View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ePrints.FRI

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Dejan Kostadinovski

Uporaba načrtovalskih vzorcev pri agilnem razvoju programske opreme

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Viljan Mahnič

Ljubljana 2014

Rezultati diplomskega dela so intelektualna lastnina avtorja. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite značilnosti agilnega razvoja programske opreme in pomen pravilnega načrtovanja za zagotavljanje kakovostnih in na spremembe odpornih programskih rešitev. Opišite najpomembnejše načrtovalske principe, s pomočjo katerih se izognemo predčasnemu »gnitju« programske kode, in načrtovalske vzorce, ki se uporabljajo pri reševanju tipičnih problemov, s katerimi se pogosto srečujemo v praksi. Uporabo teh vzorcev prikažite na konkretnem primeru razvoja računalniške aplikacije v programskem jeziku *Java*.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Dejan Kostadinovski, z vpisno številko **63100360**, sem avtor diplomskega dela z naslovom:

Uporaba načrtovalskih vzorcev pri agilnem razvoju programske opreme

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Viljana Mahniča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 10. septembra 2014 Podpis avtorja:

Mentorju izr. prof. dr. Viljanu Mahniču se zahvaljujem za njegovo vodenje, dosegljivost in potrpežljivost kakor tudi za vse njegove nasvete in predloge pri izdelavi diplomske naloge.

Posebna zahvala gre tudi moji družini, prijateljem, sošolcem in sodelavcem za podporo v celotnem času študija.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Agilni razvoj programske opreme **3**

2.1 Manifest agilnega razvoja programske opreme 4

2.2 Ekstremno programiranje (ang. Extreme Programming) 6

2.3 Načrtovanje (ang. Planning) 9

2.4 Testiranje in Preurejanje (ang. Testing and Refactoring) 10

3 Agilno načrtovanje **13**

3.1 Principi v agilnem načrtovanju 15

3.2 Vzorci agilnega razvoja programske opreme 22

4 Uporaba vzorcev pri razvoju programa *WAVCutter* **33**

4.1 Uporaba vzorca *Šablonska metoda* 35

4.2 Uporaba vzorca *Opazovalec* 39

4.3 Uporaba vzorca *Edinec* 42

4.4 Uporaba vzorca *Ničelni Objekt* 44

Sklepne ugotovitve **47**

Seznam uporabljenih kratic

kratica	angleško	slovensko
SRP	The Single-Responsibility Principle	Princip ene same odgovornosti
OCP	The Open-Closed Principle	Princip odprt-zaprt
LSP	The Liskov Substitution Principle	Princip substitucije Barbare Liskov
DIP	The Dependency-Inversion Principle	Princip obratne odvisnosti
ISP	The Interface-Segregation Principle	Princip ločenih vmesnikov

Povzetek

Namen diplomske naloge je preučiti načrtovalske vzorce in njihovo uporabo pri agilnem razvoju programske opreme z uporabo objektno usmerjenih jezikov, kot je na primer *Java*. Drugi cilj diplomske naloge pa je preizkusiti tipične primere uporabe vzorcev na čim bolj konkretnem primeru iz prakse.

Najprej so predstavljeni pomen in značilnosti agilnih metodologij za razvoj programske opreme. Potem so prikazani vzroki, zakaj programska oprema postane neodzivna in krhka. Preučili in raziskali smo posamezne vzorce pri objektno usmerjenem razvoju programske opreme in nekatere od njih smo tudi praktično uporabili.

V praktičnem delu naloge so prikazane težave in zapleti pri razvoju programa *WAVCutter*. S pomočjo uporabe načrtovalskih vzorcev so bile določene težave odstranjene in poudarjena je enostavnost celotnega programa.

Ključne besede: agilni razvoj programske opreme, načrtovalski vzorci, java, ekstremno programiranje.

Abstract

The purpose of the thesis is to study design patterns and their use in agile software development, using object-oriented programming languages like *Java*. The second objective of the thesis is to examine typical examples of the use of patterns in the most concrete case studies.

First part of this thesis presents the importance and characteristics of agile methodologies for software development. Then are shown the reasons why software becomes unresponsive and fragile. We studied and researched individual patterns in object-oriented software development and some of them practically used.

The practical part of the thesis shows the difficulties and complications in the development of the application *WAVCutter*. Through the use of design patterns some difficulties have been removed and the simplicity of the entire application has been emphasized.

Keywords: agile software development, design patterns, java, extreme programming.

Poglavje 1

Uvod

Hitrejši razvoj programske opreme posledično povzroča tudi pogoste spremembe in nadgradnje v razvoju samih aplikacij. Razvojne skupine porabijo manj razvojnega časa, ker se aplikacije gradijo hitreje, vseeno pa končni izdelek programske opreme mora ostati kakovosten in trden. Za doseganje kakovosti in odpornosti izdelka na spremembe je potreben dober načrt. Pri načrtovanju izdelka v agilnem razvoju programske opreme so zelo pomembni principi, vzorci in prakse ter tudi ljudje, ki povezujejo te principe, vzorce in prakse ter skrbijo za njihovo pravilno delovanje.

Vzorci se uporabljajo za reševanje tipičnih, ponavljajočih se problemov. Vzorec predstavlja posplošeno rešitev za določeno vrsto problema. Uporaba vzorcev omogoča načrtovalcu, da pride do rešitve veliko hitreje kot v primeru, ko si mora rešitev pripraviti sam. Povsod tam, kjer se soočimo z enakim problemom, lahko uporabimo isto rešitev. Poleg tega vzorci zagotavljajo, da je rešitev bolj prilagodljiva in zanesljiva.

V poglavju 2 smo najprej predstavili pomen agilnega razvoja programske opreme, ki ga je Martin v svoji knjigi [3] predstavil kot »*sposobnost, da se programsko opremo hitro razvije, pri soočanju s hitro spreminjajočimi se zahtevami*«. V nadaljevanju smo navedli manifest agilnega razvoja in predstavili elemente ekstremnega programiranja.

Poglavje 3 na začetku opisuje simptome, ki napovedujejo gnitje (ang. *rotting*) programske opreme in principe, ki ohranjajo kakovost načrta in preprečujejo prihodnje gnitje izvorne kode. V drugem delu poglavja smo podrobneje obdelali uporabo vzorcev pri agilnem razvoju programske opreme. Povedali smo, da so

vzorci najvišja stopnja preoblikovanja programske opreme in da uporaba vzorcev omogoča izboljšanje notranje strukture kode. Za vsak posamezen vzorec smo pojasnili njegovo uporabo, podali praktični primer uporabe in pojasnili, kako vzorec pomaga pri izboljšanju notranje strukture kode.

V poglavju 4 smo najprej opisali delovanje programa *WAVCutter*, potem pa smo tipične primere uporabe vzorcev preizkusili pri razvoju naše aplikacije. Za vsak vzorec smo navedli vzrok za njegovo uporabo. Deli izvorne kode, kjer smo vzorce uporabili, so v tem poglavju prikazani in s tem smo hoteli bralca prepričati, da uporaba vzorcev v resničnem projektu dejansko izboljša načrt in strukturo programske opreme.

Poglavje 2

Agilni razvoj programske opreme

Agilni razvoj predstavlja iterativen in inkrementen razvoj programske opreme. Martin ga je v svoji knjigi [3] predstavil kot »*sposobnost, da se hitro razvije programsko opremo, pri soočanju s hitro spreminjajočimi se zahtevami*«. To dosežemo s sprotnim preverjanjem trenutne smeri v samem razvoju in odstranjevanjem morebitnih ovir, ki se lahko pojavijo v poznejši fazi razvoja. Razvoj je iterativen, ker se stanje programske opreme ocenjuje v kratkih časovnih obdobjih (običajno med 1 in 4 tedne). V vsaki iteraciji lahko agilna skupina (razvijalci) zagotovi že delujočo funkcionalnost projekta oziroma nadgradi obstoječo opremo z novo funkcionalnostjo. V agilnem razvoju je vsak vidik razvoja, kot so načrt, zahtevki itd., sproti pregledan in preučen. Na koncu vsake iteracije, razvojna skupina še enkrat oceni stanje programske opreme in v primerjavi s tradicionalnim razvojem, lahko spremeni smer razvoja, ne da bi ta sprememba vplivala na dosedanji izdelek.

Agilni razvoj ponuja alternative, ki jih ne ponuja tradicionalno vodenje projektov. Te alternative se v glavnem uporabljajo pri razvoju programske opreme in omogočajo hitro prilagajanje spremembam v zahtevah naročnika.

2.1 Manifest agilnega razvoja programske opreme

Skupina industrijskih ekspertov, ki so se sami poimenovali kot Agilna Aliansa (ang. *The Agile Alliance*), se je leta 2001 sestala, da bi določila principe razvoja programske opreme, ki bodo omogočale razvojni skupini hitrejši in prilagodljiv razvoj programske opreme. V naslednjih mesecih je skupina sestavila manifest agilnega razvoja programske opreme (ang. *The Manifesto of the Agile Alliance*). V tem manifestu poudarjajo naslednje vrednote [15]:

- **Posamezniki in interakcije** pred procesi in orodji
- **Delujoča programska oprema** pred vseobsežno dokumentacijo
- **Sodelovanje s stranko** pred pogodbenimi pogajanji
- **Odziv na spremembe** pred togim sledenjem načrtom

»Z drugimi besedami, četudi cenimo dejavnike na desni, vseeno bolj cenimo tiste na levi [15].«

V nadaljevanju bomo pogledali, kaj točno vsaka od teh vrednot pomeni.

Posamezniki in interakcije pred procesi in orodji

Skupina meni, da so ljudje in njihovo medsebojno sodelovanje najpomembnejše sestavine uspeha. Dober postopek ne bo zaščitil projekta pred neuspehom, če razvojna skupina nima dobrih »igralcev«. Prav tako se skupina dobrih igralcev lahko sooči z neuspehom, če igralci ne znajo komunicirati med seboj. Odličen igralec ni tisti, ki je strokovnjak na svojem področju, ampak tisti, ki zna komunicirati in sodelovati z ostalimi igralci v svoji skupini.

Pred vsakim začetkom je treba najprej poskrbeti za dobro ekipo. Čez čas bo ekipa sama poskrbela, kateri procesi in orodja ji najbolj ustrezajo.

Delujoča programska oprema pred vseobsežno dokumentacijo

Izvorna koda brez dokumentacije predstavlja katastrofo tako za nove člane kot za same člane razvojne skupine. Vendar pa mora biti dokumentacija kratka in obsegati samo najvišje nivoje sistema.

Če bi bila dokumentacija zelo kratka, kako naj bi novi član razvojne skupine razumel delovanje in podrobnosti v izvorni kodi? Agilna Aliansa na to vprašanje odgovori: »*Dva dokumenta, ki sta najboljša za prenos informacije novimi člani, sta razvojna skupina in sama izvorna koda*«.

Sodelovanje s stranko pred pogodbenimi pogajanji

Iz prakse vemo, da se projekt nikoli ne konča, ne da bi se v prvotnem načrtu kaj spremenilo, nadgradilo ali celo, da bi se projekt na novo skiciral. Te prakse in nevšečnosti se pojavijo tudi v podjetjih, kjer z načrti in zahtevami delajo izkušeni inženirji.

V agilnem razvoju programske opreme naročnik izdelka in razvojna skupina pogosto komunicirata. Naročnik izdelka za vsako iteracijo določi sprejemne teste, razvojna skupina pa poskrbi, da naročnik te teste tudi sprejeme. Na ta način razvoj projekta poteka v pravo smer in je razvojna skupina sposobna reagirati na hitre spremembe.

Odziv na spremembe pred togim sledenjem načrtom

Ljudje spreminjajo svoje prioritete zaradi različnih razlogov. Ko sistem narašča in napreduje, vodja projekta in naročnik izdelka izboljšujeta svoje znanje v celotnem izdelku in potrebah razvijalcev v razvojni skupini. Tako bodo nekateri zahtevki dodani, drugi pa spremenjeni ali celo odstranjeni iz končnega produkta. Prav tako se tehnologija in programska oprema s časom spreminjata, tako da bo potrebno spremeniti tudi izvorno kodo. Na kratko: spremembe se pojavljajo in zato jih mora razvojna skupina z lahkoto sprejeti.

Manifest priporoča, da se podrobni načrt izdeluje samo za prihodnji teden, grobi načrt za naslednje tri mesece in zelo surovi načrt za časovne termine daljše od treh mesecev.

2.1.1 Principi v ozadju agilnega manifesta

Da bi lahko ljudem pomagali boljše in lažje razumeti razvoj programske opreme s pomočjo agilne metode, so pri Agilni Aliansi povzeli celotno filozofijo v 12 principih [20]. Principi, ki so jih združili, so naslednji:

- Naša najvišja prioriteta je zadovoljiti stranko z zgodnjim in nepretrganim izdajanjem vredne programske opreme.
- Sprejemamo spremembe zahtev, celo v poznih fazah razvoja. Agilni procesi vprežejo tovrstne spremembe v prid konkurenčnosti naše stranke.
- Delujočo programsko opremo izdajamo pogosto, znotraj obdobja nekaj tednov, do nekaj mesecev, s preferenco po krajšem časovnem okvirju.
- Poslovneži in razvijalci morajo skozi celoten projekt dnevno sodelovati.
- Projekte gradimo okrog motiviranih posameznikov. Omogočimo jim delovno okolje, nudimo podporo in jim zaupamo, da bodo svoje delo opravili.
- Najboljša in najučinkovitejša metoda posredovanja informacij razvojni ekipi in znotraj ekipe same, je pogovor iz oči v oči.
- Delujoča programska oprema je primarno merilo napredka.
- Agilni procesi promovirajo trajnostni razvoj. Sponzorji, razvijalci in uporabniki morajo biti zmožni konstantnega tempa za nedoločen čas.
- Nenehna težnja k tehnični odličnosti in k dobremu načrtovanju izboljša agilnost.
- Preprostost – umetnost zmanjševanja količine nepotrebnega dela – je bistvena.
- Najboljše arhitekture, zahteve in načrti izhajajo iz tistih ekip, ki so samoorganizirane.
- V rednih časovnih razdobjih ekipa išče načine, kako postati učinkovitejša ob rednem prilagajanju svojega delovanja.

2.2 Ekstremno programiranje (ang. Extreme Programming)

Ekstremno programiranje je zbirka enostavne in konkretne prakse, ki se skupaj kombinirajo v celovit agilno-razvojni proces. Veliko razvojnih skupin sprejema

ekstremno programiranje kot dobro metodo za razvoj programske opreme s trenutnimi praksami in načeli. Ostali lahko spreminjajo trenutne prakse ali celo dodajajo nove, pomembno je samo, da se obdrži glavni cilj agilnega programiranja – enostavnost. V nadaljevanju bomo definirali vloge, prakse in načela v ekstremnem programiranju.

Naročnik izdelka je posameznik ali skupina, ki definira lastnosti in podrobnosti izdelka in določa njihovo prioriteto. Predstavnik naročnika izdelka mora biti sestavni del razvojne skupine. Razvijalci programske opreme in naročnik izdelka morajo delati skupaj oziroma imeti pogoste medsebojne interakcije. Tako se bodo zavedali težav, s katerimi se sooča vsak posameznik in s skupnim sodelovanjem bodo poskusili te težave čim prej odpraviti.

Uporabniška zgodba je oblika, v kateri so podane specifikacije in zahteve za določeno lastnost izdelka. Uporabniške zgodbe omogočajo, da zapišemo toliko, kot je potrebno in, da vemo kaj je treba narediti. Z uporabo uporabniških zgodb lahko ocenimo obseg potrebnega dela in tako načrtujemo nadaljnje delo. Tudi komunikacija med uporabniki in naročnikom poteka na primeren način [1].

Uporabniško zgodbo predstavlja kartica, na kateri je zapisan opis funkcionalnosti. Služi kot orodje za planiranje (ang. *planning tool*), ki ga uporablja naročnik za kreiranje časovnice, ki bi opisovala, kdaj naj bi potekala implementacija določene funkcionalnosti. Časovna implementacija se izračuna glede na prioriteto in od razvijalcev ocenjeno zahtevnost zgodbe (ang. *estimated cost*) [3].

Sprejemni testi podrobno opisujejo uporabniško zgodbo. Sestavi jih naročnik izdelka v obliki skriptnega jezika in s tem omogoči, da se testi izvršujejo samodejno in ponavljajoče večkrat na dan. Ko uporabniška zgodba uspešno opravi vse sprejemne teste, se doda v seznam uspešno opravljenih zgodb in razvijalci poskrbijo, da se zgodba nikoli več ne odstrani iz tega seznama.

Programiranje v parih predstavlja razvoj programske opreme, kjer dva razvijalca skupaj razvijata isto celoto izvorne kode. Programiranje poteka na ta način, da en razvijalec tipka kodo, drugi pa išče napake in izboljšave. Po določenem času

razvijalca vloge zamenjata. Na ta način se zmanjšuje število napak v izvorni kodi. Oba razvijalca imata enake avtorske pravice do napisane izvorne kode.

Testno-voden razvoj predstavlja razvoj programske opreme in razvoj testnih primerov, ki preprečujejo napačno delovanje celotne funkcionalnosti izdelka. Obstajajo posebna orodja (JUnit, CTest, QtTest itd. [14]), ki izvršujejo testne primere in pisanje testov ločijo od izvorne kode.

Skupno lastništvo kode (ang. Collective Ownership) vsakemu paru dodeli pravice, da pregleda katerikoli modul in ga izboljša. Programiranje v parih in vnaprej pripravljeni testi preprečujejo vnos napak, povečuje pa se možnost učenja koristnih prijemov [2]. Skupno lastništvo kode omogoča svobodo za razvijalce, da se naučijo kaj novega izven svoje specializiranosti.

Igra planiranja (ang. The Planning Game) naročniku omogoča, da se odloči, koliko je pomembna določena lastnost (funkcionalnost) izdelka, razvijalcem pa omogoča preko skupnega glasovanja določiti, kakšna bo cena implementacije. S pomočjo Igre planiranja razvijalci izdelajo podroben načrt dela v eni iteraciji. Glede na količino dela v eni iteraciji lahko približno določijo čas razvoja celotnega izdelka.

Razvijalci se trudijo narediti čim bolj **enostaven načrt** za funkcionalnosti, ki jo razvijajo v trenutni iteraciji. Z enostavnim in preprostim sistemom se izognemo prevelikim zapletom v začetnih iteracijah. Razvijalci dajo večji poudarek na lastnosti sistema, ki so potrebne zdaj, in ne razmišljajo o funkcionalnostih, ki bodo potrebne v kasnejši fazi razvoja. Dopolnjevanje in razširjanje sistema se lahko implementira tudi v kasnejših iteracijah.

Preurejanje naredi vrsto majhnih, a pomembnih transformacij v izvorni kodi, ki ne vplivajo na pravilno delovanje kode, vendar na preprostem načrtovanju sistema. Transformacija sistema se zgodi, ko se vse majhne transformacije kombinirajo. Takrat se tudi začuti lepota čiste kode.

2.3 Načrtovanje (ang. Planning)

V tem podpoglavju bomo poskusili bolj podrobno opisati kako poteka načrtovanje v realnem projektu.

Na samem začetku naročnik izdelka in razvojna skupina **analizirajo** samo zelo pomembne uporabniške zgodbe. Naročnik izdelka izbere samo tiste zgodbe, ki se mu zdijo najbolj primerne za implementacijo v naslednjih nekaj iteracijah. Razvojna skupina s skupnimi ocenami razvijalcev poskuša **oceniti** določeno zgodbo. Vsaki zgodbi pripišejo število točk, ki bodo predstavljali relativno oceno zahtevnosti te zgodbe.

Lahko se zgodi, da določene zgodbe ni mogoče oceniti, ker je zelo obsežna ali pa zanemarljivo majhna. Če je zgodba prevelika, se mora **razdeliti** na več manjših zgodb, če je premajhna pa naj se **združi** z drugo majhno zgodbo. Po združitvi ali razdelitvi je treba zgodbo še enkrat oceniti, da imamo natančnejši podatek, koliko bo zgodba časovno trajala, oceno moramo nato zmnožiti s faktorjem »hitrost« (ang. *velocity*). Martin je v svoji knjigi [3] opisal *hitrost* kot *»število dni za realizacijo ene točke«*. Na primer, če je naša hitrost določena kot »dva dni za eno uporabniško točko« in je zgodba ocenjena s štirimi točkami, potem je časovno trajanje implementacije te zgodbe enako 8 dni.

V [3] je opisan primer, kako eno veliko zgodbo razbijejo na več manjših. Primer je naslednji:

> *»Uporabniki lahko varno nakažejo denar na svoj račun, lahko dvignejo denar s svojega računa in lahko prenašajo denar med svojimi računi.«*

Ker je zgodba zelo obsežna, je razdeljena na več manjših:

- Uporabniki se lahko prijavijo v sistem.
- Uporabniki se lahko odjavijo iz sistema.
- Uporabniki si lahko položijo denar na svoj bančni račun.
- Uporabniki lahko dvignejo denar iz svojega bančnega računa.
- Uporabniki lahko prenesejo denar iz svojega računa na drug bančni račun.

Ko so vse zgodbe ocenjene, naročnik izdelka lahko približno oceni, kolikšna bo cena in trajanje implementacije za posamezno uporabniško zgodbo. Potek razvojnega projekta se začne s sestankom za **planiranje izdaje** (ang. Release Planning), kjer so prisotni razvijalci in naročnik izdelka. Na sestanku se udeleženci dogovorijo za datum prve izdaje, ki je običajno dva do štiri mesece v prihodnosti.

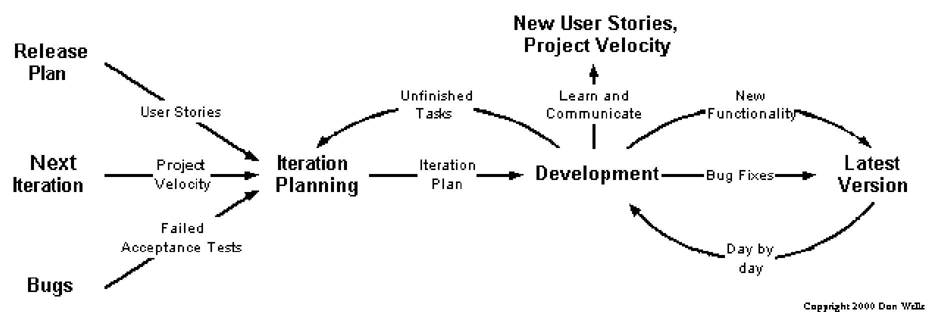
Naslednji sestanek je sestanek za **planiranje iteracije** (ang. Iteration Planning). Na tem sestanku se določi časovno obdobje iteracije, ki v ekstremnem programiranju običajno traja od enega do treh tednov [11]. Naročnik izdelka izbere uporabniške zgodbe, za katere želi, da jih razvijalci implementirajo v tekoči iteraciji. Razvijalci se odločijo za vrstni red implementacije posameznih zgodb. Iteracija se konča na vnaprej določen datum, četudi vse zgodbe v iteraciji niso dokončane. Nedokončane zgodbe in zgodbe, ki niso bile sprejete, se prenesejo v eno izmed naslednjih iteracij.

Razvijalci vsako zgodbo razbijejo na več nalog, pri čemer se ena naloga lahko implementira v najmanj 4 oziroma največ 16 urah. Vsak razvijalec se prijavi za eno ali več nalog, ki jih hoče implementirati v tekoči iteraciji. **Planiranje in razbijanje zgodb** na naloge (ang. Task Planning) se zaključi, ko je vsaka naloga iz seznama nalog dodeljena natanko enemu razvijalcu.

Na sliki 2.1 je prikazan potek ene iteracije v ekstremnem programiranju. Vsaka iteracija je sestavljena iz prej določenih uporabniških zgodb. Zgodbe, ki so uspešno sprejete, postanejo sestavni del naslednje izdaje. Neuspešne zgodbe se označijo za implementacijo v nekaj od naslednjih iteracij. Naročnik dobi novo delujočo različico sistema na vnaprej določenem datumu izdaje.

2.4 Testiranje in Preurejanje (ang. Testing and Refactoring)

Testiranje programske opreme je ključnega pomena za razvoj stabilnega sistema. Pisanje testov ne zagotavlja samo pravilnost delovanja sistema, ampak v celoti prikaže načrt končnega izdelka. V agilnem razvoju programske opreme testne primere pišemo ločeno od izvorne kode. Format sprejemnih testov omogoča samodejno nenehno preverjanje pravilnosti delovanja programske opreme. Napiše jih



Slika 2.1: Potek ene iteracije v ekstremnem programiranju (povzeto iz [11]).

sam razvijalec ali pa naročnik izdelka v obliki sprejemnih testov. Sprejemni testi so merilo, kako hitro in kakovostno napreduje razvoj sistema.

Martin Fowler je v svoji knjigi [3] definiral **preurejanje** (ang. Refactoring) kot »*način spreminjanja kode, pri katerem se njen zunanji pomen ne spremeni, vendar se notranja struktura kode izboljša*«.

Vsak modul programske opreme ima tri funkcije: prva funkcija je tista funkcija, ki jo modul izvršuje, druga funkcija je ta, da si modul lahko privošči spremembe v prihodnosti, tretja funkcija je dobra komunikacija z bralcem, t.j. berljiva koda.

Poglavje 3

Agilno načrtovanje

V agilni skupini načrt končnega izdelka nastaja skupaj s programsko opremo. Razvijalci se osredotočijo samo na trenutni načrt sistema, ne da bi izgubljali čas z razvijanjem podpore za funkcije, ki jih bodo potrebovali v prihodnosti. Z vsako iteracijo se načrt sistema izboljšuje in v danem trenutku predstavlja najboljšo strukturo sistema, ki se jo da razviti. Razvijalcem vsakič ne uspe izdelati dobrega načrta izdelka.

V [3] so opisani simptomi, ki napovedujejo ≫gnitje≪ programske opreme.

- Togost (ang. Rigidity) : načrt je težko spremeniti
- Krhkost (ang. Fragility) : načrt se lahko zlomi
- Nepremičnost (ang. Immobility) : načrt je težko ponovno uporabiti
- Viskoznost (ang. Viscosity) : težko je narediti pravo stvar
- Nepotrebna kompleksnost (ang. Needless complexity) : načrt je kompleksen in ga je težko razumeti
- Nepotrebno ponavljanje (ang. Needless repetition) : ponavljajoča izvorna koda
- Nerazumljivost (ang. Opacity) : neorganizirane funkcionalnosti

V nadaljevanju bomo bolj podrobno opisali posamezene sindrome. V poglavju 3.1 bomo s pomočjo agilnega principa razložil kako ≫gnitje≪ programske opreme

znižamo na minimalno vrednost.

Togost se pojavi, ko ne moremo z lahkoto spremeniti dela programske opreme, četudi je ta sprememba enostavna. Tako se lahko za določeno spremembo zgodi, da moramo popraviti velik del naše izvorne kode ali v najslabšem primeru tudi kodo napisati znova.

Krhkost nakazuje to, da lahko ena majhna sprememba v izvorni kodi povzroči nepravilno delovanje kode v že delujočih področjih. Ta simptom se pogosto pojavlja v področjih, ki so medsebojno konceptualno neodvisni.

Pogosto se dogaja, da določen del programske opreme ne moremo uporabiti v drugih sistemih kljub temu, da smo programsko opremo poskusili prilagoditi novemu sistemu. V tem primeru pride do simptoma **nepremičnosti**.

Viskoznost se pojavi v projektu, kjer je težko ohraniti načrt izdelka. To se pogosto dogaja, ko se razvijalci odločijo za hiter in poceni razvoj, ki začasno reši težave, ampak ne ohrani načrta in enostavnosti celotnega projekta.

Razvijalci si pripravljajo »teren« s funkcionalnostmi, ki se lahko izkažejo kot uporabni v nadaljnjem razvoju. Ta »teren« vsebuje spremenljivke, konstruktorje in metode, ki niso potrebne v trenutnem razvoju. Na ta način načrt vsebuje **nepotrebno kompleksnost**, s katero tudi zgine celotna enostavnost sistema.

Nepotrebno ponavljanje se zgodi, ko vsak razvijalec napiše svojo različico kode za isto funkcionalnost sistema. Na ta način izvorna koda postane nepregledna, težka za razumevanje in vzdrževanje ter napake v kodi (ang. *bugs*) se mora popraviti v vsaki različici kode.

Nerazumljivost pomeni, da je modul v izvorni kodi težko razumljiv, to pomeni, da ima modul neorganizirano funkcionalnost. Ko razvijalec napiše določeno funkcionalnost, se mu v tem trenutku zdi, da je implementacija te funkcionalnosti najbolj profesionalna in enostavna. Ko preteče čas in ko projekt zraste, funkcionalnosti postanejo neorganizirane do te mere, da jih tudi sam razvijalec ne more razumeti in povezati med seboj.

3.1 Principi v agilnem načrtovanju

3.1.1 Princip ene same odgovornosti (ang. The Single-Responsibility Principle)

V objektnem programiranju, princip ene same odgovornosti (v nadaljevanju SRP) navaja, da ima vsak kontekst (razred, funkcija, spremenljivka, itd.) samo eno odgovornost oziroma kontekst naj bi se spremenil samo v primeru, če obstaja samo en razlog za to spremembo [?, 21]. Ta princip določa, da če imamo dva različna razloga za spremembe moramo razdeliti odgovornost na dva različna razreda. Vsak razred bo odgovoren za samo eno spremembo in v prihodnosti, če bomo morali narediti spremembo, jo bomo naredili v razredu, ki obravnava tisto odgovornost. V [21] je podan preprost primer, kjer se SRP uporablja. Za ta primer so uporabili en objekt, ki predstavlja sporočilo e-pošte in en `IEmail` vmesnik. Izvorna koda 3.1 je podana v nadaljevanju.

Izvorna koda 3.1: Vmesnik `IEmail`

```
interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(String content);
}

class Email implements IEmail {
        public void setSender(String sender) { }
        public void setReceiver(String receiver) { }
        public void setContent(String content) { }
}
```

Če bolj podrobno analiziramo primer, bomo videli, da ima vmesnik dve odgovornosti. Prva je ta, da se lahko v prihodnosti zgodi, da poleg dosedanjega protokola kot so POP3 in IMAP, moramo dodati funkcionalnosti še za druge e-poštne protokole. Druga odgovornost oziroma sprememba, ki se lahko zgodi je ta, da bi lahko v prihodnosti sporočila imela podporo za HTML format in ne bodo več predstavljena kot objekt String. Zaradi tega so razdelili funkcionalnosti v dva različna razreda. Koda, ki uporablja princip SRP je podana v izvorni kodi 3.2.

Izvorna koda 3.2: Uporaba vzorca SRP

```
interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(IContent content);
}

interface IContent {
        public String getAsString();
}

class Email implements IEmail {
        public void setSender(String sender) { }
        public void setReceiver(String receiver) { }
        public void setContent(IContent content) { }
}
```

3.1.2 Princip odprt-zaprt (ang. The Open-Closed Principle)

Dober načrt izdelka mora poskrbeti za pogoste spremembe, do katerih prihaja pri razvoju in vzdrževanju programske opreme. Obstoječa izvorna koda se največ spreminja, ko se doda nova funkcionalnost v obstoječo aplikacijo. Število sprememb v obstoječi izvorni kodi mora biti čim manjše, saj se predpostavlja, da je obstoječa koda že testirana in nove spremembe lahko povzročijo nepravilno delovanje na že delujočih modulih [19].

Princip odprt-zaprt (v nadaljevanju OCP) načeloma določa, da se mora nova funkcionalnost implementirati na ta način, da doda minimalne spremembe v obstoječo kodo oziroma načrt mora biti zasnovan tako, da čim bolj ohranja obstoječo kodo nespremenjeno. Fowler v svoji knjigi [3] navaja, da »*moduli, ki so skladni z OCP, morajo biti odprti za razširitev in zaprti za spremembe*«.

V nadaljevanju si bomo pogledali primer [19], kjer je opisan grafični urejevalnik, ki skrbi za risanje različnih vrst likov. Slabosti izvorne kode 3.3 so:

- Grafični vmesnik moramo testirati za vsak nov lik.
- Razvijalci, ki ne poznajo strukture grafičnega vmesnika, bodo izgubili veliko časa z razumevanjem logike.

- Dodajanje novih likov lahko povzroči nepravilno delovanje logike modula.

Izvorna koda 3.3: Grafični urejevalnik

```
class GraphicEditor {
        public void drawShape(Shape s) {
                if (s.m_type==1)
                        drawRectangle(s);
                else if (s.m_type==2)
                        drawCircle(s);
        }
        public void drawCircle(Circle r) {....}
        public void drawRectangle(Rectangle r) {....}
}

class Shape {
        int m_type;
}

class Rectangle extends Shape {
        Rectangle() { super.m_type=1; }
}

class Circle extends Shape {
        Circle() { super.m_type=2; }
}
```

Izvorna koda 3.4 odpravlja pomanjkljivosti, ki se nahajajo v izvorni kodi 3.3. S popravljenim primerom ne bo več potrebno testirati grafičnega vmesnika za vsako novo obliko, novi razvijalci lahko ustvarijo nove like, ne da bi poznali logiko grafičnega vmesnika in ker smo prestavili logiko za risanje v nov razred, smo zmanjšali nevarnost nepravilnega delovanja pri dodajanju nove oblike.

Izvorna koda 3.4: Uporaba vzorca OCP

```
class GraphicEditor {
        public void drawShape(Shape s)
                s.draw();
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape  {
```

```
        public void draw() {
                // draw the rectangle
        }
}
```

3.1.3 Princip substitucije Barbare Liskov (ang. The Liskov Substitution Principle)

Ves čas oblikujemo programske module in ustvarjamo hierarhije razredov. Potem te razrede razširimo in ustvarimo nekaj novih izpeljanih razredov (ang. *derived classes*). Zagotoviti moramo, da bodo novi izpeljani razredi samo razširili osnovne razrede, ne da bi zamenjali funkcionalnosti starih razredov. Osnovna definicija principa substitucije Barbare Liskov (v nadaljevanju LSP) je:

≫Osnovni tip mora biti nadomestljiv s svojimi podtipi (ang. Subtypes must be substitutable for their base types)[3].≪

Če je S podtip tipa T, potem lahko objekte tipa T nadomestimo z objekti tipa S. Da bi princip bolje razumeli, bomo pogledali primer izvorne kode (Izvorna koda 3.5), podan v [13]. Primer je sestavljen iz dveh razredov: `Rectangle` in `Square`. Osnovni tip je razred `Rectangle`. Razred `Square` je izpeljan iz razreda `Rectangle`.

Izvorna koda 3.5: Kršitev LSP

```
class Rectangle {
        protected int m_width;
        protected int m_height;

        public void setWidth(int width) m_width = width;

        public void setHeight(int height) m_height = height;

        public int getWidth() return m_width;

        public int getHeight() return m_height;

        public int getArea() return m_width * m_height;
}

class Square extends Rectangle {
```

```
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }
}

class LspTest {
        private static Rectangle getNewRectangle()
                return new Square();

        public static void main (String args[]) {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);

                System.out.println(r.getArea());
        }
}
```

Razvijalec pričakuje, da je površina tipa `Rectangle` enaka 50. Rezultat, ki ga izpisuje program je 100. V tem primeru se srečamo s kršitvijo LSP, ker izpeljani razred `Square` spreminja obnašanje osnovnega razreda `Rectangle`.

3.1.4 Princip obratne odvisnosti (ang. The Dependency-Inversion Principle)

Pri oblikovanju programske opreme lahko razrede razdelimo na razrede nizke ravni in razrede visoke ravni. Razrede nizke ravni so tiste, ki implementirajo logiko za osnovne dejavnosti kot so dostop do diska, mrežni protokoli itd. Razredi visoke ravni implementirajo poslovno logiko kot so na primer poslovna pravila. Princip obratne odvisnosti (v nadaljevanju DIP) ima dva namena:

- Moduli visoke ravni ne smejo biti odvisni od modulov nizke ravni. Obe vrsti morata biti odvisni od abstrakcije.
- Abstrakcije ne smejo biti odvisni od podrobnosti. Podrobnosti morajo biti

odvisne od abstrakcije.

Izvorna koda 3.6: Kršitev principa DIP

```
// DIP - Bad example
class Worker {
  public void work(){}
}

class Manager {
  Worker worker;

  public void setWorker(Worker w){
    worker = w;
  }

  public void manage() {
        worker.work();
  }
}

class SuperWorker {
  public void work() {
        //.... working much more
  }
}
```

Izvorna koda 3.7: Implementacija principa DIP

```
// DIP - Good example
interface IWorker {
  public void work();
}

class Worker implements IWorker{
  public void work() {
        // ....working
  }
}

class SuperWorker  implements
    IWorker{
  public void work() {
        //.... working much more
  }
}

class Manager {
  IWorker worker;

  public void setWorker(IWorker w){
        worker = w;
  }

  public void manage() {
        worker.work();
  }
}
```

V [5] je podan primer, kjer je `Manager` razred visoke ravni in `Worker` je razred nizke ravni. V aplikacijo bo treba dodati nov modul, ki bo predstavljal nov tip delavca (ang. *worker*). Če dodamo nov modul v obstoječo kodo, kot je prikazano v izvorni kodi 3.6, moramo potem spremeniti že obstoječi razred `Manager` in moramo ponoviti testiranje delovanja celotnega modula.

Izvorna koda 3.7 podpira DIP in s tem zagotovi, da razred `Manager` ostane nespremenjen in ni več potrebe po ponovnem testiranju delovanja celotnega modula. Razred visoke ravni (`Manager`) in oba razreda nizke ravni (`Worker` in `SuperWorker`)

so zdaj odvisni od abstraktne plasti, t.j. vmesnika `IWorker`.

3.1.5 Princip ločenih vmesnikov (ang. The Interface-Segregation Principle)

Princip ločenih vmesnikov (v nadaljevanju ISP) navaja, da odjemalec ne sme biti prisiljen uporabljati metode, ki je ne potrebuje. Zato ISP razdeli vmesnike, ki so zelo obsežni (imajo veliko metod) v manjše in bolj specifične. Tako bodo odjemalci poznali samo tiste metode, ki so v njihovem interesu [9].

Princip bo bolj razumljiv, če ga uporabimo na konkretnem primeru. Primer, ki je podan v [10] je sestavljen iz enega upravitelja zaposlenih `Manager` in treh tipov delavcev (`Worker`, `SuperWorker` in `Robot`). Vsi delavci lahko delajo, ampak samo `Worker` in `SuperWorker` lahko jesta. Če bi vsi delavci implementirali vmesnik `IWorker` (Izvorna koda 3.8) bi prišlo do hude napake v delovanju sistema, ker delavec `Robot` malice ne potrebuje. Tako bi sistem zabeležil, da je opravljeno več malic kot število delavcev, ki lahko malicajo.

Izvorna koda 3.8: Vmesnik `IWorker`

```
interface IWorker {
        public void work();
        public void eat();
}
```

Princip je postal fleksibilen z uporabo ISP. Načrt principa je podan v izvorni kodi 3.9.

Izvorna koda 3.9: Uporaba principa ISP

```
interface IWorker extends Feedable, Workable {}

interface IWorkable { public void work(); }

interface IFeedable{ public void eat(); }

class Worker implements IWorkable, IFeedable{
        public void work() { ... }

        public void eat() { ... }
}

class Robot implements IWorkable{
```

```
        public void work() { ... }
}

class Manager {
        Workable worker;

        public void setWorker(Workable w) { worker=w; }

        public void manage() { worker.work(); }
}
```

3.2 Vzorci agilnega razvoja programske opreme

V podpoglavju 2.4 smo omenili, da je preoblikovanje (ang. Refactoring) način spreminjanja kode, pri katerem se njen zunanji pomen ne spremeni, vendar se notranja struktura kode izboljša. Najvišja stopnja preoblikovanja je preoblikovanje v vzorce. Zato je priporočeno, da uporabljamo vzorce na že obstoječi zasnovi, kot da bi nove zasnove zgradili sami [3].

Zanimivo za načrtovalske vzorece je, da so nosilci številnih koristnih zasnov idej. Iz tega lahko sklepamo, da lahko postanemo zelo dobri in uspešni razvijalci programske opreme, če se bomo naučili teh vzorcev.

3.2.1 Vzorec *Ukaz* (ang. Command Pattern)

V objektnem programiranju se s pomočjo vzorca *Ukaz* določen objekt uporablja za predstavitev in enkapsulacijo vseh informacij, ki so potrebne za poznejše klicanje določene metode. Informacije vsebujejo ime metode, objekt kateremu tista metoda pripada, in vrednosti parametrov metode [4]. Štirje izrazi so vedno povezani z vzorcem *Ukaz* in to so: ukaz, sprejemnik, prožilec (ang. *invoker*) in uporabnik. Objekt tipa *ukaz* vsebuje objekt sprejemnika in zažene (ang. *invoke*) metodo, ki je vsebovana v razredu sprejemnika. Sprejemnik potem izvrši kodo v določeni metodi. Objekt tipa *ukaz* je ločeno prenesen v objekt *prožilca*. Prožilec lahko tudi shranjuje zgodovino ukazov, ki so se že izvršili. Na ta način uporabniku omogočamo, da laho razveljavi zadnji izvršen ukaz.

Da bi boljše razumeli delovanje in implementacijo vzorca, si bomo pogledali primer, ki je podan v [4] in je prikazan v izvorni kodi 3.10. Primer upošteva

preprosto stikalo, ki ima dva ukaza: vklop in izklop. V primeru se stikalo uporablja za vklop in izklop luči. Prednost implementacije tega vzorca je ta, da se stikalo lahko uporabi s katerokoli napravo. Ker je konstruktor razreda `Switch` sposoben sprejeti vsak podrazred razreda `Command` lahko razred `Switch` nastavimo tudi za zagon motorja.

Izvorna koda 3.10: Primer uporabe vzorca *Ukaz*

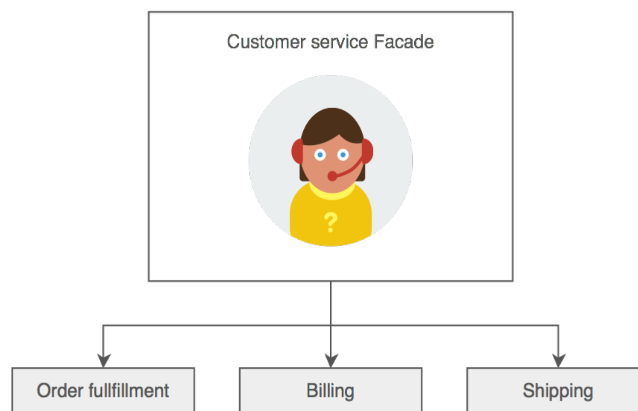
```
// Razred Ukaz
public interface Command {
   void execute();
}

// Razred Prozilec
public class Switch {
   private List<Command> history = new ArrayList<Command>();

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd);
      cmd.execute();
   }
}

// Razred Sprejemnik
public class Light {
   public void turnOn()
      System.out.println("The light is on");

   public void turnOff()
      System.out.println("The light is off");
}

// Podrazred razreda Ukaz
public class FlipUpCommand implements Command {
   private Light theLight;

   public FlipUpCommand(Light light)
      this.theLight = light;

   public void execute()
      theLight.turnOn();
}

// Podrazred razreda Ukaz
public class FlipDownCommand implements Command {
   private Light theLight;
```

```
    public FlipDownCommand(Light light)
        this.theLight = light;

    public void execute()
        theLight.turnOff();
}

// Postopek delovanja
public class PressSwitch {
    public static void main(String[] args){
        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);
        Switch mySwitch = new Switch();

        switch(args[0]) {
            case "ON":
                mySwitch.storeAndExecute(switchUp);
            break;
            case "OFF":
                mySwitch.storeAndExecute(switchDown);
            break;
            default:
                System.out.println("Argument \"ON\" or \"OFF\" is required.");
        }
    }
}
```

V izvorni kodi 3.10 je v metodi `main(String[] args)` prikazen postopek delovanja. Najprej kreiramo sprejemnik, ki ga hočemo kontrolirati. V našem primeru je sprejemnik razred `Light` in vsebuje metodi za vklop in izklop luči. Potem ustvarimo en objekt razreda `FlipUpCommand` in en objekt razreda `FlipDownCommand`. Objekta sta podrazreda razreda `Command`. Objekta vsebujeta isto instanco že ustvarjenega sprejemnika. Nazadnje kreiramo še objekt prožilca, kateremu prenesemo že kreirane komande.

3.2.2 Vzorec *Fasada* (ang. Facade Pattern)

Vzorec *Fasada* predstavlja objekt, ki zagotavlja poenostavljen vmesnik do večje skupine kode oziroma enkapsulira kompleksen podsistem znotraj enotnega vmesniškega objekta. To zmanjšuje čas in napor učenja, potrebnega za uspešno ob-



Slika 3.1: Primer uporabe vzorca Fasada (povzeto iz [6]).

vladovanje sistema. Vzorec *Fasada* ponuja enoten vmesnik do množice vmesnikov podsistemov, definira visoko-nivojski vmesnik za lažjo uporabo podsistemov ter zmanjšuje odvisnosti v zunanji kodi[7].

Vzorec *Fasada* se uporablja predvsem, ko je sistem zelo zapleten in težko razumljiv, ker ima sistem veliko število neodvisnih razredov ali ko izvorna koda sistema ni na voljo. Vzorec vključuje en ovojni razred, ki vsebuje samo funkcije, ki jih zahteva uporabnik. Te funkcije dostopajo do sistema preko fasade in skrivajo implementacijske podrobnosti.

V [6] je predstavljen preprost primer uporabe vzorca *Fasada*. Za lažje razumevanje primera si poglejte sliko 3.1.

≫Potrošniki se srečujejo z vzorcem Fasada pri naročanju iz kataloga. Potrošnik kliče eno številko in govori s predstavnikom storitev za stranke. Predstavnik storitve za stranke deluje kot fasada, ki zagotavlja vmesnik za oddelke za naročanje, fakturiranje izdajo računov in dostavo izdelkov.≪

3.2.3 Vzorec *Šablonska metoda* (ang. Template Method)

Podvojena koda je zelo uničevalna in slaba za načrt programskega modula in povzroča težave na dolgi rok. Zato mora biti poljubno podvajanje kode preprečeno, kadar je to mogoče. Vzorec *Šablonska metoda* predstavlja skelet algoritma v določeni

metodi, ki preprečuje podvajanje izvorne kode v podrazredih. Metoda, ki implementira vzorec, se imenuje šablonska metoda in podrazredom dopušča implementirati nekatere svojih funkcionalnosti, ki se kličejo znotraj algoritma. Na ta način lahko implementiramo različne funkcionalnosti v podrazredih, ne da bi spremenili strukturo algoritma.

V nadaljevanju si bomo v izvorni kodi 3.11 pogledali, kako se vzorec uporablja v splošnem, potem pa bomo v poglavju 4.1 pokazali kako smo vzorec *Šablonska metoda* uporabili v našem programu.

Izvorna koda 3.11: Primer uporabe vzorca *Šablonska metoda*

```
abstract class ExampleTemplate {

        public void templateMethod() {
                // do something

                doMethod1();
                doMethod2();
        }

        abstract void doMethod1();
        abstract void doMethod2();
        // ... other methods
}

class Example extends ExampleTemplate {
        void doMethod1() {
                // implement own logic
        }

        void doMethod2() {
                // implement own logic
        }
}

class UseExample {
        public static void main(String args[]) {
                Example example = new Example();

                example.templateMethod();
        }
}
```

Abstraktni razred `ExampleTemplate` vsebuje šablonsko metodo `templateMethod()`. Ta metoda v svoji logiki kliče druge metode (`doMethod1()` , `doMethod2()`), ki jih bo vsak podrazred implementiral na svoj način. Ker je šablonska metoda skupna za vse podrazrede, je preprečeno pisanje podvojene kode. Razred `Example` razširja abstraktni razred `ExampleTemplate` in uporablja svojo logiko v metodah, ki še niso bile implementirane v abstraktnem razredu.

3.2.4 Vzorec *Opazovalec* (ang. Observer Pattern)

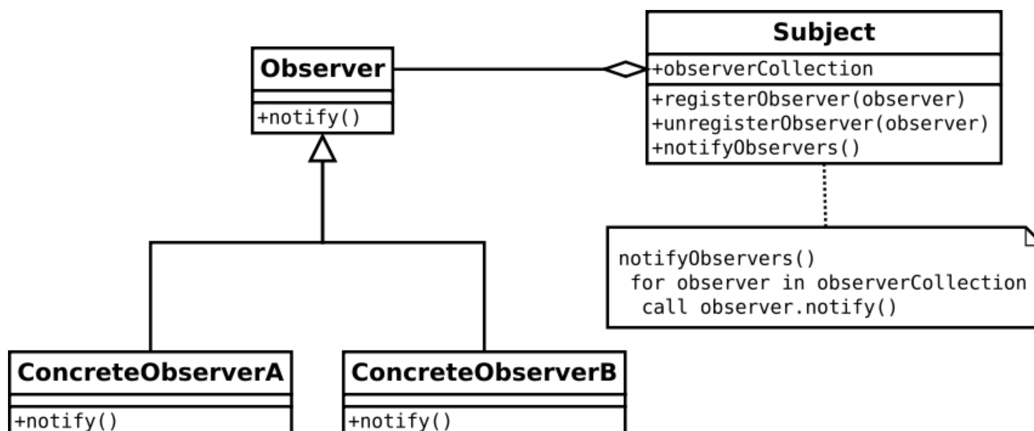
O objektnem programiranju ne moremo govoriti, ne da bi upoštevali stanje, v katerem se nahajajo posamezni objekti. Objekti komunicirajo med seboj, zato en objekt pogosto potrebuje informacijo o stanju kakšnega drugega objekta. Da bi imeli dober načrt moramo objekte ločiti med seboj in zmanjšati njihovo odvisnost, kolikor je to mogoče. Če je potrebno opazovati stanje nekega objekta s strani enega ali več drugih objektov, lahko uporabimo vzorec *Opazovalec* (ang. Observer)[18].

Vzorec *Opazovalec* je sestavljen iz objekta, ki se imenuje subjekt (ang. Subject) in vzdržuje seznam od njega odvisnih objektov, ki se imenujejo opazovalci (ang. Observers). Na ta način se opredeli odvisnost »ena proti mnogo« (ang. *one-to-many dependency*) in subjekt samodejno obvešča opazovalce o spremembah v svojem stanju. Na sliki 3.2 lahko vidimo, da subjekt `Subject` lahko registrira in odjavlja opazovalce. Subjekt hrani seznam opazovalcev in jih o svojem stanju obvesti preko metode `notifyObservers()`. Praktičen primer uporabe vzorca *Opazovalec* je prikazan v poglavju 4.2.

3.2.5 Vzorec *Edinec* (ang. Singleton Pattern)

Včasih je pomembno, da imamo samo en primerek določenega razreda. Na primer v sistemu je lahko samo en upravitelj okna (ang. *window manager*). Vzorec *Edinec* se uporablja za centralizirano upravljanje notranjih ali zunanjih virov in ostalimi objekti zagotovlja globalno točko dostopa do razreda, ki uporablja ta vzorec.

Vzorec *Edinec* je dokaj enostaven načrtovalen vzorec. Sestavljen je iz enega razreda, ki je odgovoren za svojo inicializacijo in preprečuje kreiranje več kot enega objekta. Zagotavlja globalno točko dostopa do objekta, s pomočjo katere lahko uporabljamo isto instanco objekta od koderkoli. Izvorna koda 3.12 predstavlja



Slika 3.2: Vzorec *Opazovalec* (povzeto iz [17]).

predlogo, kako naj bi se vzorec *Edinec* uporabljal. V 4.3 bomo predstavili praktični primer uporabe vzorca.

Izvorna koda 3.12: Predloga vzorca *Edinec*

```
class Singleton {
        private static Singleton instance;
        private Singleton()     { } //empty constructor

        public static Singleton getInstance() {
                if (instance == null)
                        instance = new Singleton();

                return instance;
        }

        public void doSomething() {
                ...
        }
}
```

3.2.6 Vzorec *Ničelni Objekt* (ang. Null Object Pattern)

V objektnem programiranju je ničelni objekt (ang. Null object) objekt z določenim nevtralnim vedenjem. Vzorec *Ničelni objekt* opisuje uporabo teh objektov in nji-

hovo vedenje (ali pomanjkanja le-tega) [16].

V večini objektno usmerjenih jezikov, kot je *Java*, so reference do določenega objekta lahko ničelne (ang. *null*). Te reference je treba preveriti, da bi zagotovili da niso ničelne, saj v primeru ničelnih referenc dostop do metode objekta ni možen [16].

Namesto, da bi z ničelno referenco posredovali odsotnost objekta (na primer neobstoječo stranko), lahko uporabljamo objekt razreda, ki implementira pričakovani vmesnik, vendar so njegove metode prazne oziroma nič ne naredijo. Prednost tega pristopa nad delovno privzetim vedenje ničelnega objekta je ta, da je objekt, ki uporablja vzorec *Ničelni objekt* zelo predvidljiv in nima stranskih učinkov (metoda v objektu ne naredi ničesar).

Na primer, metoda lahko vrača seznam datotek v določeni mapi in izvede neko akcijo za vsako datoteko. V primeru prazne mape metoda lahko sproži izjemo ali vrne ničelno referenco. Koda, ki pričakuje seznam datotek mora najprej preveriti, če seznam vsebuje kakšno datoteko. Preverjanje lahko zaplete načrtovanje.

Z uporabo vzorca *Ničelni objekt* ni potrebno preveriti, če ima vrnjena vrednost ničelno referenco. Klicana funkcija gre lahko normalno čez seznam, ne da bi kaj naredila.

V izvorni kodi 3.13 je prikazan primer možne uporabe vzorca *Ničelni objekt*, kjer razred `NullAnimal` predstavlja ničelni objekt. Če ne uporabljamo vzorca, bomo morali napisati dodatno kodo, ki bo lovila izjeme zaradi ničelnih referenc v primeru, ko je objekt `myAnimal` ničelni in nad njim pokličemo metodo `myAnimal.makeSound()`. Z uporabo vzorca *Ničelni objekt* ni potrebe po preverjanje ničelnih referenc in lovenje izjem.

Izvorno kodo smo povzeli iz [16].

Izvorna koda 3.13: Uporaba vzorca *Ničelni objekt*

```
public interface Animal {
        public void makeSound();
}

public class Dog implements Animal {
        public void makeSound()
                System.out.println("woof!");
}

public class NullAnimal implements Animal {
```

```
        public void makeSound() { }
}
```

Praktični primer uporabe vzorca v programu *WAVCutter* bomo predstavili v poglavju 4.4.

3.2.7 Vzorec *Tovarna* (ang. Factory Pattern)

Vzorec *Tovarna* je vzorec, ki uporablja metode za kreiranje objekta in pri tem ne določa natančno razreda objekta, ki bo ustvarjen. Kreiranje ustreznega objekta se izvede s klicanjem tovarniške metode, namesto s klicanjem konstruktorja. Tovarniška metoda je določena v vmesniku in je implementirana v podrazredu ali pa je implementirana v osnovnem razredu in po potrebi redefinirana v podrazredih. Namen tega vzorca je definirati vmesnik za ustvarjanje objekta, medtem ko razredi, ki implementirajo vmesnik določijo tip objekta, ki ga bodo ustvarili [8].

Ustvarjanje objekta pogosto zahteva kompleksne procese, ki jih ni primerno vključiti v objekt, ki ga kreiramo. Kreiranje objekta lahko povzroči podvojeno kodo, lahko zahteva informacije, ki niso dostopne ustvarjenim objektom ali pa ne zagotavlja zadostne stopnje abstrakcije. Vzorec *Tovarna* rešuje te probleme tako, da definira ločeno metodo za kreiranje objektov in s tem omogoča podrazredom, da to metodo redefinirajo in določajo tip objekta, ki ga bodo ustvarili.

V nadaljevanju bomo prikazali majhen primer, podan v [8], ki prikazuje praktično uporabo tega vzorca. Primer je naslednju:

≫Igra Labirint se lahko igra na dva načina, eden je z regularnimi sobami, ki so povezane samo s sosednjimi prostori, drugi način je z magičnimi sobami, ki omogočajo igralcem, da se naključno prevažajo od ene sobe v drugo.≪

V izvorni kodi 3.14 je prikazan način, kako se lahko implementira igro *Labirint* z regularnimi sobami. Če hočemo implementirati igro *Labirint* z magičnimi sobami, zadostuje, da redefiniramo samo metodo `makeRoom()`. Primer je prikazan v izvorni kodi 3.15.

Izvorna koda 3.14: Igra *Labirint* z regularnimi sobami

```
public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
```

```
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```

Izvorna koda 3.15: Igra *Labirint* z magičnimi sobami

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```

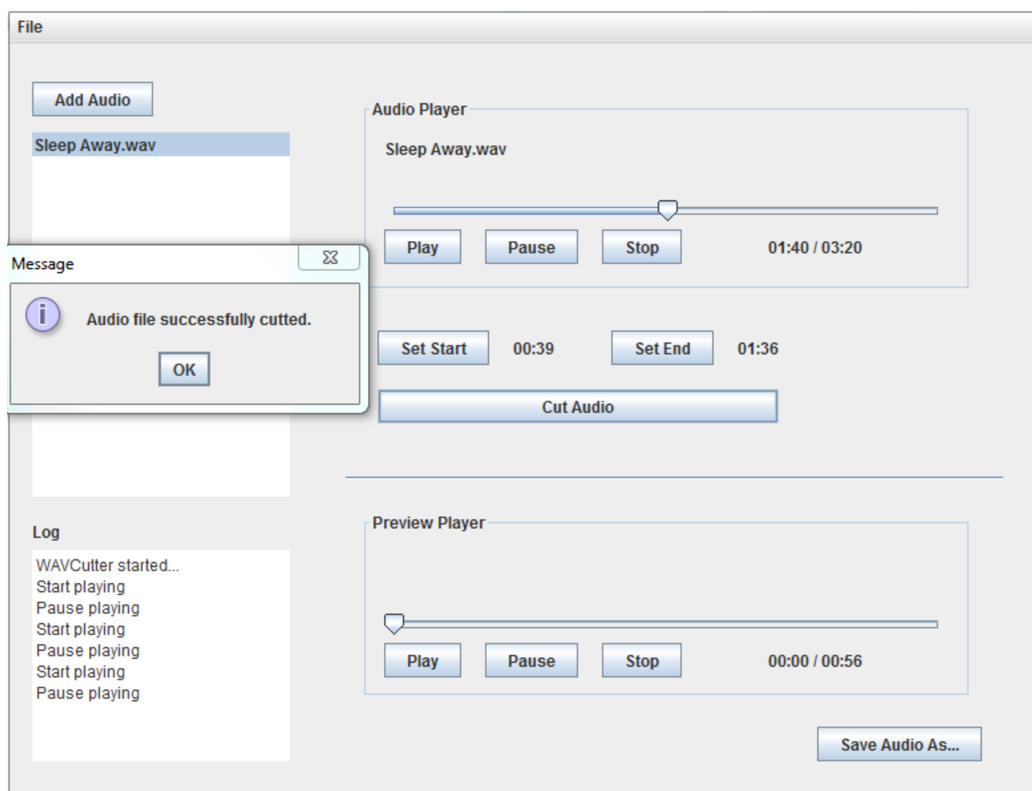
Poglavje 4

Uporaba vzorcev pri razvoju programa *WAVCutter*

WAVCutter je preprost program, ki uporabnikom omogoča predvajanje in rezanje zvočnih posnetkov. Uporabnik lahko na enostaven in lahek način ustvari melodije za zvonjenje mobilnih naprav in določi želeno dolžino zvočnega posnetka za video posnetke, filme in različne predstavitve. Po rezanju avdio podatka ohranja odlično kakovost zvoka. Uporabniški vmesnik programa je enostaven, z vsemi funkcionalnostmi, organiziranimi na enem zaslonu.

Izvorna koda programa *WAVCutter* je dostopna na [12].

Kako program deluje? Na sliki 4.1 je prikazan grafični uporabniški vmesnik programa. Zvočne datoteke, ki jih hočemo predvajati in rezati, dodamo v seznam zvočnih datotek s klikom na gumb »Add Audio«. Z dvojnim klikom na želeno datoteko naložimo avdio podatke v predvajalnik »Audio Player«. Predvajalnik ima tri osnovne funkcije: predvajaj, prekini in ustavi predvajanje. Z gumboma »Set Start« in »Set End« lahko nastavimo začetek in konec, kjer naj se zvočna datoteka prereže. S pritiskom na gumb »Cut Audio« se zvočna datoteka prereže in samodejno naloži v predogledni predvajalnik »Preview Player«, ki ima iste lastnosti kot navadni predvajalnik »Audio Player«. Program vsebuje tudi dnevnik dogodkov, ki beleži osnovne dogodke v sistemu, kot so zagon progama, začetek predvajanja, konec predvajanja itd.



Slika 4.1: Program *WAVCutter*

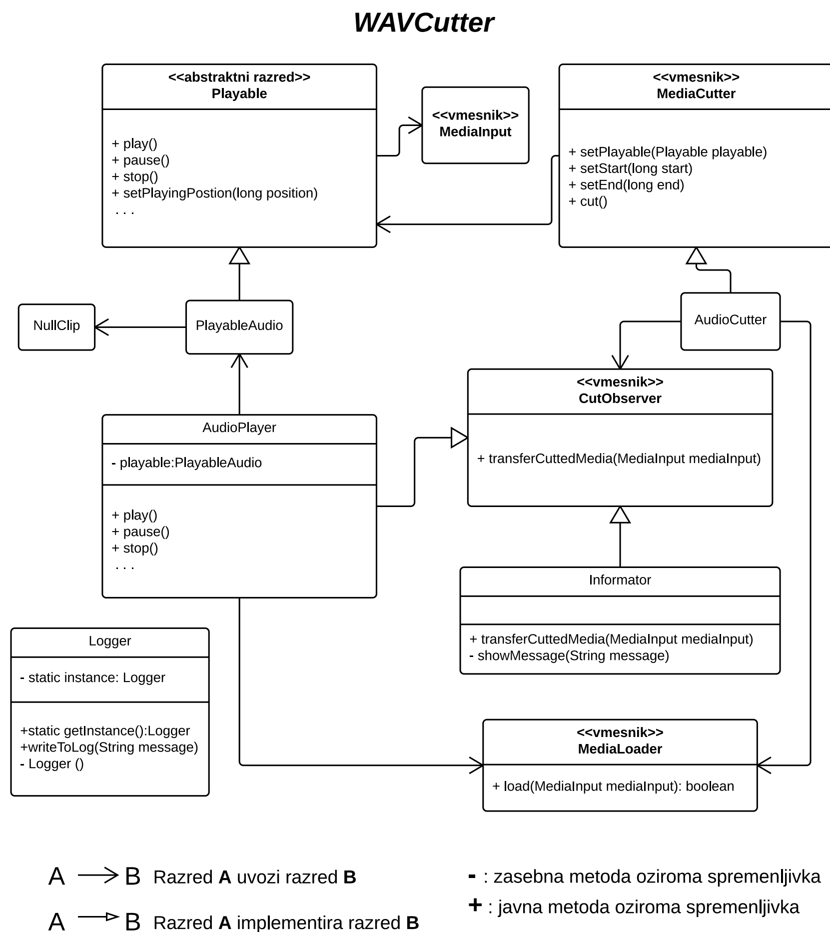
Uporabljeni razredi v programu *WAVCutter* Pri razvoju programa *WAVCutter* smo uporabili več razredov, ki jih bomo našteli v nadaljevanju. Odvisnosti med razredi so prikazani na sliki 4.2.

- `Playable` je abstraktni razred in vsebuje funkcije, ki jih potrebuje vsak predvajalnik (npr. ukaz predvajaj, ukaz ustavi, ukaz prekini, itd.). Razred lahko razširjamo pri razvoju različnih vrst predvajalnikov (npr. avdio predvajalnik, video predvajalnik, itd.).
- `PlayableAudio` razširja abstraktni razred `Playable` in implementira logiko za avdio predvajanje.
- `AudioPlayer` je razred, ki zgradi predvajalnik z grafičnim vmesnikom. `AudioPlayer` posreduje ukaze uporabnika v razred `PlayableAudio`.
- `MediaLoader` je vmesnik z metodo `load(MediaInput mediaInput)`. Ta vmesnik implementirajo razredi, ki lahko predvajajo objekte tipa `MediaInput`.
- `MediaCutter` je vmesnik, ki vsebuje metode za rezanje medijskih podatkov, kot so `setStart(long start)`, `setEnd(long end)` in `cut()`.
- `NullClip` predstavlja razred, ki uporablja vzorec *Ničelni objekt* in razširja razred `Clip`.
- `Logger` je razred, ki uporabnika obvesti o pomembnih dogodkih, ki se zgodijo v programu *WAVCutter*. Kreirali smo ga s pomočjo vzorca *Edinec*.
- `CutObserver` je vmesnik, ki skrbi za prenos prerezanega medijskega podatka.
- `Informator` implementira vmesnik `CutObserver` in na prijazen način obvesti uporabnika o uspešnosti rezanja medijskega podatka.

V nadaljevanju bomo prikazali uporabo vzorcev pri razvoju programa *WAVCutter*.

4.1 Uporaba vzorca *Šablonska metoda*

Naš program hočemo zgraditi na tak način, da je fleksibilen in da se lahko nadgradi v kakšen drug program (recimo v video predvajalnik in video skrajševalnik).



Slika 4.2: Odvisnosti med razredi v programu *WAVCutter*

Podvojeni kodi, ki se lahko pojavi ob morebitni nadgradnji, smo se hoteli izogniti z uporabo vzorca *Šablonska metoda*. Zato smo pri razvoju programa *WAVCutter* v abstraktnem razredu `Playable` uporabili vzorec *Šablonska metoda*. Abstraktni razred `Playable` sestavljajo osnovne funkcije, ki jih potrebuje vsak predvajalnik (ukaz predvajaj, ukaz ustavi, ukaz prekini, itd.). Predvajalniki so lahko različnih vrst (avdio, video, avdio-video itd.), zato je možnih veliko podrazredov, ki lahko razširjajo razred `Playable`. Vsak podrazred, implementira svoje metode za realizacijo ukazov predvajaj, ustavi, prekini itd., medtem ko je postopek predvajanja za vse podrazrede enak. Zato je smiselno ta postopek sprogramirati samo enkrat v razredu `Playable`.

Kot smo opisali v poglavju 3.2.3, vzorec *Šablonska metoda* predstavlja skelet algoritma v določeni metodi, ki preprečuje podvajanje izvorne kode v podrazredih. Metoda, ki implementira vzorec, se imenuje *šablonska metoda*.

Z uporabo vzorca *Šablonska metoda* smo v abstraktni razred `Playable` dodali šablonsko metodo `play(long position)`. Metoda `play(long position)` s pomočjo abstraktnih metod `getStart()` in `getEnd()` najprej preveri, če medijski podatek vsebuje želeno predvajalno pozicijo. Če je želena pozicija znotraj medijskega podatka, metoda `play(long position)` zaustavi predvajanje s pomočjo abstraktne metode `stop()`, nastavi prej definirano pozicijo s pomočjo abstraktne metode `setPlayingPosition(long position)` in na koncu, s pomočjo abstraktne metode `startPlaying()` začne predvajati medijske datoteke od že nastavljene pozicije naprej. Implementacija abstraktne metode je prepuščena podrazredom, ki bodo razširjale abstraktnega razreda `Playable`.

V izvorni kodi 4.1 je prikazana uporaba vzorca *Šablonska metoda* v abstraktnem razredu `Playable`. Izvorna koda 4.2 pa prikazuje implementacijo abstraktnih metod razreda `Playable` v razredu `PlayableAudio`, ki omogoča predvajanje avdio posnetkov.

Izvorna koda 4.1: Uporaba vzorca *Šablonska metoda* v razredu `Playable`

```
public abstract class Playable {

        abstract void startPlaying();
```

```
        abstract void stop();
        abstract void setPlayingPostion(long position);
        abstract long getStart();
        abstract long getEnd();

        public void play(long position) {

                // Check if position is before start of the media
                if (jumpPosition < getStart())
                        jumpPosition = getStart();

                // Check if position is after end of the media
                else if (jumpPosition > getEnd())
                        jumpPosition = getEnd();

                // Stop playing
                stop();

                // Set position
                setPlayingPostion(position);

                // Play again at new position
                startPlaying();
        }

        public long getMediaLength() {
                return getEnd() - getStart();
        }
}
```

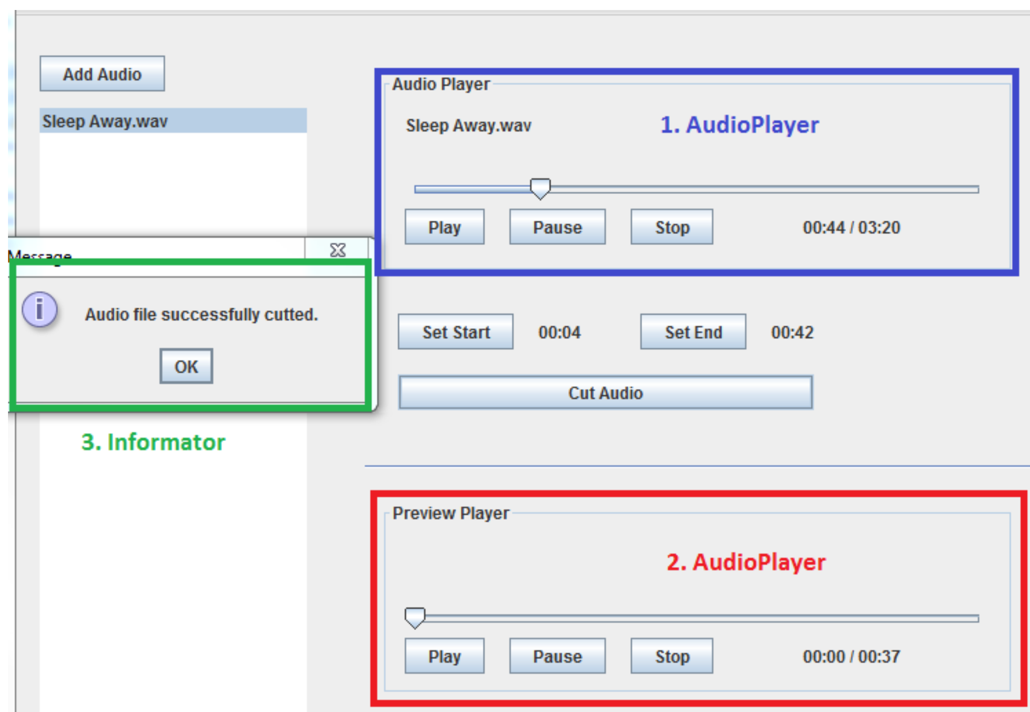
Izvorna koda 4.2: Razred `PlayableAudio` razširja abstraktni razred `Playable`

```
public class PlayableAudio extends Playable {
        private Clip clip;

        public void startPlaying() {
                if(clip == null) return;

                // Check if the file end playing
                if(getEnd() == getPlayingPosition())
                        stop();

                // Start streaming
                clip.start();
        }
```

```
	public void stop() {
		if(clip == null) return ;

		// Set position to 0
		clip.setMicrosecondPosition(0);
		clip.stop();
	}

	public void setPlayingPostion(long position) {
		if(clip == null) return ;

		clip.setMicrosecondPosition(position);
	}

	public long getStart() {
		return 0;
	}

	public long getEnd() {
		if(clip == null) return 0;

		return clip.getMicrosecondLength();
	}

	...
}
```

4.2 Uporaba vzorca *Opazovalec*

V poglavju 3.2.4 smo omenili, da objekti komunicirajo med seboj in včasih potrebujejo informacijo o stanjih drugih objektov. Pri razvoju programa *WAVCutter* smo opazili, da nekateri objekti potrebujejo informacijo o tem, kdaj bo določen objekt spremenil svoje stanje. Objekta razredov `AudioPlayer` in `Informator` potrebujeta informacijo o rezanju medijskega podatka, ki se zgodi v objektu razreda `AudioCutter`. Objekt razreda `AudioPlayer` potrebuje prerezan medijski podatek, objekt razreda `Informator` pa samo podatek, kdaj se je rezanje zgodilo. Na sliki 4.3 vidimo dva objekta razreda `AudioPlayer`. Objekt, oštevilčen s številko 1, je avdio predvajalnik in predvaja izvirne avdio datoteke (v nadaljevanju `AudioPlayer1`). Objekt s številko 2 je prav tako avdio predvajalnik, vendar predvaja samo prerezane avdio podatke (v nadaljevanju `AudioPlayer2`). Skle-



Slika 4.3: Program *WAVCutter*

pamo lahko, da samo objekta razreda `Informator` in `AudioPlayer2` potrebujeta informacijo o rezanju avdio podatka. Kako potem dosežemo, da `Informator` in `AudioPlayer2` dobita informacijo o rezanju, `AudioPlayer1` pa te informacije ne dobi?

Naš cilj smo dosegli z uporabo vzorca *Opazovalec*. Subjekt v vzorcu je postal razred `AudioCutter`, opazovalca pa razreda `AudioPlayer` in `Informator`, ki implementirata vmesnika `CutObserver`. `AudioCutter` bo o rezanju avdijskega podatka obveščal samo registrirane objekte (v našem primeru `Informator` in `AvdioPlayer2`). V izvorni kodi 4.3 je podan del izvorne kode programa *WAVCutter*, kjer smo uporabili vzorec *Opazovalec*.

Izvorna koda 4.3: Uporaba vzorca *Opazovalec* v programu *WAVCutter*

```
class AudioCutter extends JPanel implements ActionListener, MediaCutter {
        private MediaInputAudioInputStream cuttedInputStream;
        private ArrayList<CutObserver> cutObservers;
        ...
```

```
        public void cut() {
                ...
                notifyCutObservers(); // Notify observers
        }

        public void registerCutObserver(CutObserver observer) {
                cutObservers.add(observer);
        }

        public void unregisterCutObserver(CutObserver observer) {
                cutObservers.remove(observer);
        }

        private void notifyCutObservers () {
                for(CutObserver observer : cutObservers)
                        observer.transferCuttedMedia(cuttedInputStream);

                cuttedInputStream = null;
        }
}

interface CutObserver {
        void transferCuttedMedia(MediaInput mediaInput);
}

class AudioPlayer extends JPanel implements ActionListener,
                MouseListener, MediaLoader, CutObserver {

        public void transferCuttedMedia(MediaInput mediaInput) {
                if(mediaInput == null) return;

                load(mediaInput);
        }

        public boolean load(MediaInput mediaInput) { ... }
}

class Informator implements CutObserver {
        public void transferCuttedMedia(MediaInput mediaInput) {
                if(mediaInput == null) return;

                showMessage("Audio file successfully cutted.");
        }

        private void showMessage(String message) {
                JOptionPane.showMessageDialog(null, message);
        }
```

```
}
```

Izvorna koda 4.4: Registracija opazovalcev v objektu razreda `AudioCutter`

```
class MainPanel extends JPanel {
        private AudioCutter audioCutterPanel;
        private AudioPlayer cuttedSegmentAudioPlayer;
        private Informator informator;

        private void initializeComponents() {
                ...
                audioCutterPanel.registerCutObserver(
                    cuttedSegmentAudioPlayer);
                audioCutterPanel.registerCutObserver(informator);
        }
}
```

Objekti, za katere hočemo, da prejemajo informacije o rezanju, registriramo v objektu razreda `AudioCutter` preko metode `registerCutObserver(CutObserver observer)`. Če registrirani objekti nimajo več potrebe po informaciji o rezanju, jih lahko odjavimo s pomočjo metode `unregisterCutObserver(CutObserver observer)`. V izvorni kodi 4.4 je prikazan način registracije opazovalcev. `AudioCutter` preko metode `notifyCutObservers()` obvešča `Informator` in `AudioPlayer2` o rezanju avdio podatka. Po sprejeti informaciji `Informator` obvesti uporabnika, da je rezanje bilo uspešno, `AudioPlayer2` pa naloži prerezan podatek v svoj predvajalnik in pripravi okolje za predvajanje naloženega podatka.

4.3 Uporaba vzorca *Edinec*

Program izvaja nekoliko pomembnih funkcij in zato menimo, da mora biti uporabnik obveščen o tem, kdaj se te funkcije začnejo izvajati. Za ta namen smo se odločili, da naš program nadgradimo z dnevnikom dogodkov (ang. *logger*). Vsak pomemben dogodek (zagon programa, začetek predvajanj, rezanje avdio podatke itd.), ki se zgodi v programu, želimo zapisati v dnevnik in vsebino dnevnika na prijazen način predstaviti uporabniku. Ker se posamezni dogodki dogajajo v različnih modulih v našem programu in ker hočemo vse dogodke zapisati v isti dnevnik, moramo poskrbeti, da obstaja en sam dnevnik dogodkov, v katerega se lahko dogodki beležijo od koderkoli. Da bi dosegli želen cilj, smo se odločili uporabiti vzorec

Edinec v naši izvorni kodi.

V poglavju 3.2.5 smo omenili, da vzorec *Edinec* zagotovi globalno točko dostopa do razreda, ki uporablja ta vzorec. Za naš program smo ustvarili razred `Logger` (del izvorne kode razreda je prikazan v izvorni kodi 4.5) in v njem uporabili vzorec *Edinec*. Konstruktor razreda smo zaščitili pred neželenim klicanjem iz drugih objektov s tem, da smo ga označili kot zasebni (ang. *private constructor*). Razred vsebuje zasebno statično instanco samega sebe, ki ima pred prvo inicializacijo vrednost `null`. Do objekta dostopamo s pomočjo metode `static Logger getInstance()`. Če do instance dostopamo prvič, metoda `static Logger getInstance()` najprej inicializira instance razreda `Logger` in jo vrne nazaj razredom, ki jo zahteval. Vsi objekti dobijo isto instanco razreda `Logger`. V izvorni kodi 4.5 je prikazan del izvorne kode programa *WAVCutter*, kjer smo uporabili vzorec *Edinec*.

Izvorna koda 4.5: Uporaba vzorca *Edinec* v programu *WAVCutter*

```
class Logger extends JTextPane {
        private static Logger instance;

        private Logger () { }

        public static Logger getInstance() {
                if(instance == null)
                        instance = new Logger();

                return instance;
        }

        public void writeToLog(String message) {
                String oldMessage = getText();
                setText(oldMessage + message + "\n");
        }
}

class AudioPlayer extends JPanel ... {
        ...

        public void play() {
                ...
                Logger.getInstance().writeToLog("Start playing");
        }

        public void stop() {
```

```
                ...
                Logger.getInstance().writeToLog("Stop playing");
        }
}

class AudioCutter extends JPanel ... {
        ...
        public void cut() {
                ...
                Logger.getInstance().writeToLog("Audio data cutted");
        }
}
```

4.4 Uporaba vzorca *Ničelni Objekt*

Če bolj podrobno pogledamo v Izvorna koda 4.2 lahko opazimo, da vsaka metoda najprej preverja, če ima objekt tipa `Clip` ničelno referenco (ang. *null object*). V tem primeru se metoda preneha izvajati. Da bi se izognili preverjanju v vsaki metodi, smo se odločili v naši kodi uporabiti vzorec *Ničelni objekt*. V poglavju 3.2.6 smo omenili, da odsotnost nekega objekta lahko nadomestimo z uporabo tega vzorca. Objekt, ki je zgrajen po vzorcu *Ničelni objekt* je zelo predvidljiv in nima stranskih učinkov, ker metode v objektu ne počnejo ničesar.

Za naš namen smo kreirali nov razred `NullClip`, ki v svoji kodi implementira vmesnik `Clip`. Del njegove implementacije smo prikazali v izvorni kodi 4.6. Metode v novem razredu smo ustvarili tako, da ne počnejo nič ali pa ob klicu vračajo privzeto vrednost. S tem se izognemo dodatnemu preverjanju ničelne reference ob začetku vsake funkcije v razredu `PlayableAudio`, podanem v izvorni kodi 4.2.

Izvorna koda 4.6: Razred *NullClip*

```
public class NullClip implements Clip {
        ...

        public long getMicrosecondPosition() {
                return 0;
        }

        public void start() { }

        public void stop() { }
```

```
        public void close() { }

        public void open() throws LineUnavailableException { }

        public long getMicrosecondLength() {
                return 0;
        }

        public void open(AudioInputStream stream) throws
            LineUnavailableException, IOException { }

        public void setMicrosecondPosition(long microseconds) { }
}
```

S pomočjo vzorca *Ničelni objekt* smo kodo v izvorni kodi 4.2 preoblikovali v izvorno kodo 4.7. Lahko opazimo, da smo se na enostaven način izognili preverjanju ničelne reference v vsaki metodi in tako poenostavili načrt modula.

Izvorna koda 4.7: Razred `PlayableAudio` in uporabo vzorca *Ničelni objekt*

```
public class PlayableAudio extends Playable {
        private Clip clip = new NullClip();
        ...

        public void play() {
                // Check if the file end playing
                if(getEnd() == getPlayingPosition())
                        stop();

                // Start streaming
                clip.start();
        }

        public void stop() {
                // Set position to 0
                clip.setMicrosecondPosition(0);
                clip.stop();
        }

        public void pause() {
                clip.stop();
        }

        long getEnd() {
                return clip.getMicrosecondLength();
        }
}
```

Sklepne ugotovitve

Cilj diplomske naloge je bil proučiti načrtovalske vzorce in njihovo uporabo pri objektno usmerjenem razvoju programske opreme. Kot rezultat naloge je predstavljen program *WAVCutter*, kjer smo uporabo vzorcev preizkusili pri samem razvoju. Program smo zgradili s pomočjo objektno usmerjenega jezika *Java*. Uporaba vzorcev nam je omogočila zgraditi zanesljiv program, ki je razširljiv in omogoča preprosto spreminjanje funkcionalnosti. Z minimalnim naporom lahko program pretvorimo v video predvajalnik, ki bi imel možnost predvajanja in rezanja video posnetkov, ali pa ga razširimo v avdio-video predvajalnik z osnovnimi funkcionalnostmi. Nadgradnje lahko izvršimo samo z dodajanjem novih modulov, brez posegov v obstoječih.

Uporaba vzorcev nas je naučila kako preoblikovati »gnijočo kodo« v strukturo, ki jo je lažje razumeti in vzdrževati.

Literatura

[1] Viljan Mahnič *Specifikacija zahtev v obliki uporabniških zgodb*, prosojnice pri predmetu *Tehnologija programske opreme* v š.l. 2013/2014

[2] Viljan Mahnič *Ekstremno programiranje (XP)*, prosojnice pri predmetu *Tehnologija programske opreme* v š.l. 2013/2014

[3] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2003.

[4] *Command pattern*. Dostopno na: http://en.wikipedia.org/wiki/Command_pattern

[5] *Dependency Inversion Principle*. Dostopno na: http://www.oodesign.com/dependency-inversion-principle.html

[6] *Facade Design Pattern*. Dostopno na: http://sourcemaking.com/design_patterns/facade

[7] *Facade pattern*. Dostopno na: http://en.wikipedia.org/wiki/Facade_pattern

[8] *Factory method pattern*. Dostopno na: http://en.wikipedia.org/wiki/Factory_method_pattern

[9] *Interface segregation principle*, dostopno na: http://en.wikipedia.org/wiki/Interface_segregation_principle

[10] *Interface Segregation Principle (ISP)*. Dostopno na:
http://www.oodesign.com/interface-segregation-principle.html

[11] *Iteration Planning*. Dostopno na:
http://www.extremeprogramming.org/rules/iterationplanning.html

[12] *Izvorna koda programa WAVCutter*. Dostopno na:
https://github.com/dkostadinovski/AudioCutter/tree/master/SoundClip

[13] *Liskov's Substitution Principle(LSP)*. Dostopno na:
http://www.oodesign.com/liskov-s-substitution-principle.html

[14] *List of unit testing frameworks*. Dostopno na:
http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

[15] *Manifest agilnega razvoja programske opreme*. Dostopno na:
http://agilemanifesto.org/iso/sl/

[16] *Null Object pattern*. Dostopno na:
http://en.wikipedia.org/wiki/Null_Object_pattern

[17] *Observer pattern*. Dostopno na:
http://en.wikipedia.org/wiki/Observer_pattern

[18] *Observer Pattern*. Dostopno na:
http://www.oodesign.com/observer-pattern.html

[19] *Open Close Principle*. Dostopno na:
http://www.oodesign.com/open-close-principle.html

[20] *Principi v ozadju agilnega manifesta*. Dostopno na:
http://agilemanifesto.org/iso/sl/principles.html

[21] *Single Responsibility Principle*. Dostopno na:
http://www.oodesign.com/single-responsibility-principle.html