije in uporabniške vodnike po vsebovanih orodjih.

Slika 20: Proces razvoja programske opreme



Vir: Arlow, Neustadt, 2002, str.24

3.4.1 POENOTEN PROCES IN PROJEKT

Poenoten proces je generičen proces razvoja programske opreme in mora biti prilagojen na edinstvene pogoje dela v vsakem podjetju posebej (Arlow, Neustadt, 2002, str. 28). Proces prilagajanja poenotenega procesa vključuje definiranje in vključevanje:

- Internih standardov podjetja.
- Šablone dokumentov.
- Orodja – prevajalniki, orodja za konfiguracijski management, itd.
- Baze podatkov – iskanje napak, sledenje dogajanju v projektu, itd.
- Modifikacije v življenjskem ciklu – nadzor kvalitete programske opreme.

3.4.1.1 OSNOVNA NAČELA

1. *Primeri uporabe* zagotavljajo sledenje zahtevam za razvoj programske rešitve, lahko tudi rečemo, da prav zahteve upravljajo s poenotenim procesom. Pred vsakim načrtovanjem programskih rešitev je potrebno analizirati, kakšne so *potencialne nevarnosti*, ki lahko ogrozijo našo aplikacijo.

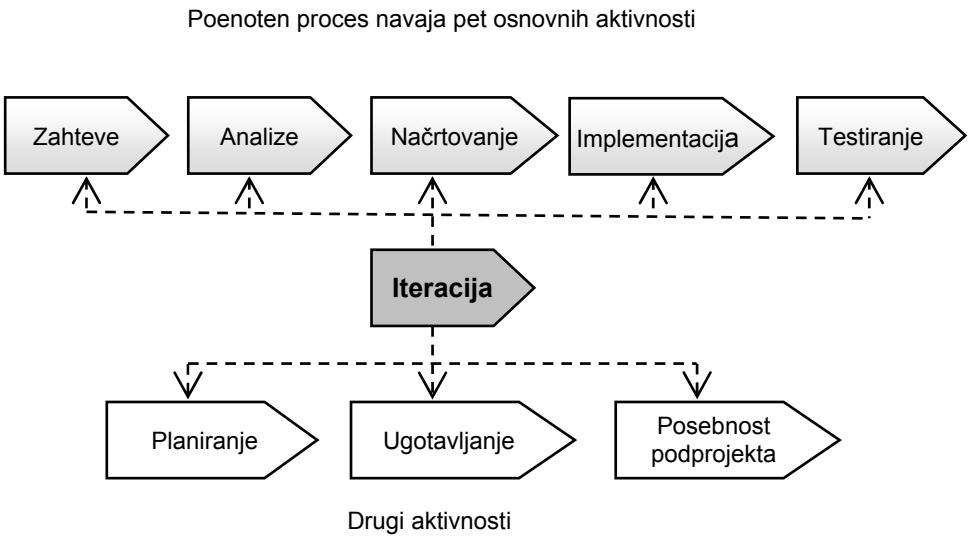
2. Namen poenotenega procesa je izdelati programsko rešitev, ki bo imela zadosti *robustno arhitekturo*. Arhitektura programske rešitve pove, kako je sistem razdrobljen na posamezne komponente, kako se komponente povezujejo in kako so te porazdeljene v ciljnem sistemu.
3. Poenoten proces je *iterativen* in *inkrementalen*. Iterativnost pomeni, da naš projekt razbijemo na več manjših podprojektov, od katerih vsak predstavlja določeno funkcionalnost. Te funkcionalnosti počasi inkrementiramo ali dopolnjujemo, tako da naš sistem dobi popolno funkcionalnost.

3.4.1.2 ITERATIVNOST IN INKREMENTALNOST POENOTENEGA PROCESA

Ideja iterativnosti je v tem, da veliko lažje rešujemo manjše probleme kot večje. Velike projekte razvoja programske opreme zato razbijemo na manjše podprojekte, s katerimi lažje upravljamo in jih nadgrajujemo. Vsak od teh podprojektov ali iteracij razvojnega procesa potrebuje opredelitev zahtev, analizo, načrtovanje, implementacijo in testiranje rezultatov.

Vsaka od iteracij je osnova za naslednjo delno verzijo sistema, zadnja iteracija preko vseh prejšnjih iteracij predstavlja končno verzijo sistema. Zadnja iteracija predstavlja prečiščeno verzijo vseh prejšnjih verzij in je tista, ki zares izpolnjuje uporabnikove zahteve, ter je hkrati enostavna in zanesljiva (Booch, 1994, str. 232).

Slika 21: Aktivnosti vsake iteracije



Vir: Arlow, Neustadt, 2002, str.31

Vsaka iteracija poenotenega procesa ima za rezultat neko osnovno verzijo, ki predstavlja interno ali eksterno izdajo tistega, kar smo uspeli izboljšati ali narediti do te iteracije. Takšen proces omogoča:

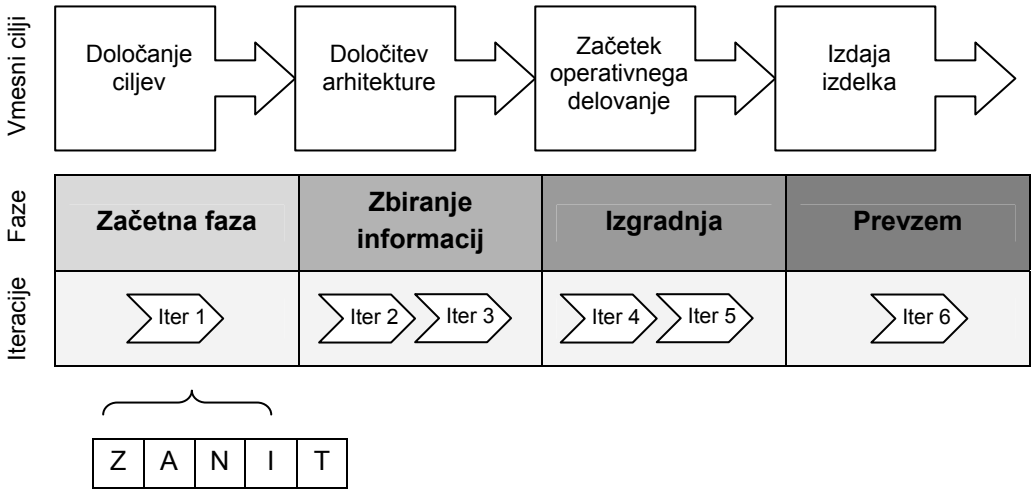
- Gradnjo novih programskih rešitev na podlagi že preverjenih programskih rešitev.
- Takšna osnova verzija, je spremenljiva samo s pomočjo posebnih procedur in omogoča nadzor sprememb.

Razlika, ki jo prineseta dve zaporedni iteraciji, predstavlja inkrement ali korak procesa.

3.5 STRUKTURA POENOTENEGA PROCESA

Življenjski cikel izdelka je razdeljen na štiri faze: začetek[22], obdelava[23], gradnja[24] in prehod[25]. Vsaka od teh faz se konča z mejnikom. V vsaki fazi naredimo eno ali več iteracij, vsako iteracijo izvedemo s pomočjo osnovnih ali sekundarnih delovnih procesov. Število iteracij na fazo je odvisno od velikosti projekta, vendar pa nobena iteracija ne bi smela trajati več kakor tri mesece.

Slika 22: Prikaz časovnega poteka poenotenega procesa



Vir: Arlow, Neustadt, 2002, str.32

Poenoten proces je sestavljen iz zaporedja štirih faz, od katerih se vsaka konča z vmesnim ciljem:

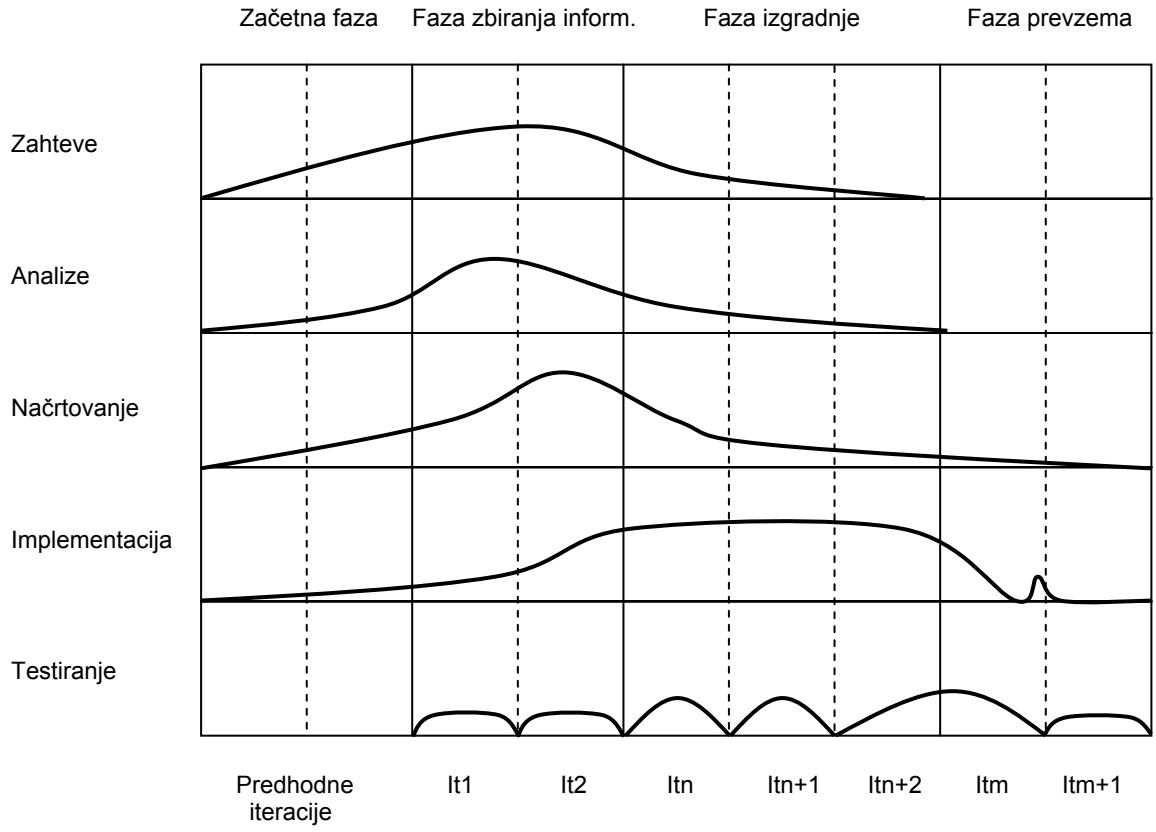
1. Začetna faza – iskanje poslovnih možnosti.
2. Faza zbiranja informacij – izdelava načrta projekta in njegove zgradbe.
3. Faza izgradnje – gradnja sistema.
4. Faza prevzema – dostava sistema končnim uporabnikom.

Ko projekt izdelave izdelka prehaja iz faze v fazo poenotenega procesa, se tudi količina dela vloženega v posamezen delovni proces iteracije med seboj razlikuje. Začetna faza z največjo količino dela se posveča zahtevam in analizam. Faza obdelave poudarja zahteve, analize, dotika

[22] ang. Inception
[23] ang. Elaboration
[24] ang. Construction
[25] ang. Transition

pa se že načrtovanja sistema. V fazi gradnje je poudarek izključno na načrtovanju in implementaciji sistema. Faza prehoda pa se ukvarja z implementacijo in testiranjem.

Slika 23: Cikel razvoja programske opreme



Vir: Booch et. al.,1999, str.196

3.5.1 GLAVNE FAZE PROCESA

Naloga vsake izmed faz je usmerjanje delovnih procesov k uresničitvi vmesnega cilja faze.

3.5.1.1 INFORAMACIJSKE POTREBE ALI ZAHTEVE

Z zahtevami se ukvarjamo v začetni fazi in v fazi zbiranja zahtev razvojnega cikla poenotenega procesa (Arlow, Neustadt, 2002, str. 35). Iskanju zahtev pravimo tudi konceptualizacija, katere naloga je najti glavne zahteve za programsko rešitev. Vendar pa se je potrebno zavedati, da konceptualizacija ne definira celotnih zahtev programske rešitve, temveč predpostavlja, kako mora ta delovati.

Zamišljanje delovanja nekega sistema je sama po sebi zelo kreativna aktivnost, zato ne sme biti omejena s strogimi pravili razvojnega procesa. V podjetju moramo omogočiti nek aparat, ki

dostavlja organizaciji zadostno število novih idej, ki jih vključimo v zahteve za nov sistem ali izdelek. Nove ideje lahko izvirajo iz večih virov: končni uporabniki, razvijalci, analitiki, marketing itd. Te ideje mora management beležiti, tako da lahko najpomembnejše ideje kasneje izloči in omogoči njihovo implementacijo. Ko se podjetje odloči za določene ideje, sledijo naslednji koraki:

- Določitev nabora ciljev, s katerimi bomo lahko ocenili koncept.
- Zbrati je potrebno primerno skupino za razvoj prototipa, pogosto pa gre samo za enega človeka, ki je tudi vizionar. Največ kar lahko razvojna organizacija naredi za pospešitev razvoja koncepta je, da se ne meša v njeno delo.
- Ocena razvitega prototipa in sprejetje odločitev za razvoj produkta ali za nadaljnje raziskave. Odločitev za razvoj produkta mora biti sprejeta z upoštevanjem potencialnih rizikov posameznih konceptualnih poizkusov.

Potrebno je identificirati akterje in se vprašati, kakšni uporabniki bodo uporabljali sistem, ter kako bodo, gledano iz sistemskega analitika, nanj vplivali. Sledi opredelitev primerov uporabe sistema in funkcionalnosti, ki jih sistem nudi enemu ali večim akterjem. Primere uporabe je potrebno podrobno opredeliti. Rezultat teh aktivnosti je diagram primerov uporabe, ki opisuje funkcionalnost sistema in zajema akterje sistema, primere uporabe, pomembne povezave med akterji in primeri uporabe. Za ravnanje z bolj strogimi zahtevami je potrebno uvesti nove naloge v aktivnosti identifikacije zahtev. Potrebno je poiskati funkcionalne zahteve, ki opisujejo obnašanje sistema, in nefunkcionalnih zahtev, ki vključujejo lastnosti in omejitve sistema. Nadalje sledi opredelitev prioritete primerov uporabe in izgradnja primerov iz opredeljenih zahtev.

Kvaliteto podjetji, ki se ukvarjajo z razvojem programske opreme, se meri glede na njihovo odzivnost na nove ideje. Vsako podjetje, ki potrošniku ne more ponuditi novih idej, je na poti k počasnemu zatonu. Kljub temu pa implementacija prevelikega števila novih idej pomeni preveliko porabo razvojnih virov. Pred začetkom vsakega novega razvoja je potrebno oceniti tveganje, ki ga prinaša implementacija nove ideje. Najbolj preudarna pot je formalizacija proizvodnega procesa in jasna pot od koncepta do proizvoda (Booch, 1994, str. 251).

3.5.1.2 ANALIZA

Analiza se začne v začetni fazi in odigra glavno vlogo v fazi zbiranja zahtev poenotenega procesa. Namen analize je določiti opis problema. Opis mora biti zaključen, dosleden, čitljiv, pregledan s strani nekoga tretjega in preverjen v luči izvedljivosti. Lahko rečemo, da je naloga analize določiti model obnašanja sistema. Poudariti je potrebno, da je naloga analitika omejena na obnašanje modela in ne na njegov izgled, kateri je domena načrtovalca. Pri analizi poskušamo opredeliti model z identifikacijo razredov in objektov, njihovih vlog, odgovornosti in sodelovanj. Drugače pa je pri načrtovanju, kjer ustvarimo izdelek, ki omogoči obnašanje zajeto v modelu analize.

Obnašanje sistema ali programske rešitve lahko zajamemo skozi vrsto scenarijev. Ključne scenarije uporabimo za ponazoritev osnovnega obnašanja, sekundarne scenarije pa za prikaz obnašanja pod posebnimi pogoji. Rezultati analize so zbrani v dokumentu, ki vsebuje zahteve in analize. Dokument predstavlja analizo obnašanja prikazano na diagramih in analizo statičnih aspektov sistema, na primer: zanesljivosti, učinkovitosti, varnosti in prenosljivosti. Poleg tega pa tudi hitro ugotovimo manjkajoče in nepopolno opredeljene zahteve. Rezultat analize pa je tudi ocena rizika, oziroma področja tehničnega rizika, kateri lahko pride do izraza v aktivnosti načrtovanja. Soočanje s problemom rizika zgodaj v razvojnem procesu nam lahko prihrani veliko težav in arhitekturnih preurejanj v kasnejših fazah razvojnega procesa.

Model analize:

- je napisan v strogo poslovnem jeziku;
- prikazuje celotno sliko sistema in se ne ukvarja z malenkostmi;
- osredotoči se le na problemski vidik, poslovne zahteve;
- mora biti uporaben za vse soudeležence v poenotenem procesu.

Z analizo zajamemo dve glavni aktivnosti: analizo problemske domene in načrtovanje scenarijev. Analiza problema služi identifikaciji razredov in objektov, ki so skupni posameznim problemskim domenam. Pred implementacijo novega sistema je koristno preučiti kakšen že obstajajoči sistem. V najboljšem primeru lahko analiza problemske domene razkrije, da nam sploh ni potrebno razvijati nove programske rešitve, temveč lahko ponovno uporabimo in prilagodimo že obstoječo (Booch, 1994, str. 253).

Načrtovanje scenarijev je prevladujoča aktivnost analize in poteka v naslednjem zaporedju:

- Identifikacija glavnih funkcionalnosti sistema in združevanje teh funkcionalnosti v skupine s podobnim obnašanjem.
- Preglednost scenarijev nam omogočijo primeri uporabe in analize obnašanja.
- Sekundarne scenarije generiramo za ponazoritev obnašanja pod posebnimi pogoji.
- Počistimo ponavljajoče vzorce scenarijev in jih združimo v splošne, generalizirane scenarije. Z vidika razrednih diagramov s te podobnosti prikazane s pomočjo asociacij.
- Na koncu dodamo nove razrede in objekte med zbrane podatke analize v kontekstu z njihovimi vlogami in odgovornostmi.

Aktivnost analize je končana, ko imamo zgrajene scenarije osnovnega obnašanja sistema. Običajno so scenariji preverjeni s strani končnega uporabnika, poznavalca problematike in sistemskega arhitekta. V povprečnem informacijskem sistemu model analize opisuje do 100 razredov analize (Arlow, Neustadt, 2002, str. 37).

3.5.1.3 NAČRTOVANJE

Naloga aktivnosti načrtovanja je razvijanje modela načrta iz modela analize. Le-ta postane prevladujoča modelna aktivnost ob koncu faze zbiranja informacij in traja do polovice faze

izgradnje. Aktivnosti analize in načrtovanja lahko potekata vzporedno, vendar pa moramo ločevati model načrta in model analize. Namen modela načrta ni zadovoljitev potreb uporabnikom, temveč omogočiti čim hitrejšo implementacijo (Arlow, Neustadt, 2002, str. 39).

Model načrta je obdelava modela analize z dodanimi podrobnostmi in tehničnimi rešitvami. Na primer, razred analize je le natančnejša skica, saj vsebuje le ključne atribute in operacije. Razred načrta pa mora biti popolnoma specificiran, vključeni morajo biti vsi atributi in operacije.

Model načrta sestoji iz:

- podsistemov načrta;
- razredov načrta;
- realizacij primerov uporabe;
- diagrama porazdelitve.

Pomemben del aktivnosti načrtovanja je uvajanje vmesnikov, ki omogočajo razdružitev našega sistema na podsisteme, kar omogoči možnost vzporednega razvijanja podsistemov. V aktivnosti načrtovanja je možno izdelati najzgodnejši primerek porazdelitvenega diagrama, ki prikazuje razporejanje razredov in podsistemov v fizična programska vozlišča. Je zelo pomemben in uporaben diagram, s katerim se v večini ukvarjamo kasneje, večinoma znotraj aktivnosti implementacije.

Razred analize razstavimo v enega ali več vmesnikov ali razredov. Pri načrtovanju se lahko zaradi uporabe nižje stopnje abstrakcije zgodi, da za en razred analize potrebujemo enega ali več razredov in/ali vmesnikov načrta.

3.5.1.4 IMPLEMENTACIJA

V aktivnosti implementacije razrede in vmesnike definirane v modelu načrta implementiramo z ustrezno izvorno kodo. Implementacija je torej proces transformacije modela načrta v izvršljivo kodo, model implementacije pa je pri tem stranski produkt. Veliko CASE orodij dovoljuje gradnjo modela implementacije neposredno iz izvorne kode z metodo vzvratnega načrtovanja[26], zato se velikokrat gradnja modela implementacije prepušča programerjem.

Gradnja modela implementacije je pomembna v primeru:

1. Generacije izvorne kode neposredno iz modela. Takšnemu postopku rečemo vnaprejšnje načrtovanje[27] in zahteva natančno definicijo izvornih datotek in komponent.

[26] ang. Reverse engineering
[27] ang. Forward engineering

2. Načrtovanja s pomočjo komponent[28] z namenom ponovne uporabe komponent. Dodelitev razredov načrta in vmesnikov v komponente postane strateško pomembna.

Model implementacije je pravzaprav le uresničitev oziroma implementacija modela načrta. Podsistemi načrta vodijo do podsistemov implementacije, razredi načrta so dodeljeni v razrede implementacije.

Model implementacije se sestoji iz dveh diagramov:

- Diagram komponent – modelira odvisnosti med odvisnostmi med programskimi komponentami, ki sestavljajo sistem.
- Diagram porazdelitve – modelira fizična programska vozlišča na katerih bo dostavljena programska oprema delovala, ter povezave med temi vozlišči. Gre za modeliranje ciljnega okolja, na fizičnem nivoju, kamor nameravamo našo programsko opremo vključimo.

3.5.1.5 TESTIRANJE

Testiranje je preverjanje, če implementacija deluje tako, kot smo načrtovali. Preverjamo delovanje funkcionalnih in nefunkcionalnih zahtev. Testiranje na nivoju komponent je opravljeno v aktivnosti implementacije, v aktivnosti testiranja pa izvedemo še integracijski in sistemski test.

Model testiranja sestavljajo testni primeri (vsebina testiranja), testni postopki (način testiranja) in testne komponente (avtomatizacija testnih postopkov). Takšen model je potrebno vedno znova dopolnjevati, saj ga lahko uporabljamo ob različnih konfiguracijah sistema.

3.5.2 GLAVNE AKTIVNOSTI PROCESA

3.5.2.1 ZAČETNA FAZA

Naloga začetne faze je začeti projekt, ugotavljati izvršljivosti in smotrnosti projekta, ter njegova tveganja. Poudarek je na projektnih zahtevah in na analizah delovnih procesov, cilj pa je preučitev poteka razvojnega cikla.

3.5.2.2 FAZA OBDELAVE

V fazi obdelave se lotimo izgradnje arhitekture projekta, ovrednotimo zahteve sistema in njihove prioritete. Sestavimo podroben plan za fazo konstrukcije. Sledi oblikovanje zahtevnika za potrebne vire, čas, opremo, osebje in stroške, ki jih bomo potrebovali v naslednjih fazah

[28] ang. Component based design

projekta. Zaradi tega, ker naslednje faze poenotenega procesa temeljijo na fazi obdelave, je prav faza obdelave ključnega pomena za projekt.

3.5.2.3 FAZA IZGRADNJE

Cilj te faze je zadovoljitev vseh zahtev izhajajočih iz analiz in načrtovanja, ter iz izdelane arhitekture projekta razviti delujoč sistem. Pri tem je potrebno paziti na celovitost arhitekture razvitega sistema, saj nenehno spreminjajoče zahteve uporabnikov velikokrat vodijo v sistem, ki je slabe kakovosti in ima visoke stroške vzdrževanja.

3.5.2.4 FAZA PREHODA

Zadnja faza poenotenega procesa nastopi takrat, ko izdelek pride v roke končnemu uporabniku. Redko je proces razvoja sistema v tej točki končan, saj velikokrat prihaja do ponavljajočih zahtev po izboljšanju in nadgradnji sistema. Naloge te faze so odpravljanje programskih napak, izobraževanje uporabnikov sistema, pisanje uporabniških priročnikov in omogočanje svetovanja za uporabnike.

3.6 UPORABLJENO CASE ORODJE

Tehnologija analiziranja, načrtovanje in izgradnje informacijskih sistemov vseskozi doživljaja velike razvojne spremembe. Te spremembe so posledica izboljšave računalniško podprtih tehnik, metod in metodologij, ki se uveljavljajo skozi vse faze razvojnega cikla sistema in dobivajo oblike zaključene tehnologije. Takšna orodja lahko imenujemo CASE orodja ali računalniško podprta gradnja programskih sistemov. Potrebno pa se je zavedati, da proizvodov, ki bi pokrivali celotni razvojni cikel trenutno še ni. Obstajajo proizvodi, ki pokrivajo ožji ali širši odsek razvojnega procesa. Nekateri so usmerjeni v začetno fazo (lower-CASE), drugi v srednji del (middle-CASE) in tretji v zaključno fazo (upper-CASE) razvojnega procesa (Kovačič, Vintar, 1994, str. 278).

Rhapsody je CASE programsko orodje namenjeno razvoju vgrajenih računalniških sistemov delujočih v realnem času. Rhapsody lahko razvrstimo kot CASE orodje, ki je usmerjeno v srednjo in zaključno fazo razvojnega procesa. Iz UML modelov je sposobno generacije ANSI[29] C kompatibilne programsko kodo, ki je optimizirana za uporabo v vgrajenih računalniških sistemov delujočih v realnem času. Kot smo rekli, je Rhapsody sposoben generacije iz UML diagramov, s čimer omogoča sinhronizacijo modela in programske kode. S tem omogoča gradnjo večjega števila UML modelov, testiranje, odpravljanje napak v programski kodi, ponovno

[29] ang. American National Standardization Institut

analizo in večkratno prevajanje modela. Potrebno pa se je zavedati, da Rhapsody ni sposoben generacije programske kode iz vseh UML diagramov. Popolne generacije kode so sposobni objektni diagrami, diagram komponent, diagrami stanj in diagrami aktivnosti. Nepopolne generacije kode so sposobni diagrami primerov uporabe, diagrami zaporedja in diagrami sodelovanja. Nepopolna generacija kode pomeni, da v programske kode ne generirajo vsi elementi UML diagrama, ampak le nekateri (Rhapsody User Guide, 1997, str. 10).

4 RAZVOJ PROGRAMSKE REŠITVE

V tem poglavju je prikazan celotni proces razvoja programske rešitve od analize zahtev do implementacije končne programske rešitve v izdelek. Celotni sistem programske rešitve, programski moduli in programska platforma, ki predstavlja abstrakcijo fizičnega nivoja sistema so zamišljeni modularno (glej Sliko 61, na strani 73). Moja naloga je narediti programsko rešitev, ki bo uporabljala funkcionalnost programskih modulov in programske platforme ter delovala v skladu z zahtevami podanimi iz marketinga.

V procesu razvoja sem si pomagal z UML orodjem Rhapsody, saj sem celotno programsko rešitev modeliral s pomočjo UML diagramov:

- faza zahtev (diagrami primerov uporabe za modeliranje funkcionalnih zahtev),
- faza analize (diagrami zaporedja za modeliranje scenarijev za identifikacijo razredov analize in objektni diagram model arhitekture programske rešitve),
- faza načrtovanja (diagrami zaporedja za bolj strukturirane razrede načrtovanja in diagrami prehajanja stanj s katerimi modeliramo obnašanje posameznih delov programske rešitve),
- in faza implementacije (diagrami komponent za modeliranje arhitekturno zgradbo končne programske rešitve).

4.1 ŠIRŠI POGLED

Iskraemeco d. d. je eden največjih evropskih proizvajalcev naprav za merjenje in obračun električne energije. Razvija, proizvaja in trži naprave in sisteme, uporabne v gospodinjstvih, industriji, ter v elektroenergetskih, proizvodnih in prenosnih sistemih.

Razvojne smernice so usmerjene v zadovoljevanje potreb uporabnikov na prostih trgih[30] električne energije. Dolgoročni cilji podjetja pa je povečanje tržnega deleža na obstoječih trgih in osvojiti nove trge, ter ostati eden vodilnih svetovnih proizvajalcev naprav za merjenje električne energije.

Izdelkov Iskraemeca ni mogoče kupovati na trgu, temveč se za naročila izdelkov navadno odločajo distribucije električne energije na podlagi razpisov. Podjetje je zaradi tega podrejeno

[30] Dereguliran trg električne energije je takšen trg, kjer si lahko porabnik sam izbira svojega dobavitelja električne energije.

prilagajanju številnim posebnim željam kupcev. Zaradi takšnega odnosa kupcev je potrebno biti še posebej odziven pri procesu razvoja programske rešitve. Zahteve za razvoj neke specifične programske rešitve lahko pridejo nenadoma in v takšnih primerih je "*time-to-market*" izredno pomemben.
Vsak razvijalec programskih rešitev ima najrajši klasičen proces razvoja programske rešitve. V tem primeru je vsak korak razvoja programske rešitve izveden samo enkrat, zahteve se napišejo, sledi analiza, načrtovanje in implementacija. Vendar pa je realnost drugačna, zahteve se nenehoma menjajo in zaradi tega je potrebno aktivnosti posameznih faz ponavljati, dokler ne pridemo do želene programske rešitve. Pride do večkratnih iteracij razvojnih faz. Poveča se vključenost naročnika in uporabnikov v razvojni proces, zahteve se lažje spreminjajo in pričakovanja so lažje dosežena. Pri razvoju programske rešitve za elektronski števec električne energije je bil cilj približati se poenotenem procesu.

Pri razvoju izdelka je bil cilj uporabiti prednosti vseh šestih praks združenih v poenotenem procesu:

- iterativni razvoj,
- upravljanje zahtev,
- uporaba arhitekture komponent,
- vizualno modeliranje,
- preverjanje kakovosti,
- in nadzor nad spremembami.

4.2 ZAHTEVE

Zahteve opisujejo, **kaj** naj sistem dela in predstavljajo dogovor med razvijalci in naročniki ali uporabniki sistema. Zahteve se delijo na *funkcionalne zahteve*, predstavljene s primeri uporabe, in dodatne*, nefunkcionalne zahteve*[31]. Model primerov uporabe je najpomembnejši rezultat faze zajemanja zahtev, saj model uporabe služijo kot rdeča nit skozi vse cikle razvoja. Isti model primerov uporabe se uporablja med zajemanje zahtev, analizo, načrtovanjem in testiranjem (Modeliranje s pomočjo objektnega modeliranja UML, 2004).

Namen aktivnosti zajemanja zahtev je zajeti in ovrednotiti zahteve s poudarkom na uporabnosti. Aktivnosti zajemanja zahtev:

- določitev poslovnih ciljev,
- oblikovanje vizije,
- izvabljanje potreb interesentov,
- upravljanje odvisnosti in sprememb,
- iskanje akterjev in primerov uporabe,
- strukturiranje primerov uporabe,

[31] Nefunkcionalne zahteve o delovanju in zanesljivosti sistema.

- določitev prioritet primerov uporabe,
- podroben opis primerov uporabe,
- modeliranje in prototip uporabniškega vmesnika, v našem primeru zunanjega izgleda elektronskega števca.

Poslovni cilj vsakega proizvodnega podjetja je prodati izdelek, s katerim bi čim več zaslužili. Potrebno je izdelati izdelek, ki se bo dobro prodajal in čim hitreje prišel od ideje do kupca. Do konkretnih informacij, ki se uporabljajo za določanje poslovnih ciljev, oblikovanje vizije in iskanje idej, pridemo s pomočjo:

- tržnih raziskovanj,
- ideje prihajajo predvsem iz marketinga,
- zunanjih sodelavcev (podjetja za analizo trga),
- sejmi, razstave,
- "juriš na zamisli"[32] .

Želimo izdelati elektronski števec električne energije, ki se bo uporabljal v industriji. Njegov namen ni samo merjenje električne energije, ampak ima še vrsto drugih funkcionalnosti. Za zagotovitev teh funkcionalnosti je potrebno imeti kakovostno programsko rešitev, ki je predmet našega razvojnega procesa.

Identificiramo tri vrste akterjev:

- *Uporabnik* elektronskega števca je navadno tisti, ki odčitava izmerjene in zabeležene podatke. Odčitavanje poteka preko zaslona, optične sonde ali komunikacije s pomočjo osebnega računalnika.
- *Merjenec* je tisti, ki ga s pomočjo elektronskega števca merimo. Zagotoviti je potrebno ustrezno merilno točnost.
- Preko *električnega omrežja* elektrodistribucija dobavlja električno energijo. Zaradi uporabe merilnih transformatorjev[33] je potrebno vedeti na katerem nivoju omrežja poteka meritev. Poleg tega pa je potrebno beležiti kakovost dobavljene energije.

4.2.1 PODROBEN OPIS PRIMEROV UPORABE

Diagram primerov uporabe za elektronski števec električne energije lahko strukturiramo na več manjših diagramov primerov uporabe:

- diagram primerov uporabe merjenih veličin,
- diagram primerov uporabe izračunanih veličin,
- diagram primerov uporabe prikaza na zaslon,

[32] ang. Brainstorming
[33] Merilni transformatorji se uporabljajo v visokonapetostnih tokokrogih zaradi zmanjšanja merjenih tokov in napetosti. Zmanjšane tokove in napetosti upoštevamo pri končnem izračunu merjenih vrednosti.

- diagram primerov uporabe branje tipkovnice,
- diagram primerov uporabe priključitve števca v omrežje,
- diagram primerov uporabe različnih vrst komunikacij,
- diagram primerov uporabe točnosti meritev.

Slika 24: Diagram primerov uporabe zahtev za elektronski števec električne energije

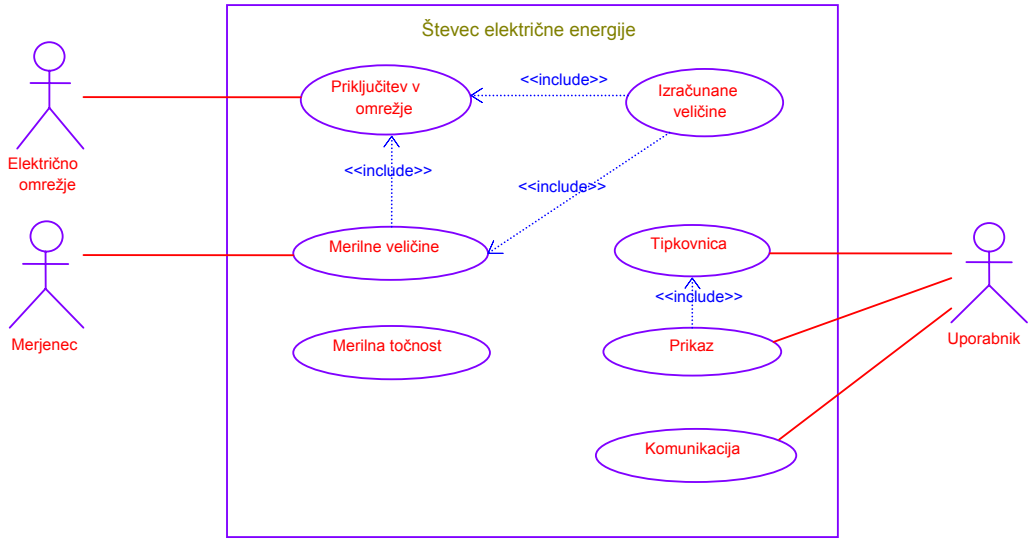


Diagram primerov uporabe merjenih veličin pove, katere veličine mora števec znati meriti. Elektronski števec električne energije mora neposredno meriti samo napetost, tok in energijo.

Slika 25: Diagram primerov uporabe zahtev merjenih veličin

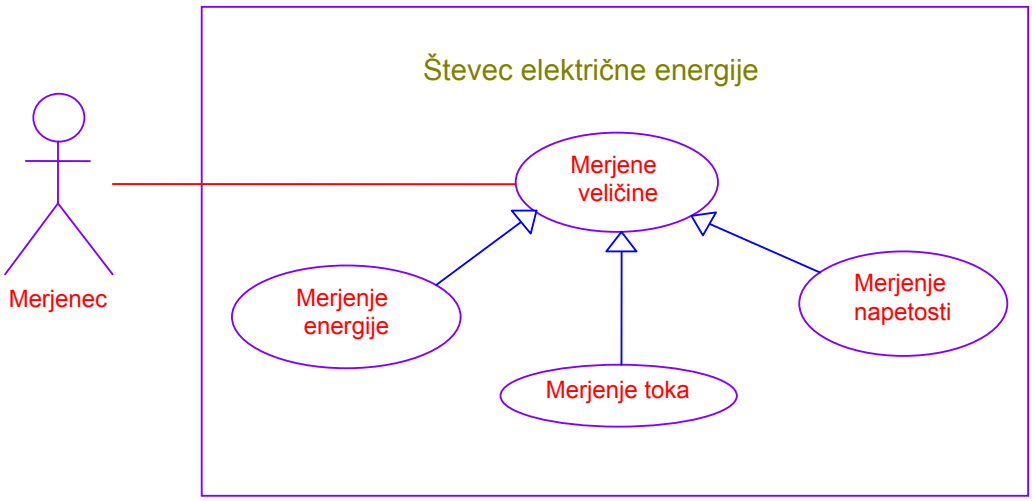


Diagram primerov uporabe prikaza na zaslon pove, katere vrednosti mora števec znati izpisovati na zaslon: električno energijo, moč, napetost, tok, rekorder meritev, čas, datum in sistemske statuse.

Slika 26: Modela primera uporabe zahtev prikaza na zaslon

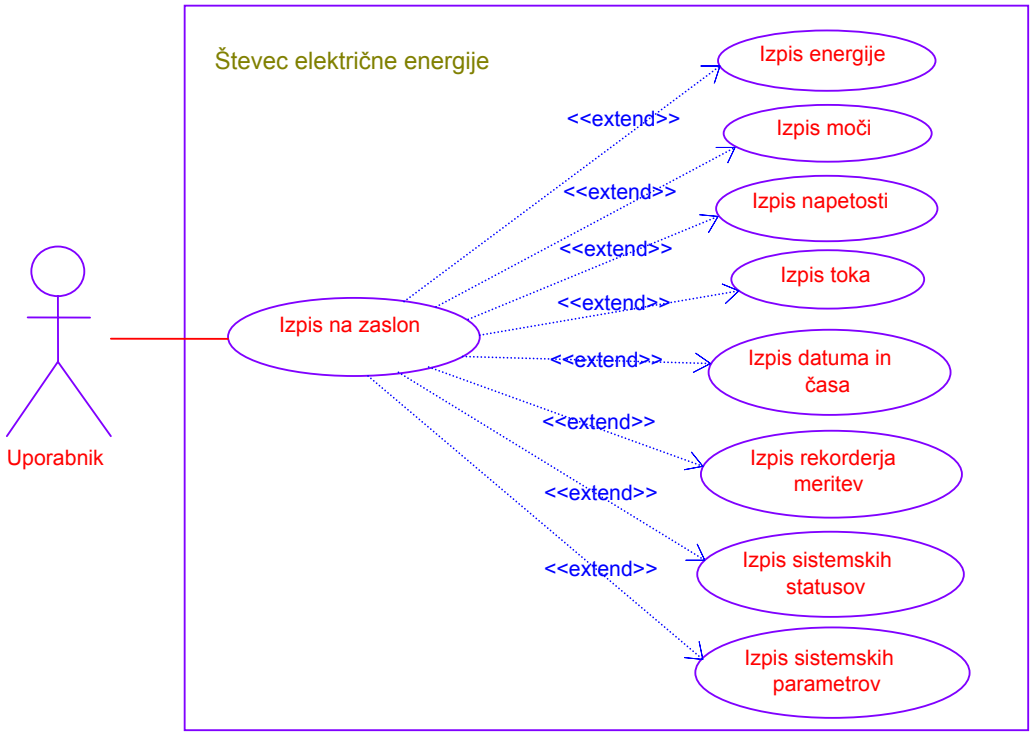


Diagram primerov uporabe branje tipkovnice prikazuje način, kako uporabnik s pomočjo pritiska na tipko vpliva na delovanje elektronskega števca .

Slika 27: Slika: Diagram primerov uporabe zahtev za branje tipkovnice

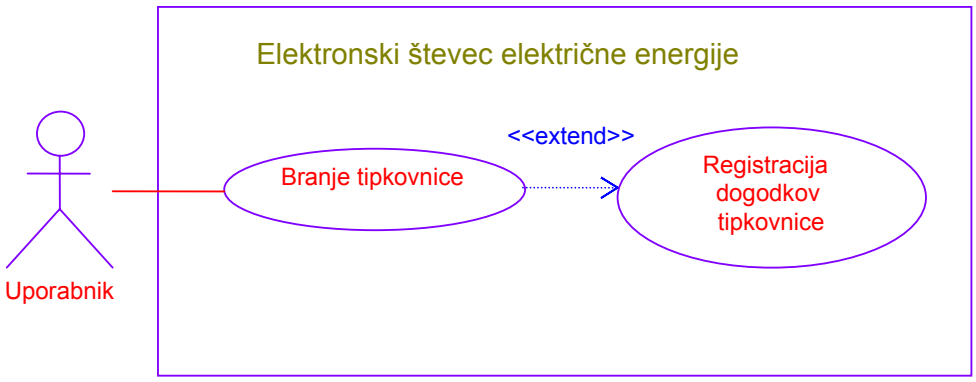


Diagram primerov uporabe izračunanih veličin pove, katere veličine mora števec izračunati, da izpolni funkcijske zahteve za industrijske elektronske števce električne energije. Moč se

izračunava iz izmerjene energije in merilne periode, morebitna slaba kakovost dobavljene energije se beleži v posebne registrirnike[34], močnostni faktor pa pomeni razmerje delovne energije in jalove energije.

Slika 28: Slika: Diagram primerov uporabe zahtev izračunanih veličin elektronskega števca

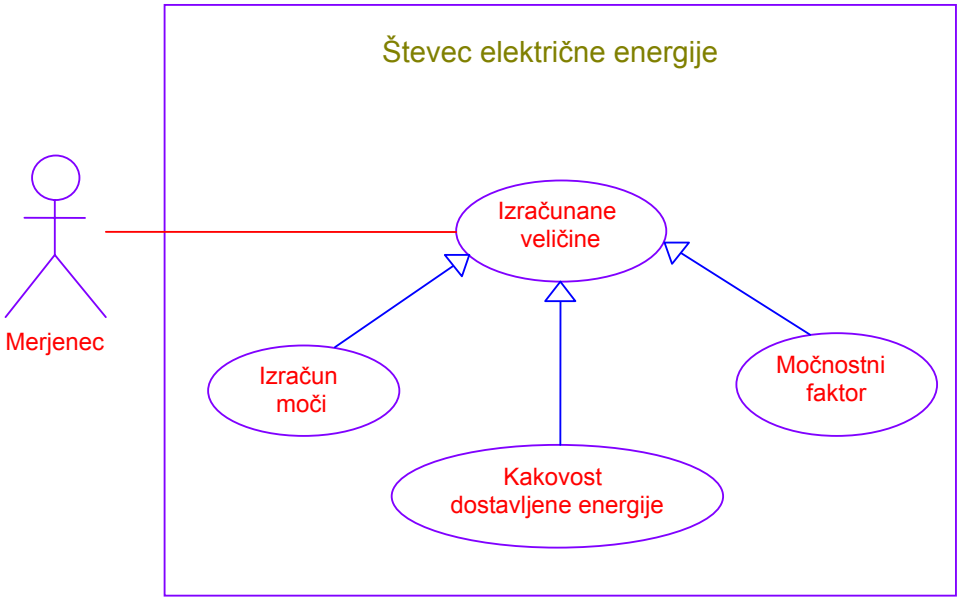
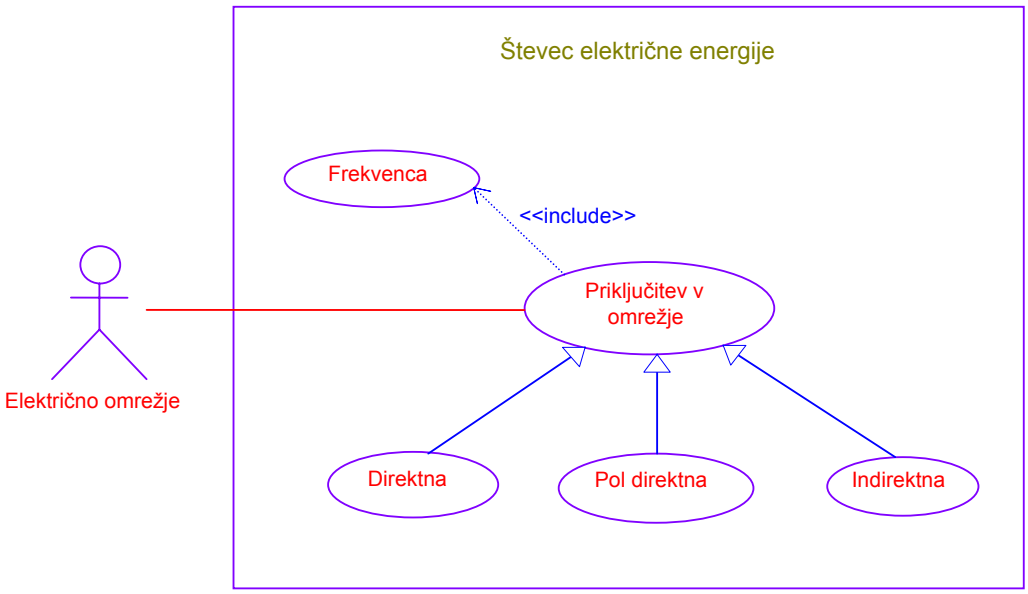


Diagram primerov uporabe načina priključitve števca nam pove, na kakšen način je lahko števec električne energije priključen v električno omrežje. Poznamo neposredno priključitev v električno omrežje in priključitev v omrežje preko delilnih transformatorjev[35] (glej Slika 29, na strani 42).

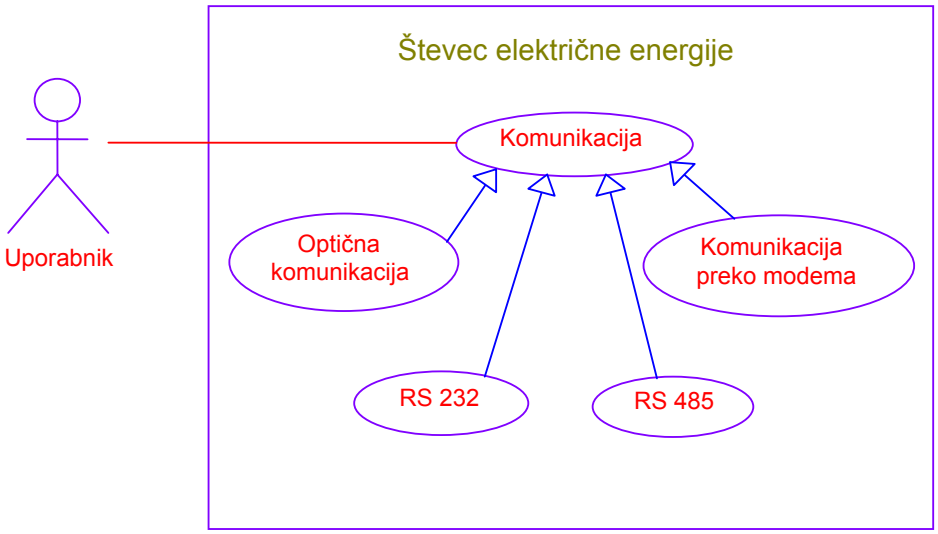
Diagram primerov uporabe nam pove, katere vrste komunikacije števec pozna. Funkcijske zahteve zahtevajo sposobnost komunikacije preko optične sonde[36], serijskih komunikacij RS 232 in RS 485, ter komunikacije preko modema.

[34] Delovna energija je realno energijo in predstavlja porabo na uporovnih porabniki (električne peči).
Jalova energija je imaginarna energija, ki nastane z zamikom toka in napetosti. (električni motorji). Takšna energija je dražja od delovne energije.
[35] Priključitev v visokonapetostna omrežja
[36] ang. Infrared IR

Slika 29: Diagram primerov uporabe zahtev za priključitev elektronskega števca v električno omrežje

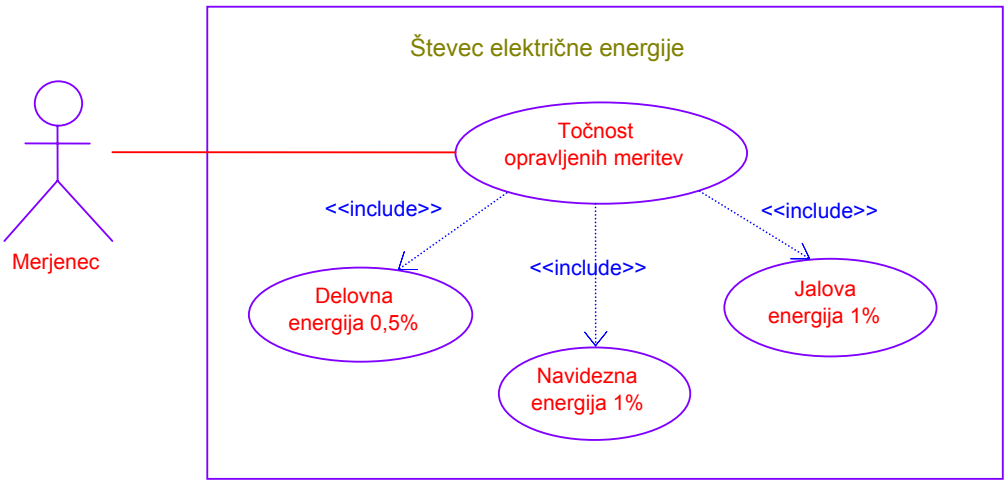


Slika 30: Diagram primerov uporabe zahtev za realizacijo komunikacije elektronskega števca električne energije



Števec električne energije mora izpolnjevati stroga merila točnosti izmerjenih veličin, predpisane so meje točnosti za delovno, jalovo in navidezno energijo.

Slika 31: Diagram primerov uporabe zahtev za točnost meritev elektronskega števca električne energije



Vse funkcijske zahteve izvirajo iz mednarodnim standardom, ki veljajo za industrijske elektronske števce električne energije (Zahtevnik za elektronske industrijske števce, 2001). Tudi izgled uporabniškega vmesnika, v našem primeru zaslona elektronskega števca, je pogojen s standardom, ki pogojuje izgled zaslona industrijskega števca.

4.2.2 MODELIRANJE IN PROTOTIP STROJNE OPREME

Pred aktivnostjo analize naredimo prototip strojne opreme, na kateri se bo izvajala naša programska rešitev. V sami analizi moramo preučiti tudi proizvodne stroške in cene strojne opreme. Kljub temu, da je cilj naloge programska rešitev, je potrebno vedeti nekaj osnovnega o delovanju strojne opreme. Potrebno se je zavedati, da programska rešitev ne more obstajati brez ustrezne strojne opreme.

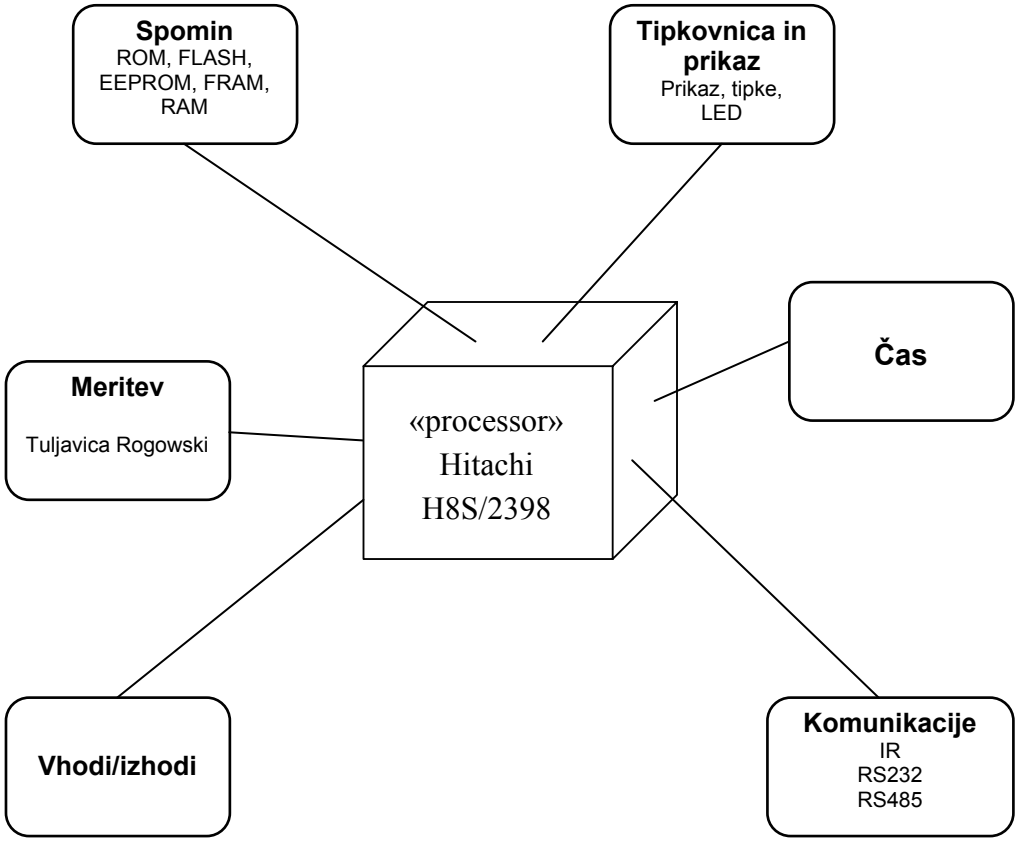
Osnovne komponente strojne opreme, vključene v elektronski števec električne energije:

- Hitachi **mikroprocesor** H8S/2398 z 8-bitnim naslavljanjem, hitrostjo ure 20MH in velikostjo statičnega pomnilnika ROM je 256kbytov, ter velikostjo dinamičnega pomnilnika 8kbytov.
- **Napajanje** omogočeno preko kondenzatorskega ali preklopnega[37] napajalnika.
- **Datum in čas** omogoča RTC[38] vezje, ki se nahaja na glavni tiskani plošči.
- **Merilni del** izveden preko tuljavice Rogowski, ki omogoča pretvorbo porabljene moči v napetost.
- **Vhodno/izhodni** del in **komunikacijski** del sta izvedena kot modularni vmesnik.

[37] ang. switch
[38] ang. real time clock

- **Grafični prikazovalnik** je izveden kot LCD[39] segmentni prikazovalnik, krmiljen s pomočjo vgrajenega krmilnika, ki je sposoben preprostih grafičnih funkcij: prižiganje, ugašanje in utripanje segmentov.
- **Takt** mikroprocesorskih prekinitev[40] je vsakih 20ms.

Slika 32: Modeliranje programske platforme



Vpogled v strojno opremo nam lahko osvetli našo problematiko iskanja primerne programske rešitve, ki je s strojno opremo neposredno povezana. Potrebno se je zavedati, da je ena od lastnosti objektno orientiranega sistema neposredna povezanost s problematiko. Programska platforma predstavlja navidezen sistem, ki ustreza abstrakciji strojne opreme. Sprememba sistemske strojne opreme vpliva le na abstrakcijo spodnjih nivojev sistema.

Uporaba tako imenovane programske platforme je pri vgrajenih računalniških sistemih, kakršen je naš skorajda obvezna. Programska platforma omogoča skrivanje lastnosti strojne opreme, ki so preveč odvisne od posamezne implementacije in so podvržene nenehnim spremembam. Primer takšne spremembe je zamenjava 8kbyte RAM dinamičnega spomina s 16kbyte RAM dinamičnega spomina. Popolna nespremenljivo bi bilo zaradi tako malenkostne spremembe

[39] ang. liquid crystal display
[40] ang interrupt

strojne konfiguracije sistema spreminjati tudi samo programsko rešitev. Programska platforma ni nič drugega kot ograditev delov sistema, ki so odvisni od strojne konfiguracije v posebne razrede oziroma vmesnike. Na primer: Dostop do trenutnega časa je omogočen preko posebnega razreda ali z uvedbo programskih modulov, oba principa služita kot programska vmesnika[41]. Pri razredu lahko dostopamo do njegovih javnih operacij in argumentov, v primeru uporabe programskega modula pa definiramo njegove javne operacije in argumente v API[42] servisov.

Abstrakcija strojnega dela s pomočjo programske platforme:

Merilni del	⇨	`Measure API`
Tipkovnica in prikaz	⇨	`Pconsole API`
Komunikacije	⇨	`Serial API`
Čas	⇨	`Ptime API`
Pomnilnik	⇨	`Pfile API`
Vhodno/Izhodni del	⇨	`PIO API`

4.2.3 MERITEV

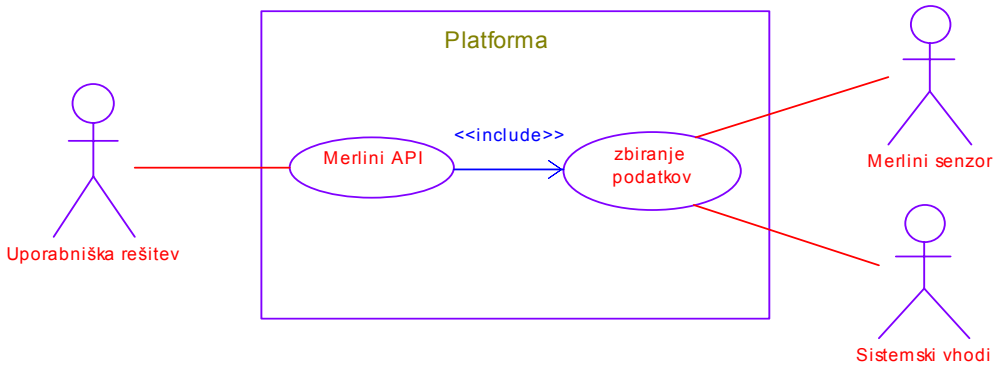
Merilna funkcionalnost števca je vsebovana v modulu platforme imenovanem `Measure API`, ki omogoča dostop do merilnih dela števca. Njegova naloga je zagotoviti standardiziran dostop do merilnih podatkov, ki jih zagotavljajo različni merilni principi.

Merilni modul platforme zajema merilne servise, ki združujejo zajemanje vrednosti merilni količin kot so:

- energija,
- moč,
- napetost,
- frekvenca,
- višje harmonske napetosti in njihovi statusi,
- prisotnost posamezne fazne napetosti,
- zaporedje faz,
- smer pretoka energije.

[41] ang. program interface
[42] ang. application programming interface

Slika 33: Model merilnega dela platforme

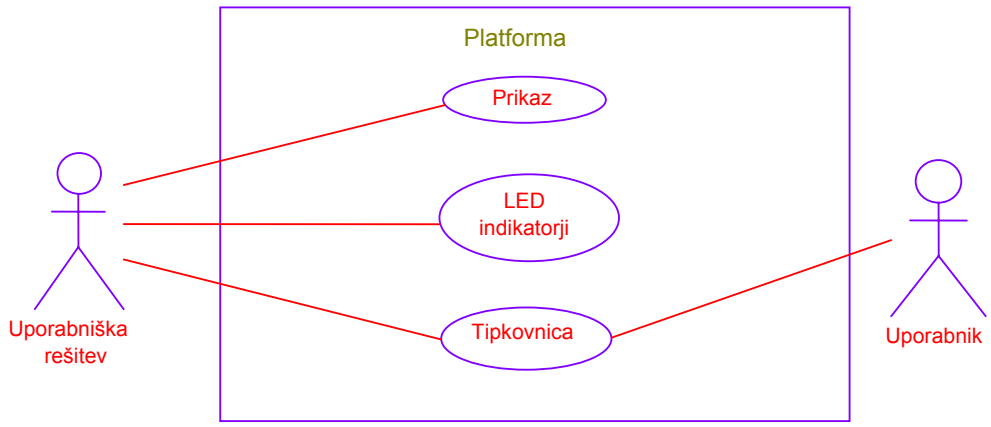


4.2.4 TIPKOVNICA IN PRIKAZ

Tipkovnica in prikaz predstavljata uporabniški vmesnik števca, njuna funkcionalnost je zajeta v `Pconsole API`. `Pconsole API` zajema servise tako imenovane konzolne servise platforme, ki vključujejo servise prikaza, tipkovnice in LED[43] indikacije. Aplikacija mora za uporabo teh servisov ugotoviti zmožnosti konzole in glede na te zmožnosti uporabljati konzolne servise.
`Pconsole API` je sestavljen iz:

1. *Prikaza*: zaslon za prikaz je sestavljen iz niza segmentov, ki so lahko v stanju vklopljeni ali izklopljeni. Poleg tega pa so lahko v normalnem ali v utripajočem stanju.
2. *LED indikacije*: indikacija je sestavljena iz niza LED indikatorjev, vsak od teh indikatorjev lahko deluje v normalnem ali utripajočem načinu delovanja.
3. *Tipkovnice*: tipkovnica je sestavljena iz tipk. Pritisk tipke ali njena sprostitev predstavlja dogodek konzole.

Slika 34: Model tipkovnice in prikazovalnega dela platforme

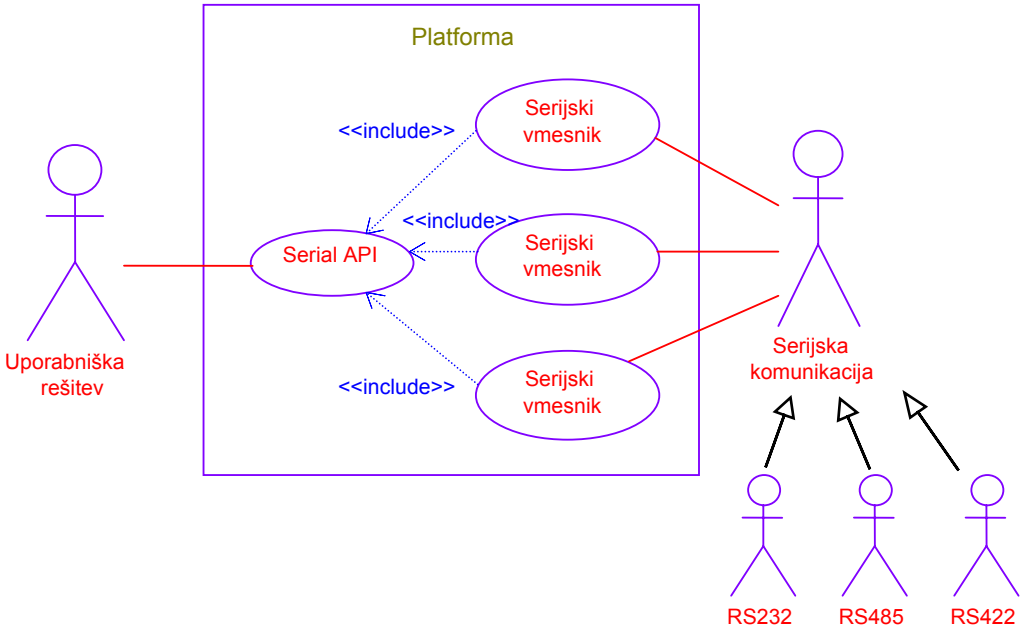


[43] ang. light emitted diode

4.2.5 KOMUNIKACIJA

Za serijske asinhrone komunikacije skrbi `Serial API` servisi platforme in so podpora za UART[44] vmesnike platforme. UART serijski vmesniki omogočajo serijski prenos podatkov preko RS232, RS485, RS422 ter drugih serijskih vmesnikov. `Serial API` funkcije nadzorujejo vse serijske vmesnike platforme in omogočajo enostavno priključitev modulov za različne komunikacijske protokole. Možno je nastavljanje parametrov serijskega prenosa podatkov (hitrost prenosa, pariteta, število podatkovnih in stop bitov), pošiljanje in sprejemanje podatkov in nadzor nad signali serijskega procesa. Signali so v skladu z RS232 standardom.

Slika 35: Slika: Model serijske komunikacije v platformi

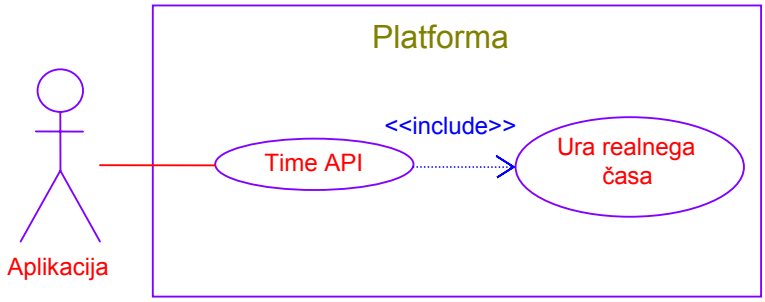


4.2.6 ČAS

Delo z realnim časom omogoča `Ptime API` . Interno je sistemski čas v GMT in se za potrebe prikaza pretvori v lokalni čas. Pretvorba se izvrši glede na časovno zono, ki je nastavljena. Čas je zapisan v več formatih, ki omogočajo različne operacije pri izračunu časovnih funkcij. Glede na to, v kateri časovni zoni se nahajamo, lahko lokalni čas prikažemo tudi kot poletni čas. Zaradi sinhronizacije na poletni čas so implementirani dvakratni premiki časa. Dodano funkcionalnost nam ponudi časovna značka, ki predstavlja število sekund pretečenih od 1. 1. 1970. Ta oblika časa je namenjena različnim časovnim operacijam in zaradi svoje kompleksnosti tudi označevanju raznih dogodkov v sistemu.

[44] ang. universal asynchronous receiver/transmitter

Slika 36: Slika: Model ure realnega časa v platformi



4.2.7 SPOMIN IN VHODNO/IZHODNI DEL

Spominske sposobnosti platforme predstavljajo statični pomnilniki kot so ROM in dinamični pomnilniki, ki pa se ločijo glede na hitrost dostopa informacij. Dinamični pomnilniki RAM in FRAM[45] omogočajo hitrejše čase dostopa, FLASH EEPROM pa daljše ob nekoliko sprejemljivejši ceni.

Vhodni del števca je namenjen sprožanju raznih dogodkov ali pa polnjenju podatkovnih registrov. Z izhodnim delom pa lahko krmilimo razne naprave, ki jih priključimo na števec. Navadno so to večji porabniki energije, za katere je smiselno delovanje v cenejši tarifi električne energije.

4.3 ANALIZA

Naloga aktivnosti analize je osvetliti obnašanje sistema, preučevanju arhitekture sistema pa je namenjena aktivnost načrtovanja. V analizi poskušamo modelirati naš problem s pomočjo identifikacije razredov in objektov, v načrtovanju pa omogočimo delovanje, ki smo ga predpisali z modelom analize.

Ker razvoj naše programske rešitve poteka v programskem jeziku ANSI C, ne bomo uporabljali izraza razred, temveč izraz objektni tip. Objektni tip vsebuje skoraj vse funkcionalnosti, ki jih ima razred, razen generalizacije in specializacije. Naš naslednji korak je identifikacija objektnih tipov analize, ki prikazujejo abstrakcijo problemske domene in realizacijo diagramov primerov uporabe.

4.3.1 KONZOLA

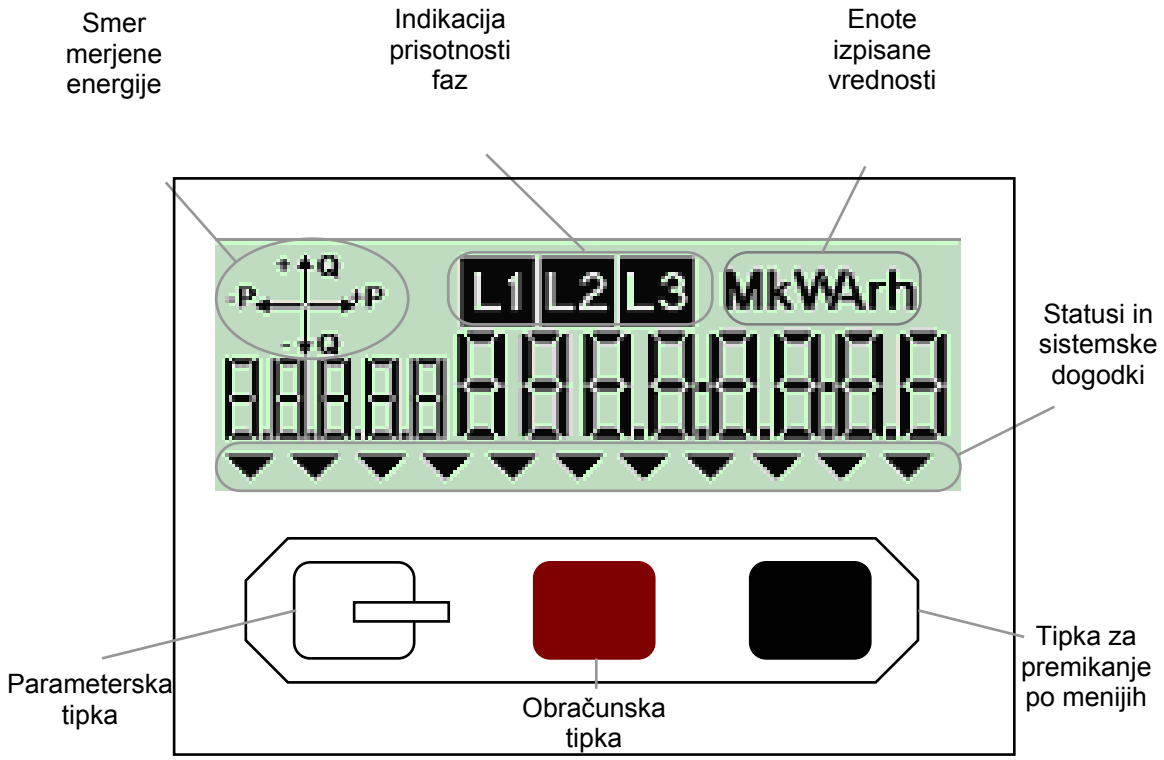
Elektronski števec električne energije je namenjen beleženju porabljene energije. Porabo energije pa mora biti uporabnik sistema zmožen tudi odčitati, zato je v napravo vgrajen zaslon preko

[45] ang. ferromagnetic random access memory

katerega lahko uporabnik vidi količino porabljene energije. Za preprosti števec električne energije, oziroma gospodinjski števec, ki se nahaja v vsakem gospodinjstvu, je naloga le merjenje delovne energije v visoki in nizki tarifi. Industrijski števec električne energije pa ima nekoliko težjo nalogo, saj mora znati prikazati tudi trenutne vrednosti merjenih veličin, potrebna pa je tudi registracija pomembnejših izmerjenih veličin in raznih dogodkov.

Kaj mora uporabniška rešitev prikazovati na zaslonu je definirano z zahtevnikom VDEW[46] za industrijske števce. VDEW je združenje nemških elektrogospodarstev, ki že desetletje ali dve narekuje hitrost razvoju industrijskih števcev v Evropi. Tudi naše podjetje je prisiljeno slediti njihovim razvojnim smernicam, tako pri razvoju programske rešitve kot tudi pri realizaciji strojne opreme.
Objektni tip `Console` je vmesnik med uporabnikom in sistemom, oziroma elektronskim števcem električne energije. Njena osnovna funkcija je izpis na LCD[47] zaslon in upravljanje z napravo tipkovnico.

Slika 37: Zaslon elektronskega števca je določen z zahtevnikom VDEW za industrijske števce



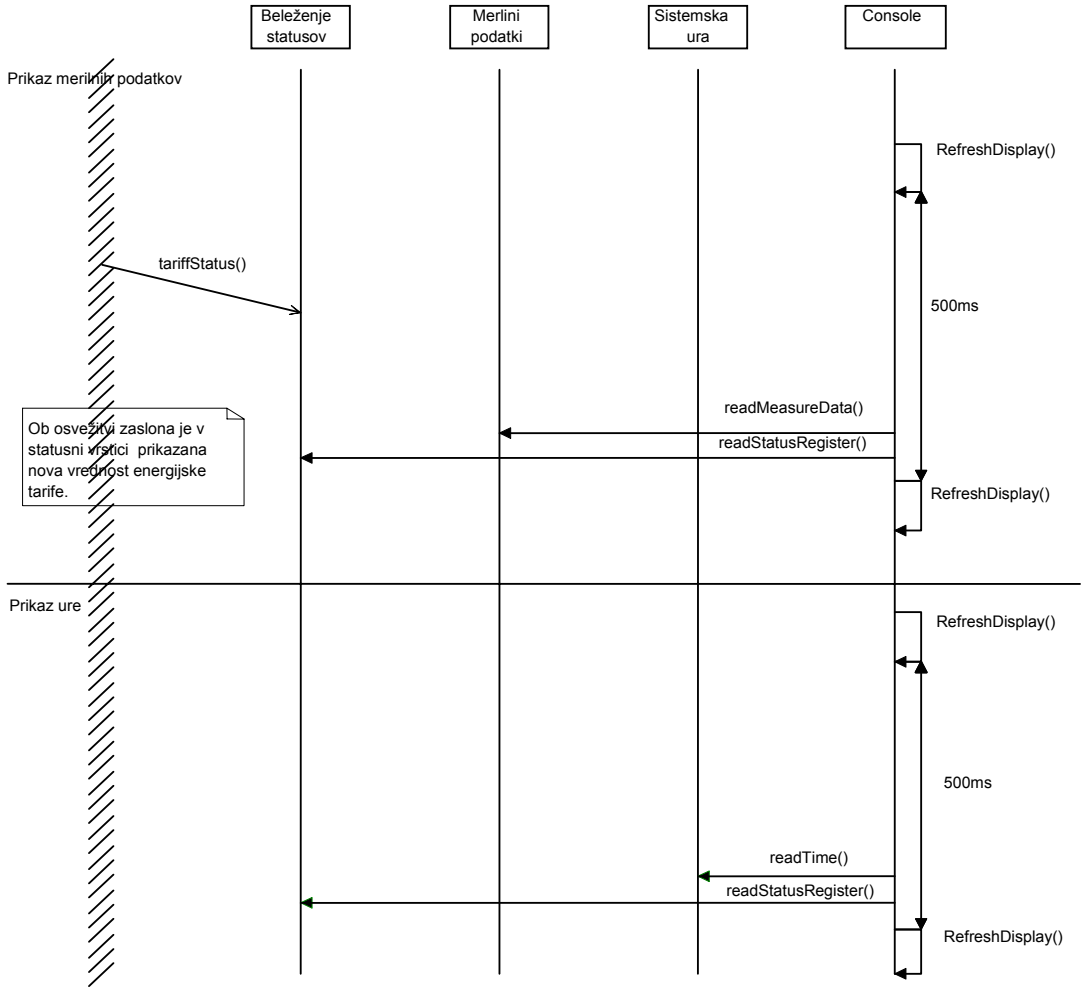
Zaslon mora znati prikazati:

- pomembne merilne podatke,
- indikacijo prisotnosti faz in smer merjene energije,
- statuse sistemskih dogodkov.

[46] nem. Verband der Elektrizitätswirtschaft
[47] ang. liquid crystal display

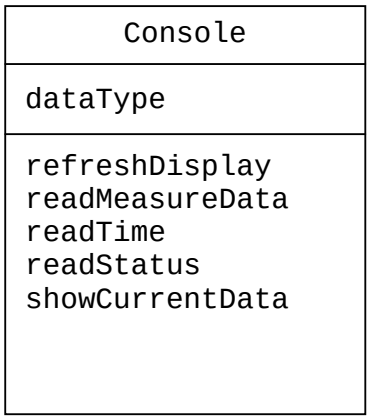
Scenariji:
S pomočjo diagrama zaporedja prikažemo scenarij za prikazovanje merilnih podatkov in časa. V prvem scenariju prikazujemo merilne podatke, ki jih preberemo s pomočjo operacije `readMeasureData`. V drugem scenariju pa prikazujemo trenutni čas, ki ga iz sistemske ure preberemo s pomočjo operacije `readTime`. Podatke na zaslonu prikazujemo s pomočjo operacije `refreshDisplay`. Pred vsakim klicem operacije `refreshDisplay` poskrbimo za branje sistemskih statusov `readStatus`, indikacije trenutne prisotnosti faz in smeri merjene energije `showCurrentData`.

Slika 38: Slika: Scenariji izpisovanja na zaslon



Na podlagi scenarija izpisovanja na zaslon lahko uvedemo nov objektni tip analize z imenom `Console`. Dodamo samo en atribut, ki bo izbiral med prikazovanjem merilnih vrednosti in prikazom časa, ter med preučevanjem scenarija identificirane operacije.

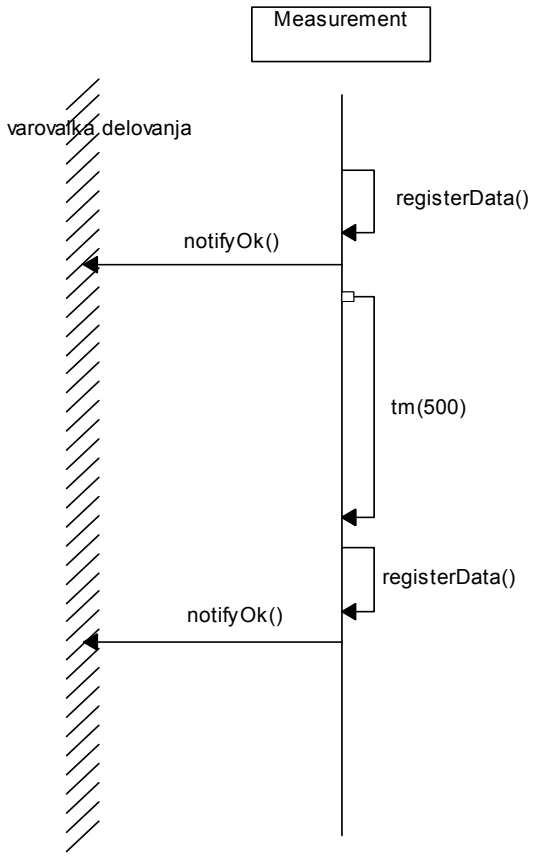
Slika 39: Objektni tip analize z imenom Console



4.3.2 REGISTRACIJSKI DEL

Registracijski del je najpomembnejši in najzahtevnejši del našega sistema. Njegova naloga je merjenje merilnih veličin in registracija podatkov ter različnih dogodkov. Pomembna je tudi časovna obdelava podatkov, saj se opravila med seboj ne smejo motiti. Obdelava opravil v operacijskem sistemu je sinhrona ali zaporedna, kar pomeni da mora operacijski sistem opravilo končati in šele nato lahko začne naslednje opravilo.

Slika 40: Scenarij za beleženje izmerjenih podatkov in proženje varovalke delovanja

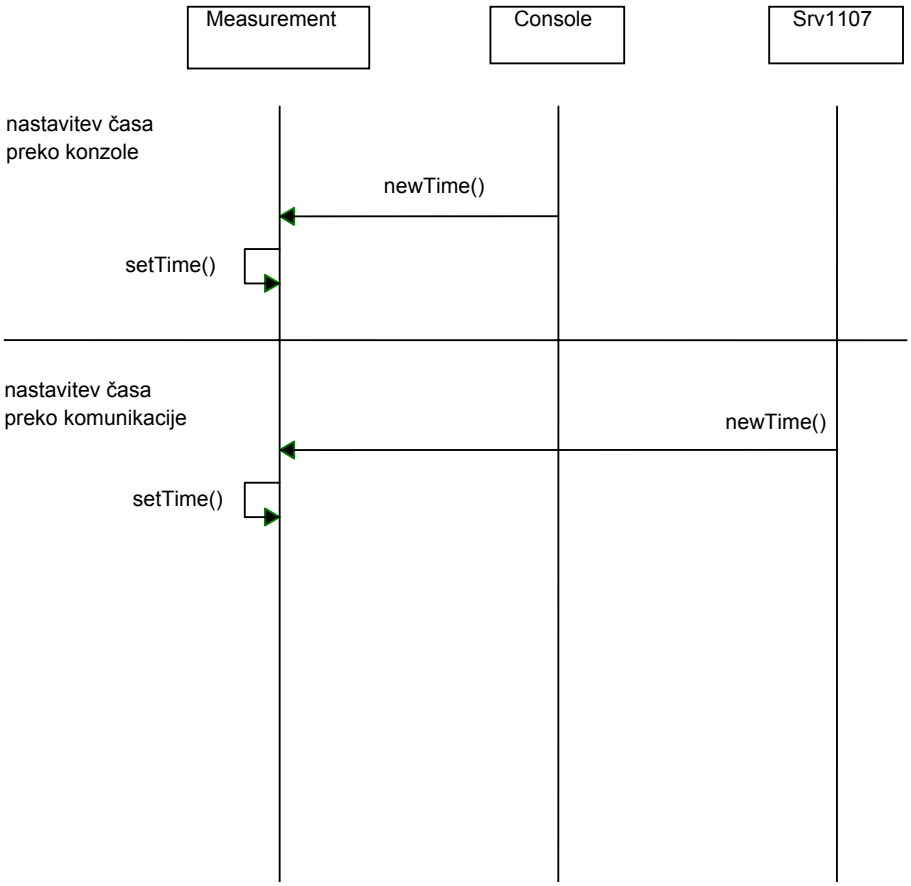


Naloge registracijskega dela:

- registracija merilnih veličin,
- varovalka delovanja sistema,
- nastavljanje ure,
- zagotavljanje varnostne kopije pomembnih podatkov,
- periodični klic programskih modulov.

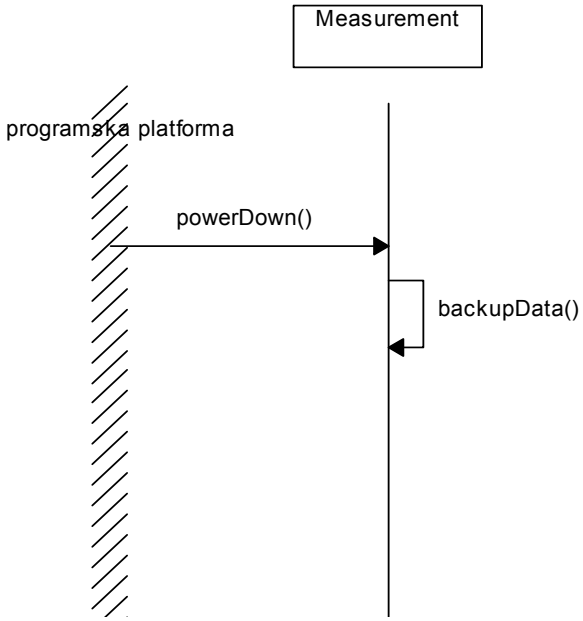
Operacija `registerData` periodično beleži izmerjene podatke v posebne registrirnike (energije/moči in napetosti/toka). Varovalka delovanja sistema ni nič drugega kot števec, ki mora biti klican vsako sekundo. V primeru, da števec delovanja ni klican v zahtevanem času, bo sistem izklopljen in ponovno zagnan (zgodi se `reset` sistema). Operacija `notifyOK` zagotovi klic varovalke sistema.

Slika 41: Slika: Scenarij nastavljanja časa



Nastavljanje ure lahko prožimo iz dveh objektov `Console` in `Srv1107` s pomočjo operacije `newTime`. Dejanska nastavitev ure je izvedena v registracijskem delu s klicem `setTime`, kjer se izvede tudi preverjanje pravilnosti nastavljenega časa.

Slika 42: Slika: Scenarij generacije varnostne kopije



Ob primeru izklopa sistema mora registracijski del s pomočjo operacije `backupData` narediti varnostno kopijo pomembnejših merilnih podatkov. Nikakor pa ne smemo pozabiti minutnega takta, ki ga registracijski del daje programskim modulom. Klici morajo biti izvršeni vsako minuto s operacijo `callModule`.

Ustvarimo nov objektni tip `Measurement`, ki bo služil registraciji pomembnih merilnih podatkov in nadzoru pravilnega delovanja sistema. Dodamo še atribut `timeSource`, ki nam pove izvor nastavitve ure.

Na podlagi scenarijev sem identificiral nov objektni tip analize `Measurement`.

Slika 43: Objektni tip analize `Measurement`

Measurement
timeSource
registerData notifyOK newTime setTime BackupData moduleCall

4.3.3 SENZOR

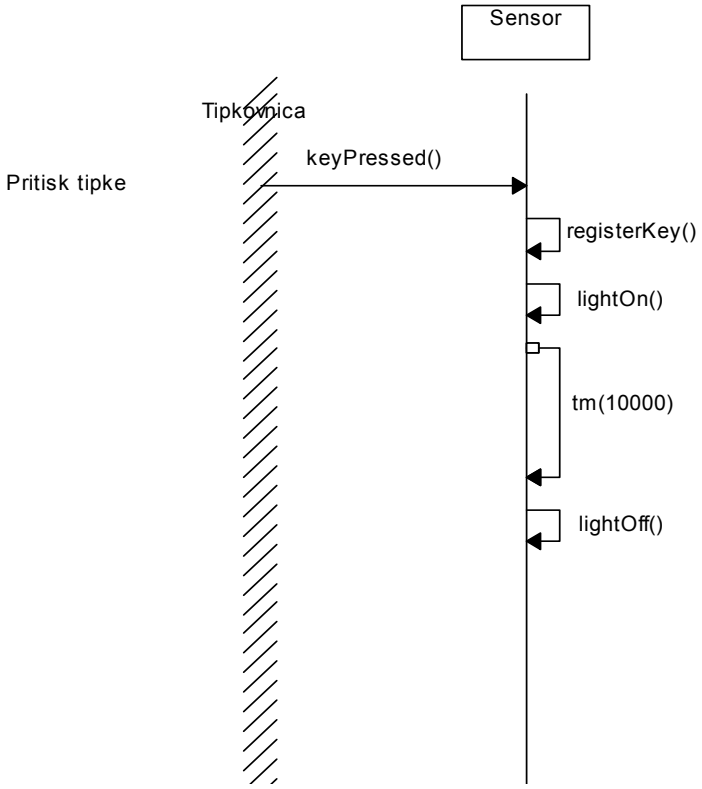
Naloga senzorja je nadzor nad vhodnim delom elektronskega števca z izjemo izpisa na zaslon. Vhodni del predstavljajo tipke, fizični moduli in krmiljenje osvetlitve zaslona. Senzor v grobem predstavlja neskončna zanka, ki prebira dogajanje na opisanih vhodih in se nanje ustrezno odzive.

Naloge senzorja so:

- ugotavljanje stanja tipkovnice,
- ugotavljanje stanja fizičnih modulov,
- krmiljenje osvetlitve zaslona.

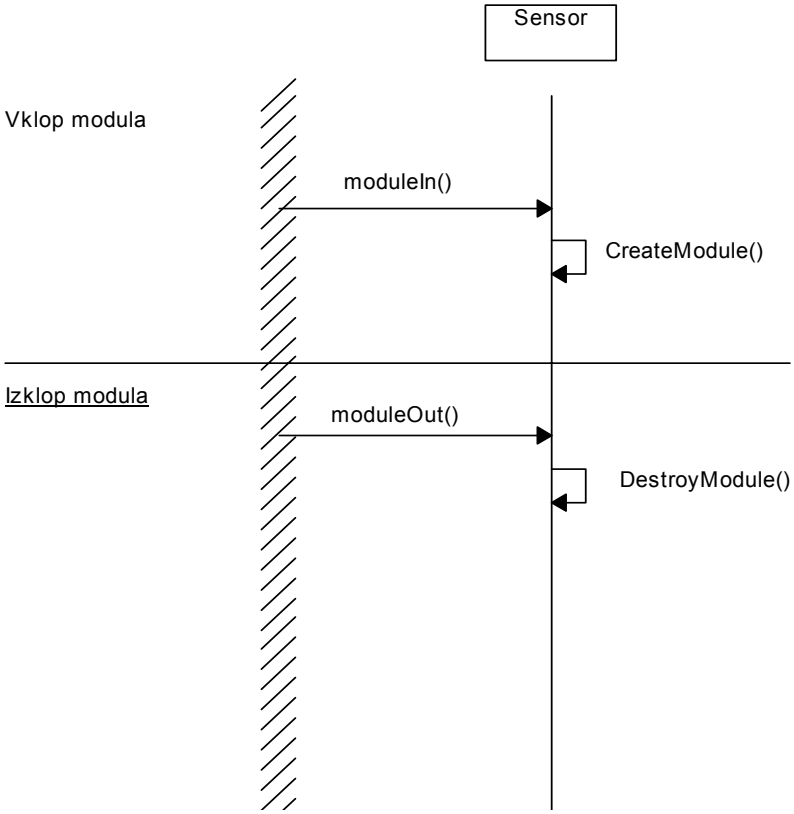
Scenariji:

Slika 44: Slika: Scenarij za pritisk tipke in osvetlitev zaslona



Senzor periodično bere stanja tipkovnice in s pomočjo operacije `registerKey` beleži pritiske katere od tipk. V primeru, da je tipka pritisnjena, mora senzor osvetliti zaslon `lightOn`. Po določenem času neaktivnosti tipkovnice, pa mora osvetlitev izključiti `lightOff`.

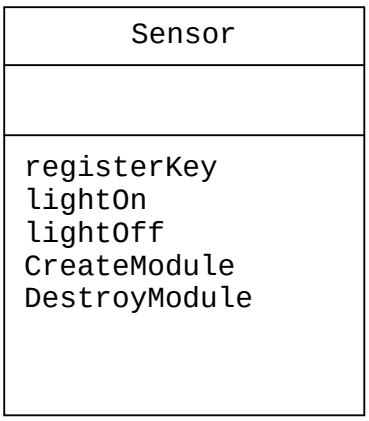
Slika 45: Scenarij za zaznavo fizičnih modulov



Princip delovanja fizičnih modulov je takšen, da mora programska rešitev sama zaznati fizično priključitev modula in se nanjo ustrezno odzvati. Ko uporabnik v izdelek vstavi fizični modul, se mora na novo programsko opremo ustrezno odzvati s pomočjo operacije `createModule`. Ko pa uporabnik fizični modul iztakne, je postopek obraten in sledi klic `destroyModule`.

Identificiramo nov objektni tip `Sensor` in analiziramo različne scenarije njegovega obnašanja, poskušamo ugotoviti katere operacije omogočijo obnašanje senzorja, ki smo si ga zamislili s podanimi nalogami. Na podlagi scenarijev sem identificiral nov objektni tip analize `Sensor`.

Slika 46: Objektni tip analize Sensor



4.3.4 KOMUNIKACIJA PO PROTOKOLU IEC1107

Za namen komunikacije imamo pripravljene tri različne komunikacijske vmesnike: vmesnik za komunikacijo po IEC 1107 protokolu, IEC 870 in po ETHERNET protokolu. Zaenkrat bo programska rešitev podpirala samo komunikacijski protokol IEC 1107, druga dva pa bosta realizirana kasneje. IEC 1107 standard je protokol za prenos podatkov namenjen avtomatskemu merjenju in nadzoru porabnikov električne energije. Je največkrat uporabljen komunikacijski protokol za prenos podatkov v industriji.

Namen same programske rešitve ni realizacija komunikacijskega protokola po ISO[48]-OSI modelu. Pri uporabniški rešitvi želimo uporabiti samo aplikacijski nivo realizacije komunikacijskega protokola IEC 1107 in z njim realizirati vse komunikacijske zahteve, ki nam jih določa zahtevnik za komunikacijo. Zahtevnik za komunikacijo pa je zajet v VDEW zahtevniku za elektronske industrijske števce.

Naloge komunikacijskega strežnika so:

- branje in vpis podatkov preko komunikacije,
- nalaganje programa preko komunikacije.

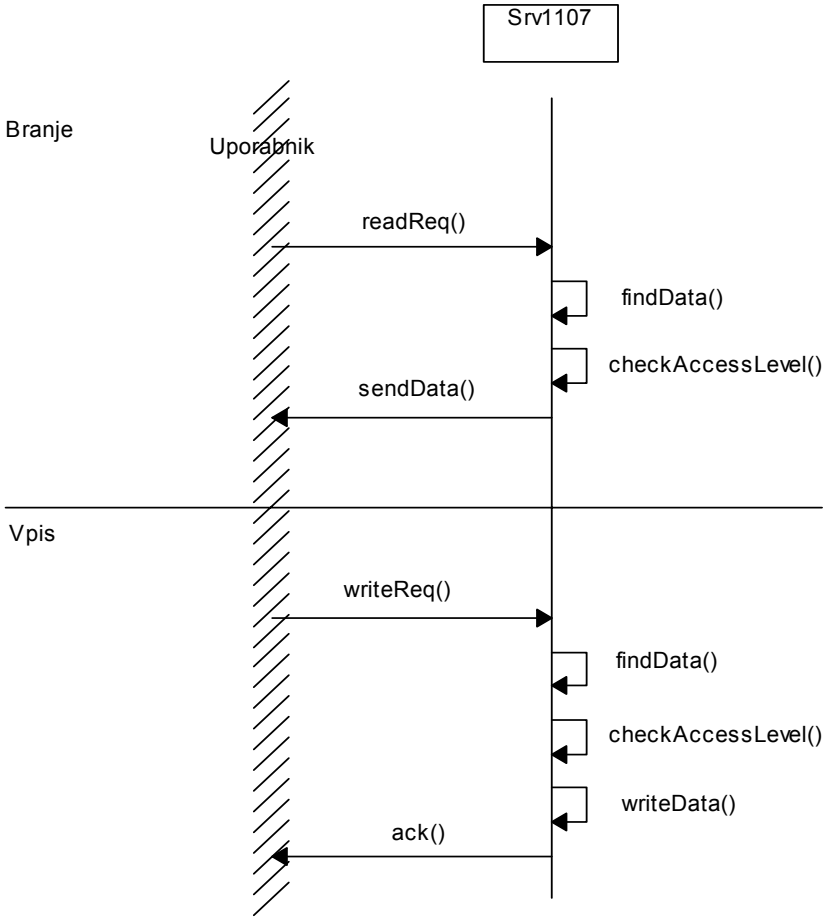
Scenariji:
Branje podatkov: Uporabnik pošlje zahtevo za branje podatka s pomočjo operacije `readReq`, ki jo komunikacijski strežnik obdela in preko `findData` poišče podatek v bazi podatkov. Komunikacijski strežnik v bazi podatkov preveri tudi nivo dostopa uporabnika `checkAccessLevel`. Če je podatek najden in nivo dostopa pravilen, dobi uporabnik podatek preko operacije `sendData`, v nasprotnem primeru pa dobi v odgovor `nok`.

Vpis podatkov: Uporabnik pošlje zahtevo za vpis podatka s pomočjo operacije `writeReq`, katero strežnik obdela na podoben način kot pri branju. V primeru vpisa podatka, ki obstaja in pravilnega nivoja dostopa, je podatek vpisan `writeData`, uporabniku pa vrnjen `ack`. V nasprotnem primeru dobi uporabnik v odgovor `nok` (glej Sliko 48, na str. 57).

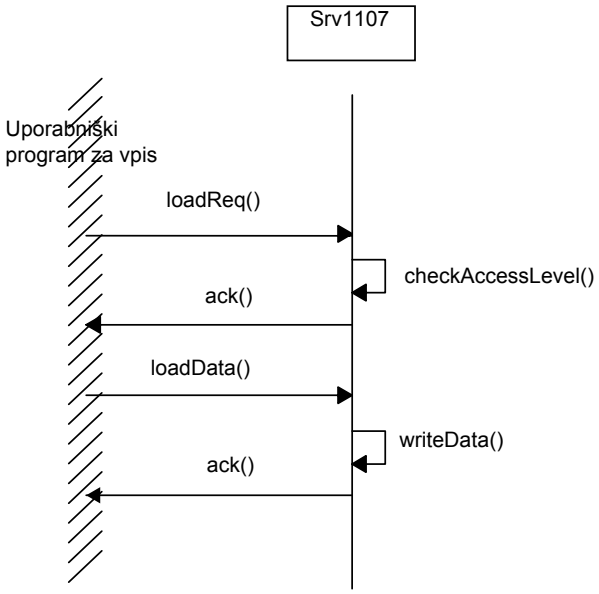
Zahtevo za nalaganje novega programa komunikacijskemu strežniku pošlje uporabniški program z operacijo `loadReq`. Podobno kot pri prejšnjih primerih preverimo nivo dostopa uporabnika. Če je nivo dostopa dovolj visok, je v odgovor poslan `ack`. V tem primeru uporabniški program lahko nadaljuje z nalaganjem programa z operacijo `loadProg`. Komunikacijski strežnik z `ack` obvesti uporabniški program, če je bil prenos programa končan (glej Sliko 49, na str. 57).

[48] ang. international standard organization

Slika 47: Slika: Scenarija za vpis in branje s komunikacijo

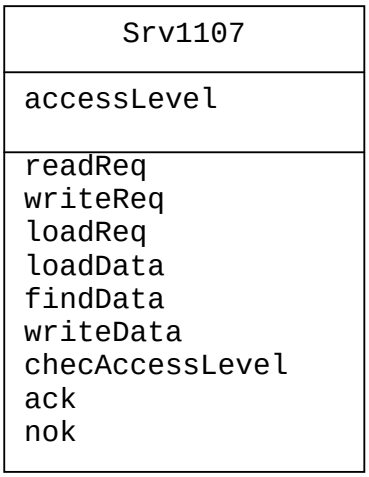


Slika 48: Scenarij za vpis programa preko komunikacije



Na podlagi scenarijev identificiramo nov objektni tip analize `Srv1107`:

Slika 49: Slika: Objektni tip analize Srv1107



4.3.5 OBJEKTNI DIAGRAM PROGRAMSKE REŠITVE

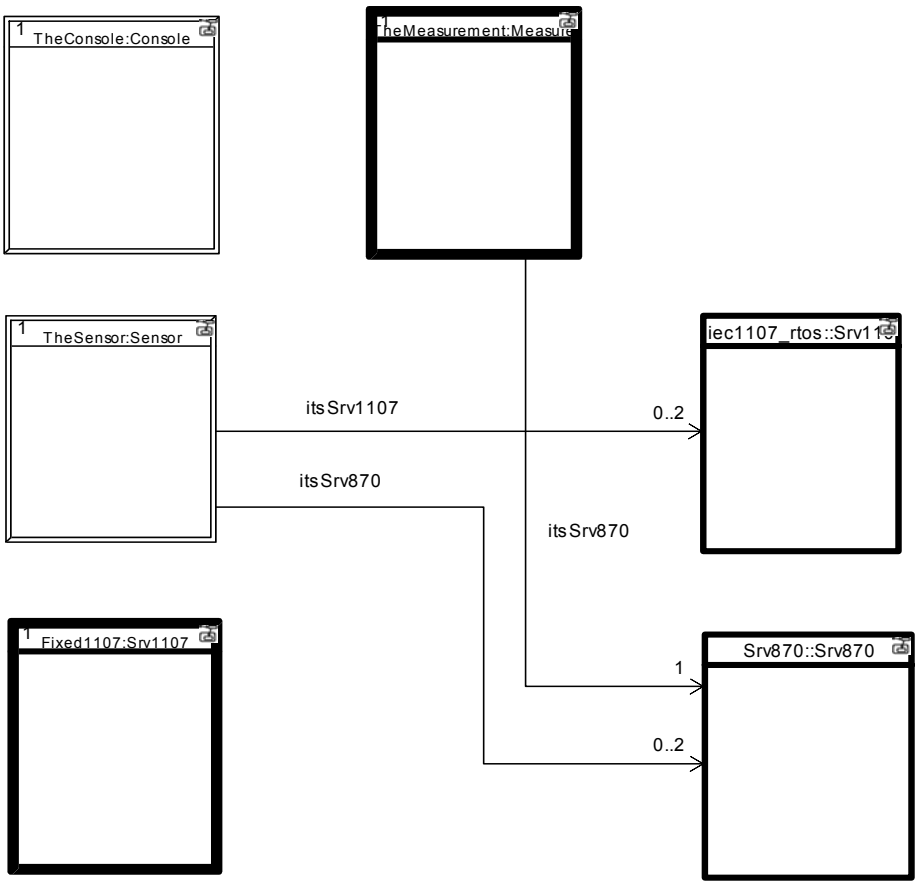
Z objektnim diagramom programske rešitve povežemo vloge objektnih tipov v končni programski rešitvi. V objektnem diagramu so prikazane instance objektnih tipov, ki smo jih identificirali med postopkom analize. Objektni tip `Console` je namenjen prikazu na zaslon, `Measurement` registraciji in nadzoru delovanja sistema, `Sensor` branju tipkovnice in priklopa modulov, `Srv1107` in `Srv870` pa sta komunikacijska strežnika, ki skrbita za komunikacijo. Posamezni objekti so večinoma neodvisni drug od drugega.
Medsebojno povezavo imajo:

- objektni tip `Sensor` , ki ima asociacijo z komunikacijama `Srv1107` in `Srv870,`
- objektni tip `Measurement`, ki ima asociacijo z komunikacijo `Srv870`.

4.4 NAČRTOVANJE

V aktivnosti načrtovanja je potrebno identificirane objektne tipe analize razviti do te mere, da omogočimo njihovo implementacijo. Objektni tip analize največkrat za svoj opis potrebuje dva ali več objektna tipa načrtovanja. Identificirano število objektnih tipov je v našem primeru zadosti veliko, saj naša programska rešitev ne potrebuje še večjega strukturiranja objektnih tipov.

Slika 50: Arhitektura uporabniške rešitve prikazana z njenim objektnim diagramom.



4.4.1 KONZOLA

V aktivnosti analize objektni tip Console še nekoliko bolj razdelamo in s tem omogočimo kasnejšo implementacijo:

1. Sistem ima štiri načine prikazovanja podatkov: avtomatski, ročni na poziv, ročni na poziv z večjo natančnostjo in nastavitveni.
2. Prikaz želenega formata podatka, format vpliva na maksimalno in minimalno vrednost prikazanega podatka.
3. Indikacija prisotnosti faz in smer merjene energije na zaslonu s pomočjo segmentov.
4. Indikacija statusov in sistemskih vrednosti na zaslonu s pomočjo kazalcev.
5. Omogočanje branje rekorderja meritev (Rekorder meritev shranjuje porabo energije in moči v časovnih intervalih).
6. Zaključitev obračunske periode (Obračunska perioda je časovni interval znotraj katerega je zabeležena poraba energije in moči.)
7. Parameterska tipka omogoča inicializacijo sistema ob prvem zagonu.

Ime:

`Console`

Odgovornosti:

Izpis na zaslon. Omogočanje upravljanja z napravo preko zaslona in tipk.

Operacije:

`ChangeValue`
`Forcereset`
`Init`
`LCDTest`
`NextItem`
`NextSetItem`
`NextManualMenuItem`
`NextSetMenuItem`
`Refresh`
`RefreshFlagState`
`SetValue`
`StartAuto`
`StartBillingReset`
`StartLCDTest`
`StartManual`
`StartManualMenu`
`StartSet`
`StartSetMenu`
`StartTest`
`SartTestScroll`
`TimeTrigger`
`Transform`

Objektnega tipa načrtovanja `Console` nazorno opisuje diagram prehajanja stanj, saj je v diagramu popolnoma zajeto obnašanje končne implementacije.

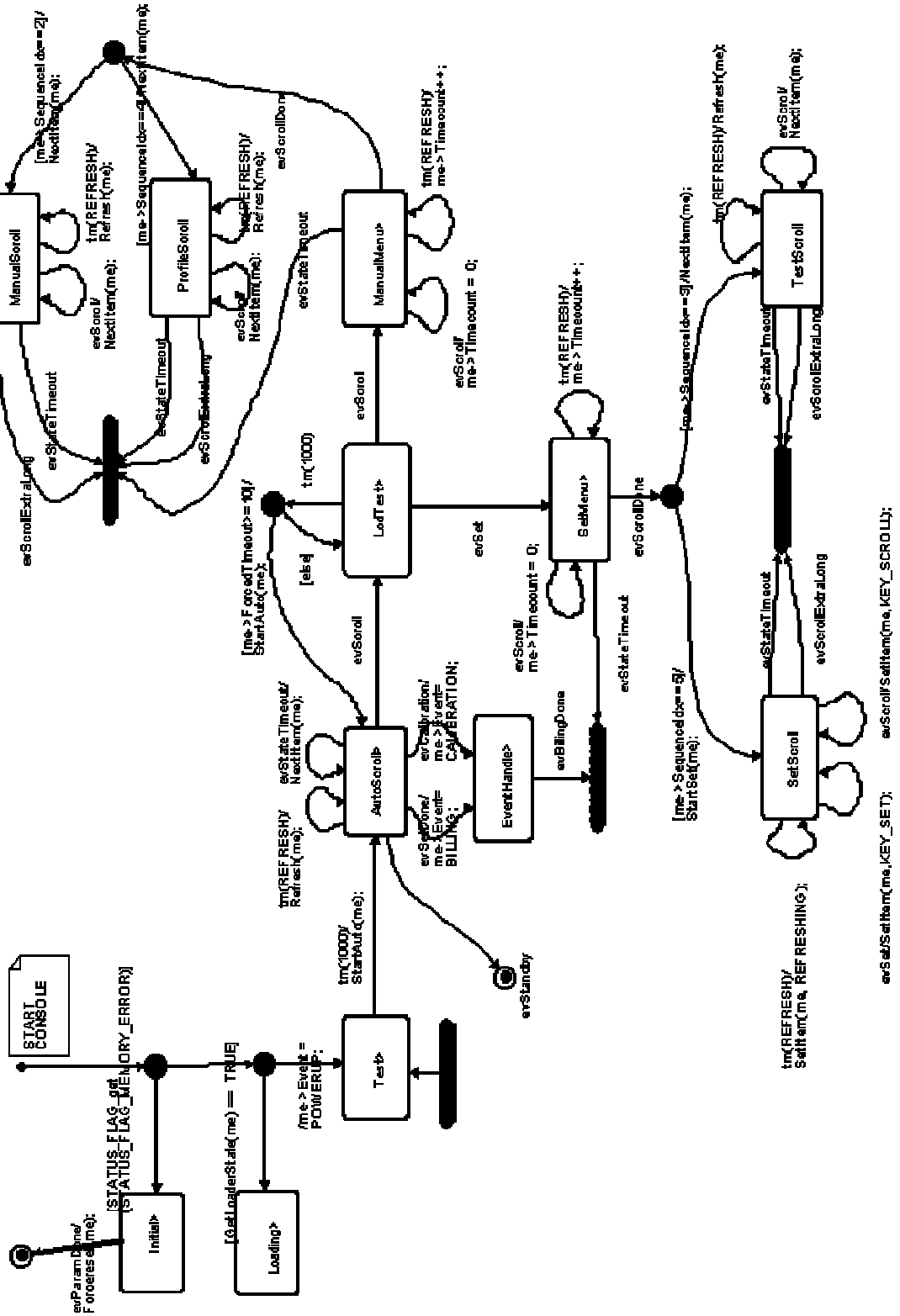
4.4.2 SENZOR

Tako kot pri objektnem tipu `Console` nekoliko strukturiramo funkcionalnosti, ki smo jih identificirali že pri analizi, in objektnemu tipu omogočimo kasnejšo implementacijo.

Stanje tipkovnice ugotavljamo s pomočjo klica `Pconsole` programske platforme, ki nam pošilja dogodke v primeru pritisnjene in spuščene tipke. Števec električne energije ima možnost naslednjih fizičnih modulov**:**

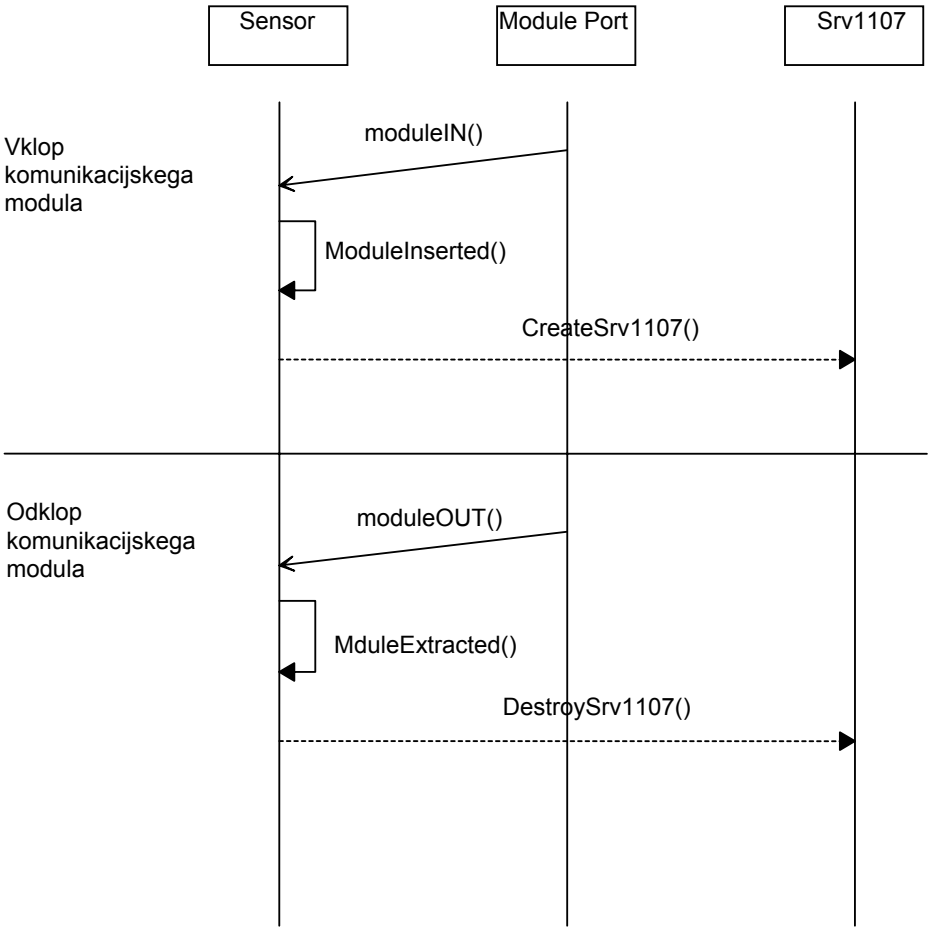
- vhodno/izhodni modul.
- komunikacijski modul.

Slika 51: Diagram prehajanja stanj za konzolo



Fizični vhodno/izhodni moduli naši napravi omogočijo dodatne funkcionalnosti, saj lahko naprava preko vhodov sprejema dogodke. Preko vhodov je lahko krmiljena aktivna tarifa, poleg tega pa vhodi omogočajo merjenje energije. Za detekcijo spremembe vhodov skrbi operacija `Scann`, ki potem pošlje dobljene podatke iz programskega modula `INPUTS` v registracijski del. Za izhode je odgovoren programski modul `OUTPUTS`.

Slika 52: Slika: Diagram sodelovanja za detekcijo komunikacijskih modulov



Ime:

`Sensor`

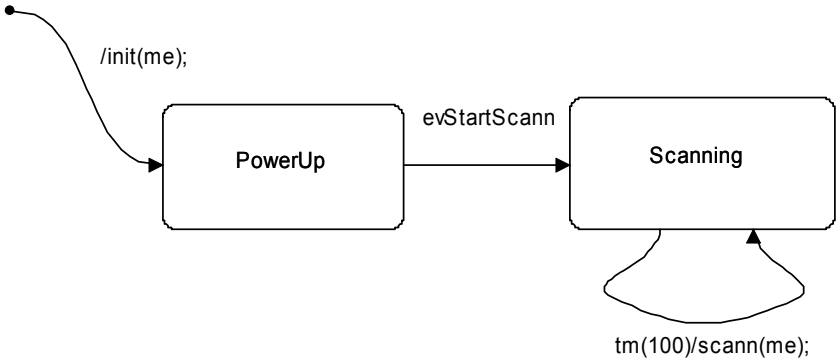
Odgovornosti:

Upravljanje s tipkovnico sistema in fizičnimi vhodi naprave. Detekcija komunikacijskih modulov.

Operacije:

`init`
`ModuleExtracted`
`ModuleInserted`
`scann`
`CreateSrv1107`
`CreateSrv870`

Slika 53: Diagram prehajanja stanj za senzor



4.4.3 REGISTRACIJSKI DEL

Registracijskemu delu dodamo nekatere funkcionalnosti, ki jih bomo kasneje potrebovali v implementaciji.

Naloge registracijskega dela:

- Registracija energije in moči.
- Registracija napetosti, toka, frekvence in kakovosti napetosti.
- Varovalka delovanja sistema.
- Nastavitev in sinhronizacija ure.
- Izvajanje obračuna merilne periode.
- Nastavitev tarife.
- Registracija podatkov preko fizičnih vhodov.
- Krmiljenje bremen preko fizičnih izhodov.
- Varnostna kopija pomembnih podatkov.
- Periodični klic programskih modulov.

Ime:

`Measurement`

Odgovornosti:

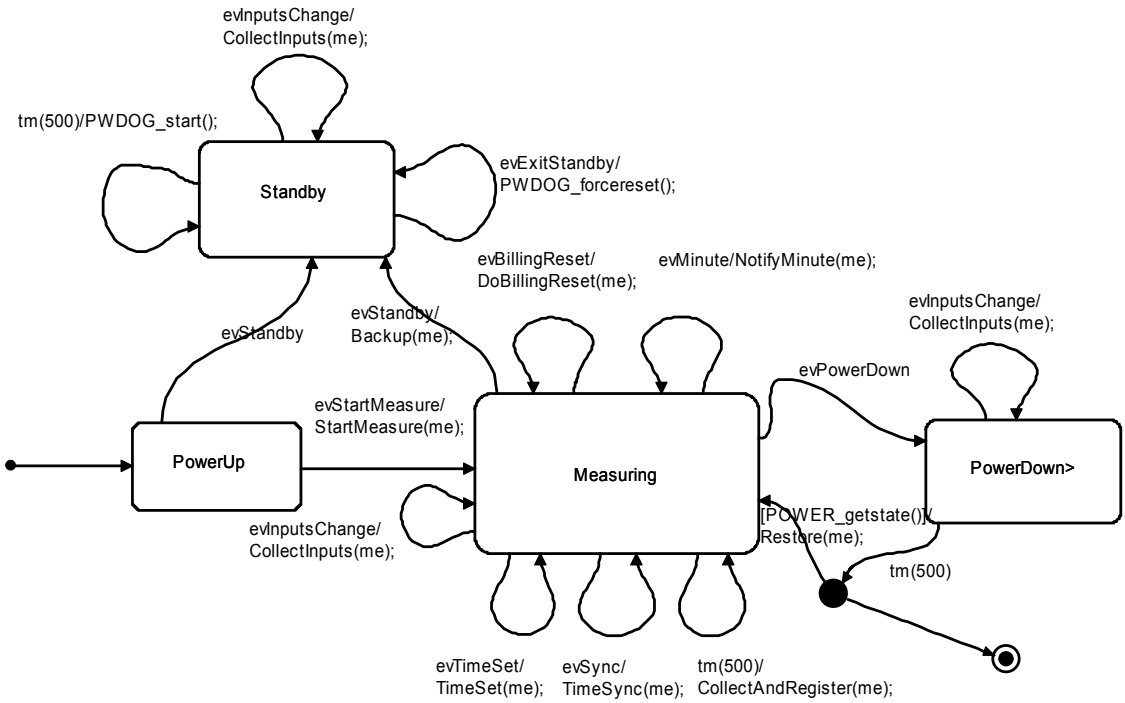
Registracija merilnih podatkov, upravljanje s sistemsko uro in nadzor delovanja sistema.

Operacije:

```
Backup
CollectAndRegister
CollectInputs
DoBillingReset
MeasPeriodeHandle
NotifyMinute
Restore
```

```
StartMeasure
TariffHandle
TimeSet
TimeSync
```

Slika 54: Diagram prehoda stanj za registracijski del



4.4.4 KOMUNIKACIJA PO PROTOKOLU IEC1107

Iz dodatnih zahtev ali potreb uporabnikov ugotovimo dodatne funkcionalnosti, ki jih mora imeti komunikacija.

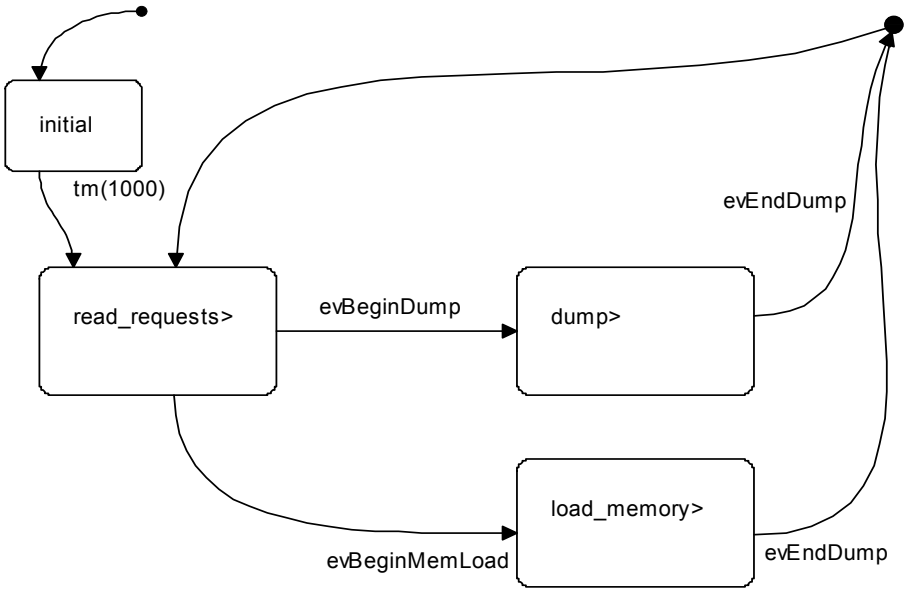
Naloge komunikacijskega strežnika:

- branje in vpis sistemskih parametrov,
- branje navideznih registrov števca,
- branje rekorderja meritev,
- branje rekorderja dogodkov,
- proženje obračunske periode preko komunikacije,
- nalaganje uporabniškega programa preko komunikacije.

Sistemskih parametri so podatki, s katerimi lahko krmilimo obnašanje sistema med samim delovanjem. Pomen sistemskih parametrov je še toliko večji, če lahko uporabnik na daljavo

krmili obnašanje sistema. Med sistemske parametre štejemo podatke, ki opisujejo format izpisa na zaslon in format pri komunikaciji, koledar dogodkov sistema in tiste, ki opisujejo obnašanje sistema. S pomočjo operacije `GetRequest`, ki predstavlja vhodno/izhodni medpomnilnik, obdelujemo zahteve podane komunikacijskemu strežniku. Sistemski parametri so pravzaprav neka tabela, v kateri so našteti vsi parametri, identifikacijske številke, njihove vrednosti in dostopnost. Branje in vpis sistemskih parametrov preko komunikacije je pogojeno z nivojem dostopa uporabnika. Gre za nekaj nivojev dostopa, ki so dosegljivi le z gesli naprave. Samo branje ali vpisovanje sistemskih parametrov je enostavno. S pomočjo posebnega programskega orodja Meterview (Meterview je orodje namenjeno parametriranju elektronskih števcev) vpišemo ustrezno geslo in identifikacijsko številko podatka, naprava kot odgovor pošlje zahtevan sistemski parameter. Za vpis pa je potrebno dodati še vrednost podatka, ki ga želimo vpisati.

Slika 55: Diagram prehajanja stanj za komunikacijski strežnik Srv1107



Ime:

`Srv1107`

Odgovornosti:

Komunikacija po komunikacijskem protokolu IEC 1107 s podporo VDEW zahtevnika.

Operacije:

`Cleanup`
`Dump`
`DumpLogBook`
`DumpMemory`
`DumpProfile`

```
DumpReadout
GetRequest
Init
LoadMemory
```

4.5 IMPLEMENTACIJA

Glavna aktivnost implementacije je pisanje programske kode, kljub temu pa je v implementaciji mogoče uporabiti nekaj elementov UML. Aktivnost implementacije je transformacija UML modelov načrtovanja v izvedljivo kodo. Za predstavitev delovanja izvedljive kode je potrebno skonstruirati model implementacije, vendar je ta model le stranska aktivnost implementacije. Model implementacije pomeni razporeditev objektnih tipov načrtovanja v komponente, kako to storimo pa je v veliki meri odvisno od uporabljenega programskega jezika (Arlow, Neustadt, 2002, str. 350).

Model implementacije je sestavljen iz:

- diagrama komponent (modeliranje odvisnosti med komponentami programske kode),
- diagrama porazdelitve (modeliranje ciljnega sistema, kjer bo programska rešitev uporabljena).

Vhodi v generator programske kode so model komponent, strukturni model, model obnašanja in nastavitve generatorja. Izhodi iz generatorja pa so izvedbene datoteke[49], opisne datoteke[50] in sestavne datoteke[51]. Prevajalnik in povezovalnik prevedeta te datoteke in jih povežeta skupaj z ostalimi datotekami, knjižnicami in ogrodjem za izvajanje objektov v izvršljivo programsko kodo (glej sliko 56).

Implementacija uporabniške rešitve je dokaj kompleksna in na tem mestu nepomembna, saj se neposredno ne dotika procesa razvoja uporabniške rešitve. Sama implementacija uporabniške rešitve oziroma aplikacije sistema obsega preko 10000 vrstic programske kode, celoten sistem z programskimi moduli in programskimi vmesniki sistemske platforma pa še veliko več.

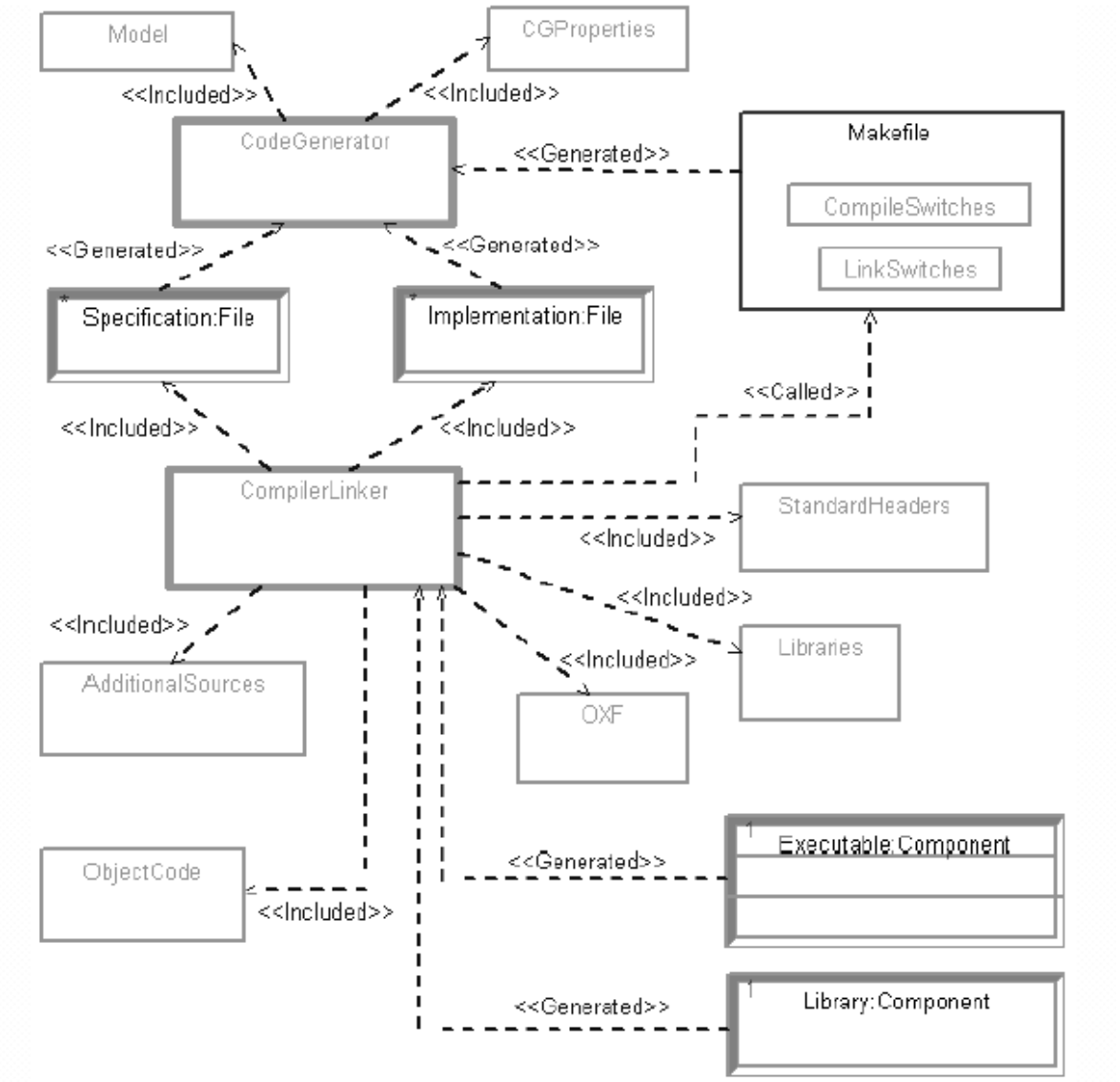
4.5.1 ARHITEKTURA PROGRAMSKE REŠITVE

Vsaka programska rešitev mora imeti čim enostavnejšo, a kljub temu zanesljivo organizacijsko strukturo. Naslednji korak v razvojnem procesu je izgradnja arhitekturnega ogrodja programske rešitve za implementacijo funkcionalnosti sistema. Za izvajanje svojih funkcionalnosti

[49] ang. Implementation files
[50] ang. Specification files
[51] ang. Make files

programska rešitev uporablja programske module in programske vmesnike[52] sistemske platforme (glej sliko 57).

Slika 56: Generator programske kode



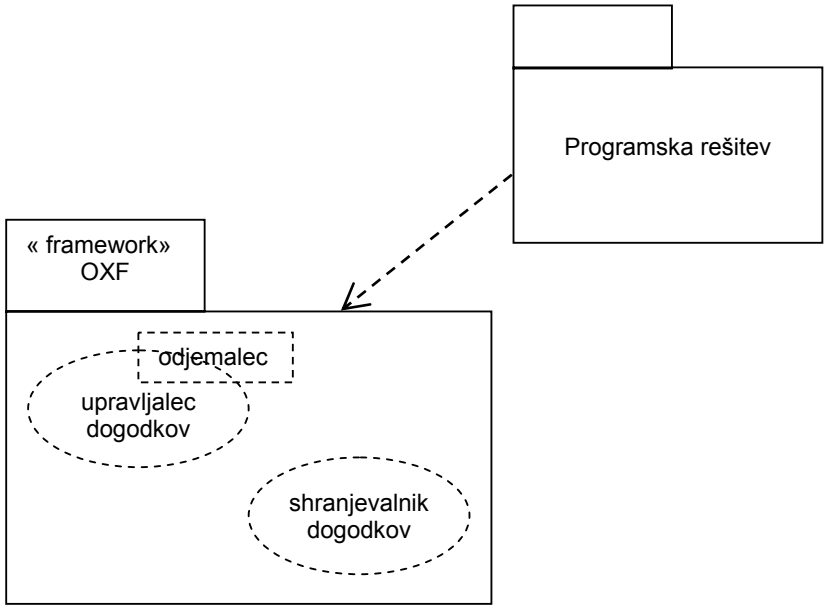
Osnovno načelo programske rešitve je naslednje:

- Programska rešitev lahko dostopa do poljubnih programskih modulov in programskih vmesnikov sistemske platforme.
- Programski modul je napisan zaradi dodatne funkcionalnosti uporabniške rešitve, ima dostop do programskih vmesnikov sistemske platforme.

[52] ang. application programming interface (API)

- Programski vmesniki sistemske platforme so preslikave funkcionalnosti fizične platforme v obliko, da so uporabne za uporabniško rešitev in programske module.

Slika 57: Programsko ogrodje



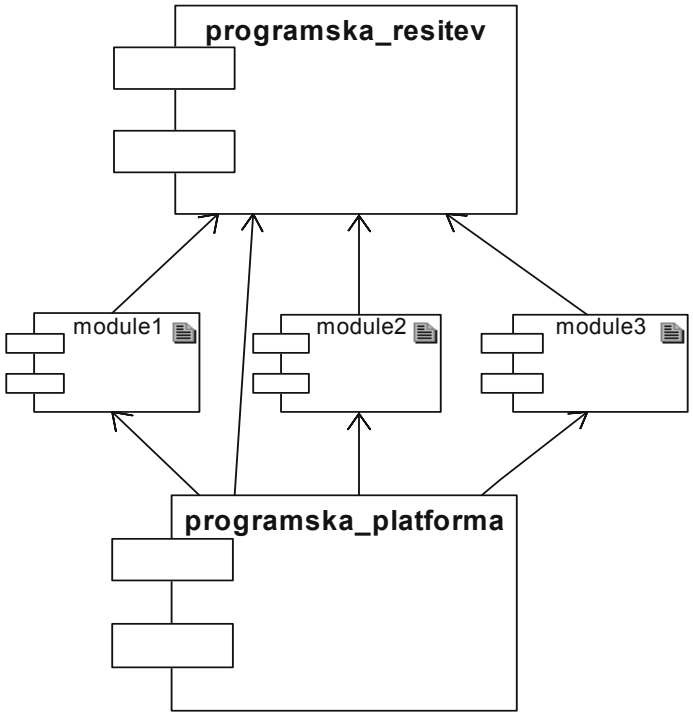
Razdelitev razvoja programske opreme na sistemsko platformo in aplikacijo omogoča izvajanje nespremenjene aplikacije na več platformah in hkrati uporabo platforme na več aplikacijah. Pomembna platforma je tudi PC platforma, ki predstavlja tako razvojno kot tudi preizkusno platformo.

Sistemska platforma predstavlja osnovni nabor servisov, ki so povezani s strojno opremo. Vsi servisi platforme so v obliki standardiziranih programskih vmesnikov. Prav tako so del platforme servisi RTOS[53], ki omogočajo razvoj platforme v obliki procesov.

Programska rešitev predstavlja nabor procesov, ki se izvajajo na platformi. Procesi s platformo komunicirajo preko servisov platforme, ki so enaki za različne platforme. Programska rešitev je neodvisna od platforme in se lahko izvaja na različnih platformah v primeru, da so servisi platforme enaki. Procesi programske rešitve med seboj komunicirajo z medprocesno komunikacijo, ki je del servisov RTOS operacijskega sistema.

[53] ang. real time operating system

Slika 58: Diagram komponent programske rešitve



4.5.2 PROGRAMSKI MODULI

Aplikacija ali programska rešitev števca je zgrajena iz procesov, ki se izvajajo na aplikacijskem nivoju preko več procesne komunikacije. Procesi uporabljajo module, ki so zaključene celote in jih lahko prenašamo med različnimi aplikacijami. Moduli nudijo aplikacijskim procesom na razpolago svoje servise oziroma operacije, katere le-ti lahko uporabljajo. Vsak modul mora imeti definirane svoje konfiguracije, parametre in atribute.

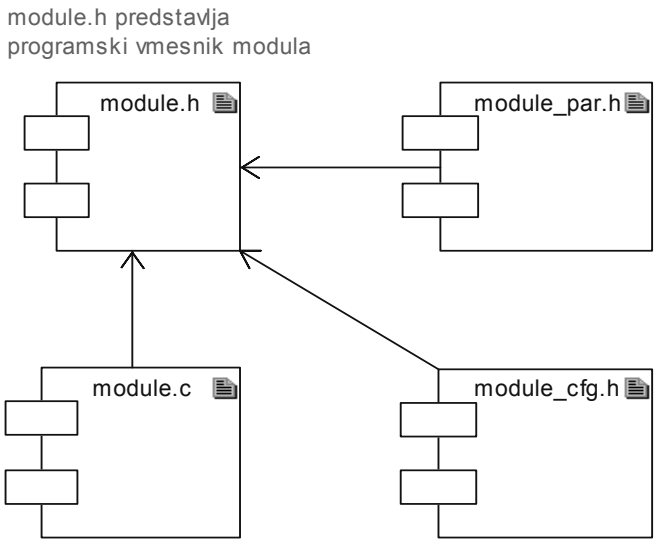
Osnovna zahteva za načrtovanje naprave je velika fleksibilnost pri prilagajanju različnim zahtevam. Da bi čim bolj ustregli tem zahtevam, morajo biti tudi sestavni deli te naprave oziroma programski moduli načrtovani tako, da jih lahko prilagajamo različnim zahtevam.

1. Konfiguracijski parametri modula so tisti, ki jih v času izvajanja programske rešitve ni mogoče spreminjati, predstavljajo grobo nastavitev sistema glede na zahteve. S konfiguracijo je mogoče spreminjati velikost pomnilnikov, izbirati je mogoče različne vrste prikaza, določanje vhodnega in izhodnega dela naprave itd. Največja prednost je, da ima proizvajalec naprave možnost v najkrajšem času prilagoditi izdelek konkretni aplikaciji.

2. Parametri modula so tisti, ki jih je v času izvajanja programske rešitve možno spremeniti. Njihova dostopnost je zaščitena s pomočjo modula, ki skrbi za hranjenje modulov (OBIS

map modul). Spreminjanje parametrov imenujemo parametriranje. Parametriranje imenujemo fino nastavitev programske rešitve, ki je namenjeno tudi za uporabnika.

Slika 59: Moduli programske rešitve



5 UGOTOVITVE PRI UPORABI POENOTENEGA JEZIKA ZA MODELIRANJE

Za razvoj programskih rešitev za vgrajene[54] računalniške sisteme, med katere sodi tudi elektronski števec, se navadno ne uporabljajo posebne metodologije za razvoj programskih rešitev. Proces razvoja programske rešitve za takšne sisteme največkrat ni najbolj časovno optimiziran, saj sama iteracija od spremembe zahtev do dejanske implementacije traja preveč časa. Zaradi pospešitve razvojnega cikla je v našem podjetju nastala potreba po uporabi UML orodja za načrtovanje in implementacijo programske rešitve. UML orodje je bilo namenjeno implementaciji sami in generaciji programske kode, zato se mi je porodila zamisel o uporabi UML za celotni razvojni cikel programske rešitve za vgrajeni računalniški sistem. V razvojnem procesu sem poskušal slediti smernicam metodologije poenotenega procesa.

5.1 KLASIČNI RAZVOJ PROGRAMSKE REŠITVE ZA VGRAJENE RAČUNALNIŠKE SISTEME

V splošnem je zelo težko govoriti, kam lahko uvrstimo arhitekturni pristop razvoja vgrajenih računalniških sistemov v okviru našega podjetja. Glede na moje izkušnje in delo na dosedanjih projektih lahko trdim, da je ta pristop še najbliže metodologiji življenjskega cikla. Po mojem mnenju tak razvojni proces pred nekaj leti sploh ni bil tako napačen, programsko rešitev je bilo potrebno razviti in med življenjskim ciklom ni bilo potrebno ničesar spreminjati ali nadgrajevati. Potrebno se je zavedati, da je potek razvojnega procesa zelo odvisen od ciljnega vgrajenega računalniškega sistema in tega, kateri prevajalnik programske kode uporabljamo. Še pred petimi leti so bili zmogljivejši mikroprocesorji zelo dragi in so za vgradnjo v takšne izdelke, kot so elektronski števci, uporabljali le cenejše različice. Za razvoj programske rešitve za takšne sisteme ni bilo potrebno večjega števila ljudi, dovolj sta bila že dva ali trije razvijalci. Časovni razvoj takšne programske rešitve ni smel presegati treh let. Takšna programska rešitev je bila v celoti napisana v C jeziku ali pa celo zbirnem jeziku in je kljub manjšemu številu funkcionalnosti presegala 50 kbytov programske kode. Že iz tega lahko vidimo pravilnost prejšnje trditve, da je bilo vnašanje novih funkcionalnosti v takšno programsko rešitev silno zapleteno.

[54] ang. embedded computer systems

5.2 RAZVOJ PROGRAMSKE REŠITVE ZA VGRAJENE RAČUNALNIŠKE SISTEME S POMOČJO UML

Nekdanji procesi razvoja programskih rešitev so imeli poudarek na realizaciji delujoče programske rešitve, problem pri takšnem arhitekturnem pristopu je nastal pri vzdrževanju programske rešitve. Nastala je potrebe po izdelavi programske rešitve, ki bi bila kos spreminjajočim zahtevam, razvoj takšne programske rešitve omogočajo objektno orientirane metode. Držati se je potrebno dejstva, da je razvoj računalniškega sistema podoben nekemu procesu nenehnih sprememb, zato se je potrebno držati dveh načel:

- Potrebno je razviti programsko rešitev, ki bo imela zadosti robustno arhitekturo.
- Proces razvoja mora potekati iterativno in inkrementalno. Iterativnost pomeni razbitje projekta na več manjših projektov, to smo dosegli z modularno zgradbo programske rešitve. Inkrementalnost pa pomeni postopno dopolnjevanje programske rešitve do končne funkcionalnosti.

Zaradi potrebe po robustni arhitekturi smo posegli po Rhapsody CASE orodju, kateri podpira UML poenoten jezik za modeliranje. S pomočjo modelov sem uspel narediti zadosti robustno arhitekturo, ki omogoča preprosto dodajanje novih funkcionalnosti. Z UML sem naredil splošni načrt rešitve, ki je sestavljen iz objektnega diagrama, ki opisuje arhitekturo programske rešitve, in diagramov prehajanja stanj, ki opisujejo obnašanje objektov. Iz tega modela načrtovanja izhaja celotna programska rešitev.

Proces razvoja mora potekati iterativno, kar smo v našem primeru naredili tako, da smo celotni vgrajeni računalniški sistem razbili na več delov:

- programska rešitev razvita z UML, ki je osrednji samega vgrajenega sistema,
- programski moduli služijo za razširitev funkcionalnosti programske rešitve,
- programska platforma, ki je povezava med programsko rešitvijo in fizičnim delom vgrajenega sistema.

Programsko rešitev sem razvijal sam, za programske module in programsko platformo pa so skrbeli drugi člani projektnega tima.

Objektna orientiranost ponuja številne nove možnosti in pristope, ki omogočijo, da že med samim inkrementalnim razvojem ali razvojem sorodne programske rešitve uporabimo narejene programske module. To lahko povzroči, da lahko velik del vgrajenega računalniškega sistema naredimo iz že obstoječih programskih delov. Ponovna uporaba računalniške kode ali celo celotnih programskih modulov pomeni velike stroškovne prihranke in, kar je še veliko pomembnejše, časovne prihranke pri razvoju sorodnih programskih rešitev.

5.2.1 RAZŠIRITEV UPORABE UML

V okviru projekta smo UML uporabljali v fazi načrtovanja v zelo omejenem obsegu in v fazi implementacije. Naša naloga je bila le izpopolniti razredni diagram, diagrame prehajanja stanj in diagram komponent do te mere, da so bili sposobni generacije programske kode. Model smo uporabili za implementacijo programske rešitve in operacijam razredov dodali funkcionalnosti s pomočjo izvorne kode v programskem jeziku ANSI C. V magistrski nalogi pa sem se odločil, da uporabo UML poenotenega jezika razširim na celotni razvojni cikel programske rešitve, na fazo zbiranja zahtev, analize, načrtovanja in implementacije, ki govori tudi o vzdrževanju programske rešitve.

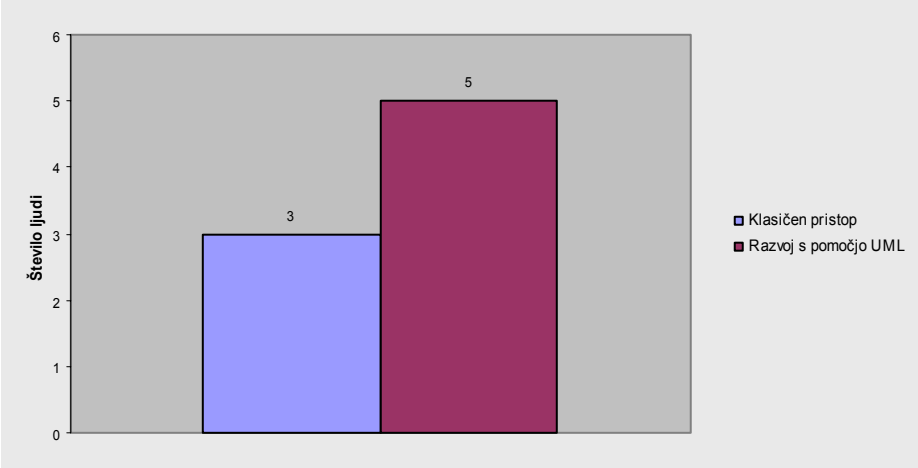
Celotni proces razvoja programske rešitve s pomočjo UML:

- *Faza zbiranja zahtev* se osredotoči na oblikovanje poslovne vizije, zbiranje potreb interesentov in iskanje akterjev in primerov uporabe.
- Glavna naloga *faze analize* je analiza primerov uporabe. Iz analize primerov uporabe zgradimo razrede analize, nato pa je potrebno primere uporabe realizirati. Razredi analize predstavljajo jasno sliko poslovne domene in vsebujejo samo najpomembnejše atribute in operacije opisane z visoko mero abstrakcije. Realizacija primerov uporabe prikazuje kako medsebojno delovanje instanc razredov analize omogoča funkcionalnosti sistema, to delovanje opišemo z diagrami interakcij.
- V *fazi načrtovanja* je potrebno razrede analize dopolniti do te mere, da je kasneje mogoča njihova implementacija, rezultat aktivnosti pa so razredi načrtovanja. Razredi načrtovanja poleg informacij iz problemske domene vključujejo tudi informacije iz domene rešitev. Aktivnost realizacije primerov uporabe načrtovanja, podobno kot pri razredih, dopolni diagrame interakcij z elementi implementacije. Za vsak razred zgradimo natanko en diagram prehajanja stanj, ki natanko opisuje njegovo obnašanje.
- Kot zadnja *sledi faza implementacije*, katere glavna aktivnost je generacija programske kode, oziroma pretvorba modela načrtovanja v kodo. Pri generaciji kode neposredno iz modela je pomembno zgraditi diagram komponent, ki modelira povezave med programskimi komponentami sistema. Za vzdrževanje programske rešitve je do neke mere pomemben diagram porazdelitve, saj modelira vključitev končnega sistema v realno okolje.

5.3 PRIMERJAVA OBEH PRISTOPOV

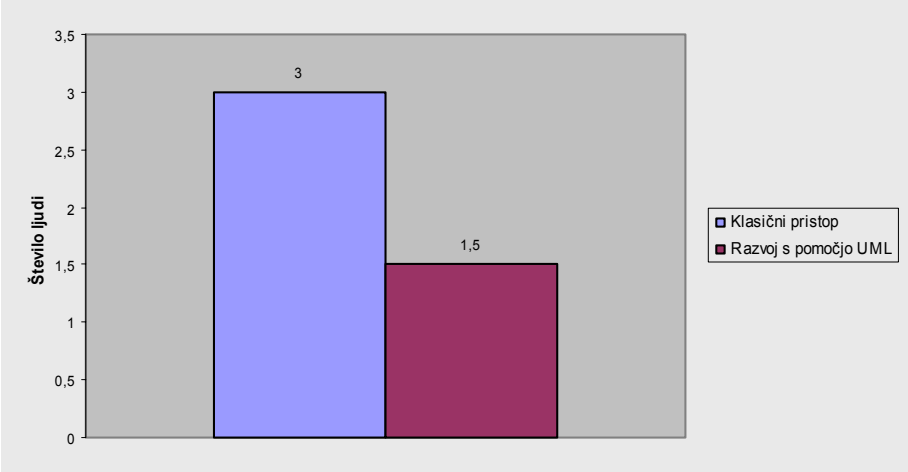
Na sliki 60 (glej naslednjo stran) vidimo primerjavo števila ljudi zaposlenih pri razvoju vgrajenega računalniškega sistema, katero je nekoliko večje pri razvoju s pomočjo UML. Več ljudi potrebujemo predvsem zaradi razvoja programskih modulov, vendar pa je potrebno vedeti, da bodo ti programski moduli lahko ponovno uporabljeni tudi pri sorodnih programskih rešitvah.

Slika 60: Število ljudi zaposlenih pri razvoju vgrajenega računalniškega sistema



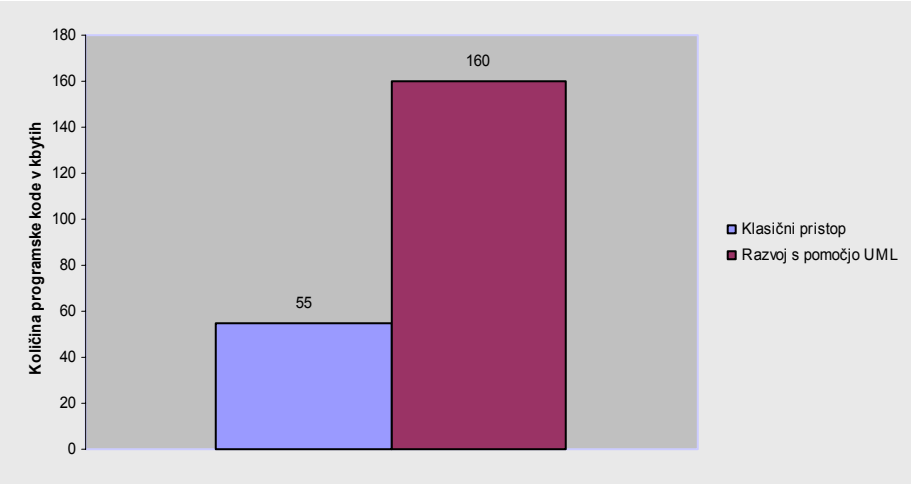
Slika 61 prikazuje trajanje razvoja vgrajenega računalniškega sistema iz katere je razvidno, da novejši pristop omogoča veliko hitrejši razvoj do prvega prototipa.

Slika 61: Trajanje razvoja vgrajenega računalniškega sistema



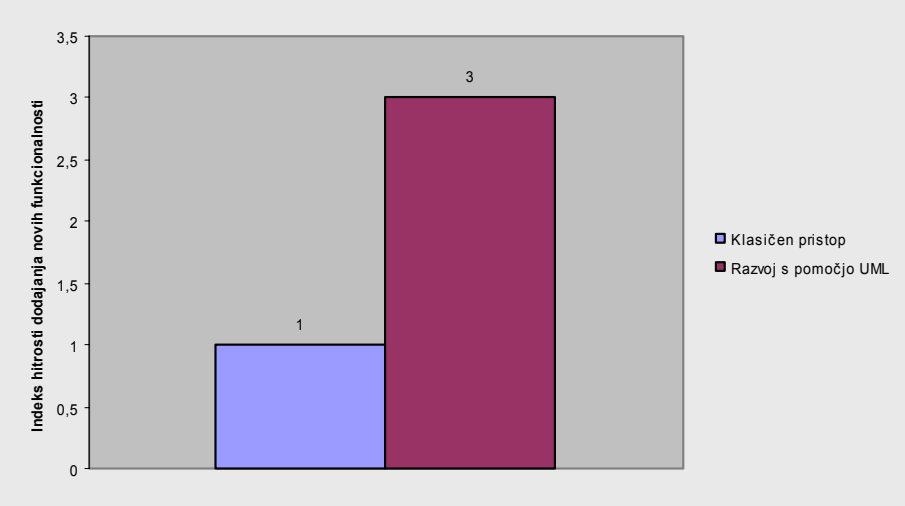
Slika 62 (glej naslednjo stran) prikazuje količino programske kode, ki jo prinese posamezni pristop. Podatke o količini programske kode ne moremo jemati kot povsem realne, kajti potrebno je vedeti, da ima programska rešitev s pomočjo klasičnega pristopa dosti manjšo funkcionalnost, kot rešitev razvita s pomočjo UML.

Slika 62: Količina programske kode vgrajenega računalniškega sistema



Podatki o indeksu hitrosti dodajanja novih funkcionalnosti, ki jih prikazuje slika 63, so po mojem mnenju najpomembnejši. Jasno je razvidno, da ima programska rešitev razvita s pomočjo UML veliko boljšo arhitekturo in zato je razširjanje funkcionalnosti programske rešitve veliko hitrejše.

Slika 63: Indeks hitrosti dodajanja novih funkcionalnosti v vgrajeni računalniški sistem



5.4 SWOT ANALIZA (ANALIZA PREDNOSTI, SLABOSTI, PRILOŽNOSTI IN NEVARNOSTI)

SWOT analiza je metoda, ki nam lahko služi kot vodilo pri strateškem planiranju ali pa le za podporo odločanju. SWOT analizo uporabljajo mnogi, od svetovalcev do planerjev po vsem

svetu, saj omogoča zelo dobro oceno prednosti in slabosti podjetja, glede na priložnosti in nevarnosti okolja (Mintzberg et. al, 1998, str. 24).

Pomen kratice SWOT razlagajo štirje angleške besede:

Strengths	Prednosti
Weaknesses	Slabosti
Opportunities	Priložnosti
Threats	Nevarnosti

Informacije za SWOT analizo zberemo v tabelo na naslednji način:

	Pozitivne	Negativne
Notranji vidik	Prednosti	Slabosti
Zunanji vidik	Priložnosti	Nevarnosti

Za iskanje najbolj primerne programske rešitve prav tako potrebujemo neko strategijo, saj je potrebno identificirati, kakšno programsko rešitev sploh potrebujemo in koliko sredstev moramo zagotoviti za njeno izvršitev (Ward, Peppard, 2003, str. 286). Nekatere programske rešitve so zastarele in nepotrebne, druge pa so lahko bistvenega pomena za podjetja, ki za svojo funkcionalnost potrebujejo določena finančna sredstva. Kot kriterij odločanja za nadaljnji razvoj in uporabo obstoječih programskih rešitev uporabimo SWOT analizo.

SWOT analizo ustreznosti programske rešitve lahko izvedemo iz dveh različnih vidikov (Ward, Peppard, 2003, str. 82):

- Notranji vidik zadeva trenutno strategijo razvoja programske rešitve, ter obravnava prednosti in slabosti projektne skupine, ki programsko rešitev razvija.
- Zunanji vidik vključuje analizo konkurence, tako da ugotovimo kje se nahaja podjetje glede na konkurenco in kakšne so lahko njegove prihodnje strategije za razvoj novih programskih rešitev.

S SWOT analizo bom primerjal klasičen način razvoja vgrajenih računalniških sistemov (glej tabelo 5) in razvoja vgrajenih računalniških sistemov s pomočjo UML (glej tabelo 6). Na tej točki se ne bom spuščal v podrobnosti obeh postopkov, saj sta oba postopka kompleksna in nepomembna za samo analizo. Z analizo bom poudaril značilnosti zunanjega in notranjega vidika obeh pristopov, ki imajo svojo težo pri uspešnosti programske rešitve. Zaključke SWOT analize bom poskušal razširiti na razvoj poljubnega računalniškega sistema s pomočjo UML.

Slika 64: SWOT analiza klasičnega pristopa razvoja programske rešitve za vgrajene računalniške sisteme

	Pozitivne	Negativne
Notranji vidik	• Manjša poraba programskih virov (cenejši mikroprocesor, manjša poraba pomnilnika). • Preklopi med programskimi procesi so neposredno vezani na programske prekinitve. • Količina programske kode. • Operacijski sistem je enostaven in zato ne prinaša dodatnih napak.	• Nepreglednost programske kode, saj vsak načrtovalec programske opreme rešuje probleme na svoj način. • Programska rešitve ne omogoča modularnosti in ponovne uporabe kode. • Načrtovanje in implementacijo takšne programske rešitve je zelo težko razdeliti na več razvijalcev, kar bistveno podaljša čas razvoja. • Dodajanje novih funkcionalnosti je zamudno in lahko hitro povzroči napačno delovanje starih funkcionalnosti.
Zunanji vidik	• Pri manjših programskih rešitvah pristop omogoča majhne stroške, kar pomeni pomembno konkurenčno prednost.	• Zaradi podaljšanega časa razvoja izdelka nas konkurenca lahko prehiti. • Težko je slediti spremembam potreb in željam kupcev. • Takšna programska rešitev je močno povezana s programsko platformo (strojno opremo), zato postajamo odvisni od dobaviteljev strojne opreme.

Iz analize vidimo, da je klasični pristop razvoja vgrajenih računalniških sistemov primernejši za enostavnejše programske rešitve, zahteve katerih se med življenjskim ciklom ne spreminjajo. Za klasični pristop je značilno, da programsko rešitev načrtujejo in implementirajo največ trije razvijalci. Na drugi strani razvoj programske rešitve s pomočjo UML omogoča številne prednosti, razvoj modularne programske rešitve do ponovne uporabe kode, kar lahko prinese hitrejšo diverzifikacijo izdelkov. V fazi načrtovanja in implementacije sodeluje več razvijalcev, kar ima za posledico krajši čas razvoja izdelka. Poleg tega UML lahko omogoča dodajanje funkcionalnosti, ki je lahko ključnega pomena pri hitrih spremembah zahtev kupcev. UML orodje Rhapsody omogoča generacijo kakovostne programske kode, katera temelji na programskem ogrodju UML orodja, ki prinese neodvisnost programske rešitve od programske platforme. Programsko rešitev je mogoče uporabljati in testirati tako na osebnem računalniku kot tudi na ciljnem mikroprocesorskem sistemu. Takšna prenosljivost prinese veliko hitrejše in enostavnejše izvajanje testnih procedur, ter ugotavljanje napak. Izredno uporabna je tudi

generacija poročil iz UML modela v tekstovni obliki, ki vsebujejo številne informacije o trenutni konfiguraciji programske rešitve.

Slika 65: : SWOT analiza razvoja programske rešitve za vgrajene računalniške sisteme s pomočjo UML

	Pozitivne	**Negativne**
Notranji vidik	• Največja prednost je vsekakor modularnost in omogočanje ponovne uporabe kode. • Načrtovanje in implementacijo programske rešitve je mogoče razdeliti med več neodvisnih razvijalcev, kar pospeši razvojni proces. • UML nam omogoča lažjo vpeljavo predpisanih postopkov in metodologij za razvoj programske rešitve. • Vpeljava novih funkcionalnosti je enostavna, saj robustna arhitektura onemogoča vplive med deli zgrajena sistema. • Operacijski sistem, ki podpira generirani UML model, je že preskušen in zato dobro deluje.	• Količina programske kode hitro narašča. • Cena UML orodja. • Število razvijalcev se poveča in hitro nastane problem komunikacije med razvijalci.
Zunanji vidik	• Modularnost programske kode omogoča prenosljivost kode med različnimi izdelki, tako da podjetje hitro izdela komplementaren proizvod. • Zaradi preproste vgradnje novih funkcionalnosti se lahko podjetje hitreje širi na nove trge in osvaja nove segmente trga. • Pri izbiri strojne opreme nismo vezani na samo enega proizvajalca, saj UML orodje omogoča, hitro menjavo programske platforme.	• UML orodij za razvoj vgrajenih računalniških sistemov je malo, tako da smo vezani na enega proizvajalca.

Ugotovitve, do katerih sem prišel pri SWOT analizi razvoja programske rešitve za vgrajene računalniške sisteme s pomočjo UML, lahko razširimo tudi na večje računalniške ali informacijske sisteme. Prednosti in priložnosti, ki jih UML prinaša z uvedbo modeliranja in

abstrakcijo problemske domene, se z večanjem kompleksnosti programske rešitve samo še povečujejo. Pri razvoju večjih programskih rešitev izginejo tudi nekatere slabosti in nevarnosti, na primer nismo več vezani na enega samega proizvajalca UML orodja, pa tudi cena UML orodja, glede na vrednost večjih sistemov, ne predstavlja več takšnega izdatka.

6 SKLEP

V okviru magistrskega dela sem prikazal proces razvoja vgrajenega računalniškega sistema s pomočjo poenotenega jezika za modeliranje UML. Proces razvoja sem prilagodil smernicam poenotenega procesa, ki predstavlja eno največkrat uporabljenih objektno orientiranih metodologij. Z UML orodjem zelo nazorno prikažemo proces razvoja programske rešitve v obliki diagramov. Proces razvoja tako postane lažje razumljiv tudi tistim, ki niso neposredno udeleženi v razvojnem procesu. Na eni strani je mogoče z diagrami primerov uporabe hitro modeliranje in opisovanje zahtev trga ali kupca. Na drugi strani pa z diagrami prehajanja stanj in objektnimi diagrami pregledno opisujemo končno implementacijo programske rešitve. Že pri modeliranju razmeroma preprostega vgrajenega računalniškega sistema, kot je števec električne energije, je bilo opaziti, da je potrebno diagrame ali pakete dobro strukturirati, če želimo ohraniti preglednost modeliranega problema. Če želimo z nekim diagramom zajeti preveč širok problem, lahko hitro izgubimo preglednost modela, kar pa je pravzaprav največja prednost modeliranja.

Poenoteni proces je generičen proces razvoja programske opreme in mora biti prilagojen na edinstvene pogoje dela v samem podjetju. Pokriva vse faze razvoja informacijskega sistema od faze zbiranja zahtev, analize, načrtovanja, do same implementacije. Prednost same metodologije je v tem, da so prehodi med fazami neproblematični in tekoči. Primeri uporabe omogočajo sledenje zahtevam ali informacijskim potrebam za razvoj programske rešitve, lahko rečemo, da so zahteve vhodni parameter v poenoteni proces. Cilj poenotenega procesa pa je izdelati robustno arhitekturo programske rešitve, saj mora programska rešitev ohraniti svojo osnovno funkcionalnost v procesu nenehnih sprememb. Poenoteni proces je iterativen, lažje rešujemo manjše projekte kot velike projekte, in inkrementalen, manjše podprojekte počasi dopolnjujemo, tako da programska rešitev počasi dobiva popolno funkcionalnost.

Potrebno se je zavedati, da je proces razvoja prvega prototipa programske rešitve s pomočjo UML daljši kot pri klasičnem pristopu, saj potrebujemo nekaj časa, da osvojimo UML orodje in obvladamo tehniko risanja diagramov. Poleg tega je za prvo programsko rešitev potrebno na novo napisati knjižnico programskih modulov, ki omogočajo programski rešitvi dodatne funkcionalnosti. Dodajanje novih funkcionalnosti programske rešitve s pomočjo UML orodja je zelo enostavno, pomembno pa je tudi to, da zaradi robustnosti arhitekture te nove funkcionalnosti ne ogrozijo stabilnost delovanja programske rešitve. Ko enkrat imamo nek prototip programske rešitve, neko znanje o UML orodju in knjižnico programskih modulov, je

razvoj podobnih programskih rešitev veliko hitrejši kot pri klasičnem pristopu. Zaradi tega razvoj programske rešitve s pomočjo UML podjetju omogoča hitro izdelavo novih komplementarnih izdelkov in hitrejšo pridobitev novih segmentov trga, kar pa je končni cilj vsakega podjetja.

LITERATURA

1. Arlow Jim, Neustadt Ila: UML and The Unified Process. Harlow, England:Addison-Wesley, 2002, 396 str.
2. Booch Grady: Object- Oriented Analysis and Design with Applications. Rational, Santa Clara, California: The Benjamin/ Cummings, 1994. 589 str.
3. Booch Grady, Rumbaugh James, Jacobson Ivar: The Unified Modeling Language User Guide. Reading, Massachusetts: Addison- Wesley, 1998, 482 str.
4. Douglass Powel Bruce: Doing Hard Time. Developing Real-Time Systems with UML, Objects, Frameworks and Patterns. New Jersey: Addison-Wesley, 1999. 747 str.
5. Coad Peter, Yourdon Edward: Object-Oriented Design. Englewood Cliffs: Prentice Hall, 1991. 197 str.
6. Damij Talib, Grad Janez, Jaklič Jurij: Izbrane teme iz informacijske tehnologije. Ljubljana, Ekonomska fakulteta, 1995. 316 str.
7. Demarco Tom, Boehm W. Barry: Controlling Software Projects: Management, Measurement, and Estimates. New Jersey: Prentice Hall, 1986. 296 str.
8. Eckel Bruce: Thinking in C++. New Jersey: Prentice Hall, 2000. 814 str.
9. Eriksson Hans-Erik, Penker Mangus: UML Toolkit. New York: John Wiley&Sons. 1998. 397 str.
10. Fowler Martin, Scott Kendall: UML Distilled. A Brief Guide to the Standard Object Modeling Language. Second Edition. Boston: Addison-Wesley, 2000. 185 str.
11. Grad Janez, Jaklič Jurij: Baze podatkov. Ljubljana: Ekonomska fakulteta, 1996. 256 str.
12. Graham Ian: Object Oriented Methods. New Jersey, 1994. 470 str.
13. Grehan Rick, Moote Robert, Cyliax Ingo: Real Time Programming. A guide to 32-bit Embedded Developement. New Jersey: Addison-Wesley, 1999. 720 str.
14. Gričar Jože, Piskar Sebastjan: Sistemski inženiring. Metodologija izgradnje sistemov. Ljubljana: Zavod za organizacijo poslovanja, 1985. 255 str.
15. Hauc Anton: Projektni management. Ljubljana: GV Založba, 2002. 340 str.
16. Hoffer Jeffrey A., George Joey F., Valacich Joseph S.: Modern Systems analysis and design. Reading: Benjamin/Cumings, 1996. 896 str.
17. Jacobson Ivar: Object-Oriented Software Engineering, Reading: Addison-Wesley, 1994. 528 str.
18. Johanssen H. Anne Mette: Configuration Management Principles and Practice. New Jersey: Addison-Wesley, 2002. 432 str.

19. Kahmann Martin, Zayer Peter: Elektrizitätsmesstechnik. Frankfurt am Main: VWEW Energieverlag, 2003. 824 str.
20. Kovačič Andrej: Načrtovanje in gradnja informacijskih sistemov. Ljubljana: DZS, 1994. 316 str.
21. Kovačič Andrej: Informatizacija poslovanja. Ljubljana: Ekonomska fakulteta, 1998. 214 str.
22. Leffingwell Dean, Widrig: Managing Software Requirements: A Unified Approach. New Jersey: Addison-Wesley, 1999. 528 str.
23. Martin James: Principles of Object-Oriented Analysis and Design. Englewood Cliffs: Prentice-Hall, 1993. 412 str.
24. Minzberg H., Ahlstrand B., Lampel J.: Strategy Safari. New York: The Free Press, 1998. 399 str.
25. Pogorelc Janez, Terbuc Martin: Specificiranje, dokumentiranje in izdelava programske opreme s programskim jezikom C. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, 2000. 197 str.
26. Powel Douglass Bruce: Real Time UML. Developing Efficient Objects for Embedded Systems. New Jersey: Addison-Wesley, 1998. 400 str.
27. Rozman Rudi, Kovač Jure, Koletnik Franc: Management. Ljubljana: Gospodarski Vestnik, 1993. 312 str.
28. Rumbaugh James, Jacobson Ivar, Booch Grady: The Unified Modeling Language Reference Manual. Reading, Massachusetts: Addison- Wesley, 1999. 550 str.
29. Sturm Jake: VB 6 UML. Design and Development. Birmingham: Wrox Press, 1999. 581 str.
30. Ward John, Peppard Joe: Strategic Planning for Information Systems. West Sussex: John Wiley and Sons, 2002. 620 str.
31. Wiegers Eugene Karl: Software Requirements. New York: Microsoft Press, 1999. 350 str.
32. Wright Peter, Kroll Mark J., Parnell John: Strategic Management: Concepts and Cases. Englewood Cliffs: Prentice Hall, 1996. 1022 str.

VIRI

1. UML Resource Page. [URL: http://www.omg.org/uml/], Object Management Group, 2.4.2003.
2. Specifikacija programske opreme: Virtualni števec. Kranj : Iskraemeco d.d., 2001, str. 41.
3. Živkovič Aleš, Heričko Marjan, Rozman Ivan: Vpeljava poslovnega modeliranja in primerov uporabe za uspešnejše izvajanje programskih projektov. Kranj : Organizacija, 2002, str. 312-319.

4. Živkovič Aleš, Heričko Marjan: Uporabnost jezika za objektno modeliranje UML . Maribor, Univerza za elektrotehniko, računalništvo in informatiko, 2004.
5. OMG Unified Modeling Language: Specification. Version 1.4, 2001, str.566.
6. Kobryn Cris: UML 2001 A Standardization Odyssey. New York: Communication Of The ACM, Oktober 1999, str 29-37.
7. Alojzij Mišmaš: So naši uporabniki in naročniki zadovoljni? [URL: http://www.drustvo-informatika.si/dogodki/arhiv/dsi2001/sekcija_g/mismas.doc], 13.11.2003.
8. Rhapsody User Guide. Version 4.1, 1997, str. 606.
9. OBIS: Object Identification System. [URL: http://www.dlms.com/en/faq/OBISnames_021216.pdf], DLMS User Association, 15.1.2004.
10. VDEW Lastenheft - Zahtevnik za elektronske industrijske števce, verzija 2.1, 2001.