SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Krešimir Maržić

PRILAGODBA METODE EKSTREMNOG PROGRAMIRANJA ZA PROJEKT RAZVOJA JAVNE ELEKTRONIČKE USLUGE

MAGISTARSKI RAD

Zagreb, 2005.

Magistarski rad je izrađen u *Centru za komunikacijska rješenja, usluge, logistiku i e-sustave* u kompaniji *Ericsson Nikola Tesla d.d.* u Zagrebu i na *Zavodu za telekomunikacije Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu*.

Mentor: Doc.dr.sc. Željka Car, dipl.ing.
Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu

Magistarski rad ima: 132 stranice.

Rad br. _____

Povjerenstvo za ocjenu rada u sastavu:

1. Prof.dr.sc. Ignac Lovrek - predsjednik,
   Fakultet Elektrotehnike i računarstva, Sveučilište u Zagrebu
2. Doc.dr.sc. Željka Car - mentor,
   Fakultet Elektrotehnike i računarstva, Sveučilište u Zagrebu
3. Prof.dr.sc. Ivica Crnković,
   Mälardalen University, Department of Computer Science and Electronics, Västerås, Sweden

Povjerenstvo za obranu rada u sastavu:

1. Prof.dr.sc. Ignac Lovrek - predsjednik,
   Fakultet Elektrotehnike i računarstva, Sveučilište u Zagrebu
2. Doc.dr.sc. Željka Car - mentor,
   Fakultet Elektrotehnike i računarstva, Sveučilište u Zagrebu
3. Prof.dr.sc. Ivica Crnković,
   Mälardalen University, Department of Computer Science and Electronics, Västerås, Sweden
4. Prof.dr.sc. Marijan Kunštić - zamjenik,
   Fakultet Elektrotehnike i računarstva, Sveučilište u Zagrebu

Datum obrane: 06. travnja 2005.

Zahvaljujem svima koji su mi na bilo koji način pomogli da ovaj rad doživi svjetlo dana.

Zahvaljujem kompaniji, *Ericsson Nikola Tesla d.d.*, čiji sam stipendist od proljeća 2003. godine, te *doc.dr.sc. Željki Car, dipl.ing*, voditeljici i mentorici.
Veliko hvala i kolegama s posla s kojima sam sudjelovao na *HT Telex over IP* i *PZZ* projektima, s kojima sam radio i od kojih sam puno naučio. Hvala i onima koji su mi izravno pomogli u realizaciji ovog rada te ih izdvajam abecednim redom: *dipl.ing. Tomislav Belobrajdić*, *dipl.ing. Dalibor Brnas*, *dipl.ing. Ivan Crkvenac*, *doc.dr.sc. Saša Dešić*, *dipl.ing. Miljenko Kalenik*, *dipl.ing. Hrvoje Lučić*, *mr.sc. Karlo Šmid*.
Zahvaljujem se i svima onima koji su mi pomagali u životu i radu. Molim da mi oproste svi oni koje zbog ograničenog prostora ali i vlastite zaboravljivosti nisam poimence nabrojao i zahvalio im.

Na kraju, posebno zahvaljujem kolegi *dr.sc. Hrvoju Sertiću*, bez kojeg svega ovoga najvjerojatnije ne bi bilo.

Sadržaj

Popis slika **viii**

Korištene kratice **ix**

1 Uvod **1**

2 Pregled metodologija razvoja softvera **4**
2.1 Uvod u metodologije razvoja softvera 4
2.2 Povijesni aspekti nastanka i razvoja metodologija 5
2.3 Primjeri metodologija razvoja softvera 6
2.3.1 *Top-down* i *Bottom-up* model 6
2.3.2 Vodopadni model 7
2.3.3 Spiralni model 9
2.3.4 Model kaosa 9
2.3.5 Model protipa 10
2.3.6 Iterativni i inkrementalni razvoj 11
2.3.7 *Rational Unified Process* 13

3 Agilne metode (engl. *Agile Methods*) razvoja softvera **14**
3.1 Povijesni aspekti nastanka agilnih metoda 14
3.2 Svojstva agilnih metoda 15
3.2.1 Pregled i definicije 15
3.2.2 Karakteristike 17
3.3 Primjeri agilnih metoda 18

4 Razvojna metodologija ekstremnog programiranja **19**
4.1 Uvod u ekstremno programiranje 19
4.2 Područja primjene metodologije ekstremnog programiranja 21

5 Postupci metodologije ekstremnog programiranja **24**
5.1 Planiranje razvoja softvera u ekstremnom programiranju 24
5.1.1 Korisničke priče 25
5.1.2 Planiranje isporuke 26

5.1.2.1 Kvantificiranje projekta s četiri varijable 27
5.1.3 Male isporuke 28
5.1.4 Brzina projekta 28
5.1.5 Iterativni razvoj 29
5.1.6 Planiranje iteracija 30
5.1.7 Kretanje ljudi 31
5.1.8 Dnevni stojeći sastanci 32
5.1.9 Popravak XP-a 33
5.2 Dizajn softvera u ekstremnom programiranju 33
5.2.1 Jednostavnost 34
5.2.2 Metafora sustava 34
5.2.3 CRC karte 34
5.2.4 Rješenje tehnoloških probi 35
5.2.5 Izbjegavanje dodavanja funkcionalnosti 35
5.2.6 Refaktoriranje 36
5.3 Kodiranje softvera u ekstremnom programiranju 37
5.3.1 Stalna prisutnost naručitelja u razvoju 37
5.3.2 Standardi pisanja kôda 38
5.3.3 Kodiranje testova za jedinice programa prije implementacije funkcionalnosti 39
5.3.4 Programiranje u pâru 40
5.3.5 Kontinuirana integracija kôda 41
5.3.5.1 Česta integracija 41
5.3.6 Zajedničko vlasništvo nad kôdom 42
5.3.7 Optimizacija kôda na kraju 43
5.3.8 40-satni radni tjedan 43
5.4 Testiranje softvera u ekstremnom programiranju 43
5.4.1 Testiranje jedinice programa 44
5.4.2 Pronalaženje neispravnosti 45
5.4.3 Često izvršavanje testova prihvaćenosti 46
5.5 Životni ciklus XP projekta 46
5.5.1 Istraživačka faza 47
5.5.2 Faza planiranja 48
5.5.3 Faza od iteracija do isporuke 48
5.5.4 Faza isporuke 48
5.5.5 Faza održavanja 49
5.5.6 Faza završetka 49
5.5.7 Rizici u životnom ciklusu XP projekta 49
5.6 XP projektne uloge 50
5.6.1 Programer 51
5.6.2 Naručitelj 51

5.6.3 Tester . . . 51
5.6.4 Tragač . . . 52
5.6.5 Trener . . . 52
5.6.6 Savjetnik . . . 53
5.6.7 Menadžer . . . 53
5.7 Slijed dnevnih aktivnosti u XP-u . . . 53

6 Primjene razvojnih postupaka metode ekstremnog programiranja 55
6.1 Opis projekta razvoja javne elektroničke usluge . . . 55
6.2 Primjene razvojnih postupaka metode ekstremnog programiranja . . . 59
6.2.1 Programiranje u pâru . . . 60
6.2.2 Tehnika refaktoriranja . . . 60
6.2.2.1 Duplicirani kôd . . . 61
6.2.2.2 Dugačka metoda . . . 64
6.2.2.3 Golema klasa . . . 68
6.2.2.4 Dugačka lista parametara . . . 68
6.2.3 Jedinično testiranje . . . 69
6.2.3.1 Implementacija testova . . . 70
6.2.3.2 *JUnit* okružje za testiranje . . . 70
6.2.3.3 Integracija *Rational TestManger*-a i *JUnit*-a . . . 73
6.2.3.4 *csUnit* okružje za testiranje . . . 74
6.2.3.5 Potreba za razvojem novog testnog okružja . . . 76
6.2.4 Male i česte isporuke . . . 78
6.3 Rezultati primjene postupaka metode ekstremnog programiranja na projekt razvoja javne elektroničke usluge . . . 79
6.3.1 Period promatranja projekta . . . 79
6.3.2 Struktura tima . . . 80
6.3.2.1 Voditelj projekta . . . 81
6.3.2.2 Tehnički koordinator . . . 82
6.3.2.3 Programer . . . 82
6.3.2.4 Tester . . . 83
6.3.3 Karakteristika softvera . . . 84
6.3.4 Postupci XP-a . . . 84
6.3.4.1 Programiranje u pâru . . . 85
6.3.4.2 Tehnika refaktoriranja . . . 86
6.3.4.3 Jedinično testiranje . . . 87
6.3.4.4 Male i česte isporuke . . . 89
6.4 Osvrt na budući rad iz područja efikasne implementacije XP razvojne metodologije . . . 89

7 Zaključak 91

A Izvorni kôd **94**
A.1 BN.java 94
A.2 TC_BN.java 96
A.3 TM_BN.java 107
A.4 Telecom_Factory_TC.cs 108

Literatura **112**

Indeks **119**

Sažetak **119**
Ključne riječi 119

Summary **120**
Keywords 120

Životopis **121**

Popis slika

2.1 Vodopadni model (engl. *Waterfall model*) 8

3.1 Odnos agilnih metoda i ostalih pristupa razvoja softvera 16

5.1 Razvoj projekta u procesu ekstremnog programiranja 26
5.2 Iteracija u ekstremnom programiranju 30
5.3 Zajedničko vlasništvo kôda 32
5.4 Razvoj (engl. *Development*) u Ekstremnom programiranju 33
5.5 Životni ciklus XP procesa 47

6.1 *HalthCare Agent* 56
6.2 Područje zaustavljanja (engl. *PullUpField*) 62
6.3 Oblik predloška metode (engl. *Form Template Method*) 63
6.4 Izdvajanje klase (engl. *Extract Class*) 64
6.5 Uvođenje parametriziranog objekta (engl. *Introduce Parameter Object*) . 65
6.6 Zamjena metode s metodom objekta (engl. *Replace Method with Method Object*) 67
6.7 Izdvajanje potklase (engl. *Extract Subclass*) 68
6.8 *JUnit* testno okružje 71
6.9 *csUnit* testno okružje 76
6.10 *HCAgentTestTool* 77

Korištene kratice

ACM	*Association for Computing Machinery* Udruženje računalne opreme
ANSI	*American National Standards Institute* Američki nacionalni institut za standarde
ASD	*Adaptive Software Development* Prilagodljivi razvoj softvera
API	*Application Program Interface* Aplikacijsko programsko sučelje
ATM	*Asynchronous Transfer Mode* Asinkroni mod prijenosa
CCW	COM *Collable Wrapper*
COM	*Component Object Model* Komponentni objektni model
CMM	*Capability Maturity Model* Starosni model sposobnosti
CVS	*Concurrent Versions System* Repozitorij konkurentnih verzija
DMIM	*Domain Message Information Model* Informacijski model poruka
DSDM	*Dynamic Systems Development Method* Dinamička metoda razvoja sustava
FDD	*Feature Driven Development* Karakteristikama usmjeravan razvoj
FSF	*Free Software Foundation* Slobodna softverska fundacija
GPL	*General Public License* Opća javna licenca
GUI	*Graphical User Interface* Grafičko korisničko sučelje
HC	*Health Care* Zdravstvena njega
HL7	*Health Level Seven* Međunarodna organizacija koja se bavi standardiziranjem informatizacije zdravstva
IBM	*International Business Machines Corporation* kolokvijalno, *Big Blue*

ICT	*Information and Communications Technology* Informacijska i komunikacijska tehnologija
IDE	*Integrated Development Environment* Integrirano razvojno okružje
ISO	*International Organization for Standardization* Međunarodna organizacija za standardizaciju
OO	*Object - Oriented* Objektno - orijentirano
OSI	*Open System Interconnection* Otvoreni sustav međuspajanja
OSS	*Open Source Software Development* Razvoj softvera otvorenog kôda
PLEX	*Programming Language for EXchanges* Programski jezik za centrale
QA	*Quality Assurance* Osiguravanje kvalitete
RAD	*Rapid Application Development* Rapidni razvoj aplikacija
RIM	*Reference Information Model* Referentni informacijski model
RUP	*Rational Unified Process* *Rational* unificirani proces
SDK	*Software Development Kit* Alat za razvoj softvera
TDD	*Test Driven Development* Testiranjem usmjeravan razvoj
UML	*Unified Modelling Language* Unificirani jezik za modeliranje
XP	*eXtreme Programming* Ekstremno programiranje

Poglavlje 1

Uvod

U današnjem informacijskom dobu, u svijetu tehnike, računala i komunikacija softver je jedno od najvažnijih dostignuća ljudskog stvaralaštva. Softver je svakim danom sve složeniji zbog neprestanog razvoja sustava koji ga koriste. Kao posljedica te složenosti, raste i broj razvojnih timova koji unutar softverskih kuća rade na razvoju softvera.

Kao neizostavna činjenica nameće se problematika razvoja softvera koji će biti isporučen na vrijeme, sa što manje otkrivenih pogrešaka i sa što zadovoljnijim naručiteljima i kupcima. Ova problematika nameće, kako u prijašnjim godinama razvoja složenog softvera, a tako i u današnjem dobu, korištenje odgovarajućih procesa za razvoj softvera, nasuprot neorganiziranom i nesređenom razvoju.

Pod pojmom procesa razvoja softvera podrazumijeva se slijed akcija koje se izvode u postupku razvoja softvera. Proces razvoja softvera je svojevrsna poveznica ljudi koji sudjeluju u razvoju softvera, alata koji se koriste u razvoju te procedura.

Prema svome bitku, svi procesi razvoja softvera su općeniti do određene mjere. Na razvojnom timu ili kompaniji je da ga prilagodi svojim specifičnim potrebama kako bi ga mogla efikasno koristiti.

Pojam procesa se često odnosi na standardiziranu i dokumentiranu metodologiju koja je već prije korištena na sličnim projektima razvoja softvera ili koja se već koristi unutar neke organizacije ili kuće.

U minulim periodima razvijale su se različite metodologije koje su naglašavale različite pristupe u razvoju softvera. Opstale su i danas se koriste one koje su se sa svojim praksama pokazale dovoljno kvalitetnima i prilagodljivima da mogu odgovoriti na rastuće probleme u današnjem razvoju softvera. Mogu se izdvojiti tradicionalne, procesno-orijentirane metode tradicionalnih pristupa u razvoju softvera te agilne, nove i manje opsežne metode usredotočene na male jedinice posla i s naglaskom na vrijednostima i principima umjesto na procesu.

Samo jedna metodologija razvoja softvera ne može djelovati na čitavom spektru različitih projekata. Sve metodologije i nisu primjenjive u svim projektima. Usprkos tomu, projektni menadžment treba identificirati specifičnu prirodu projekta te odabrati najprikladniju razvojnu metodologiju. Zato i postoji potreba i za agilnim metodologijama i za procesno-orijentiranim metodologijama razvoja softvera [1].

Ekstremno programiranje (engl. *Extreme Programming*, XP) je pristup, odnosno

metoda razvoja softvera koja adresira probleme od najranije faze prikupljanja zahtjeva (engl. *requirements*) do isporuke (engl. *delivery*).
Ekstremno programiranje je, prije svega, programerska disciplina sa svojom jezgrenom inovacijom implementacije testova prije sâmog kodiranja programskog sustava (aplikacije) koja tim pristupom iz temelja mijenja postupak razvoja softvera. Zatim, XP je i timska disciplina koja uključuje prakse koje pomažu izgradnju tima visokih performansi i solidnog znanja. Također, XP je i disciplina za rad s kupcima jer sadrži specifičan proces planiranja i dnevne aktivnosti u procesu razvoja softvera.

Ovaj rad bavi se problematikom uvođenja prikladnih praksi XP metodologije razvoja softvera u tekuće projekte velike organizacije (ili određene organizacijske jedinice) koja se bavi razvojem softvera.

Magistarski rad ima za cilj pokazati glavne prakse i temeljne karakteristike XP metodologije razvoja softvera. Istražuje se primjena određenih praksi XP-a u projektima s različitih stajališta osoba uključenih u realizaciju razvojnog projekta (programera, testera, arhitekta sustava i menadžera). Istražuje se mogućnost uvođenja nekih praksi XP-a u pojedinim fazama projekta. Analizirane su prednosti i nedostaci praksi XP i ne-XP metodologije, korištenih resursa u sâmoj implementaciji i kvaliteta kôda, tj. kakvoće proizvoda, uz uočavanje glavnih prednosti i nedostataka obaju pristupa.

Rad je organiziran u slijedeća poglavlja:

Drugo poglavlje opisuje metodologije i procese razvoja softvera. Daje definiciju metodologije i procesa razvoja softvera, opisuje povijesne aspekte nastajanja i razvoja metodologija te daje pregled danas najvažnijih i najraširenijih metodologija razvoja softvera, posebno ističući nove, agilne metode razvoja softvera.

Treće poglavlje opisuje glavne karakteristike agilnih metoda kao novijih metoda koje obilježava brži razvoj softvera. Kao glavni aspekt naglašava jednostavnost implementacije softvera, brzinu isporuke softvera, dobivanje povratne informacije (engl. *feedback*) od naručitelja te daljnje usmjeravanje razvoja softvera na dobivenim povratnim informacijama. Navedeni su glavni predstavnici agilnih metoda.

Četvrto poglavlje predstavlja uvod u metodologiju ekstremnog programiranja razvoja softvera. Opisuje ekstremno programiranje kao glavnog predstavnika agilnih metoda razvoja softvera. Prikazane su osnovne karakteristike ove metodologije. U ovom poglavlju se pokušava dati odgovor na pitanje kada treba koristiti ekstremno programiranje te koje su prednosti takve razvojne metodologije.

Peto poglavlje donosi detaljan opis glavnih pravila i praksi, odnosno fundamentalnih karakteristika ekstremnog programiranja, koje su razvrstane u četiri tipične discipline u razvoju softvera: planiranje razvoja, dizajn softvera, kodiranje (tj. implementacija) softvera te testiranje. Prikazan je i životni ciklus idealnog XP projekta te tipične projektne uloge.

Šesto poglavlje donosi prikaz *HC Agent* razvojnog projekta te problematiku uvođenja nekih praksi metodologije ekstremnog programiranja razvoja softvera u navedeni razvojni projekt. Istražuje se primjena određenih praksi XP-a u projekt s različitih stajališta projektnog tima (programera, testera, arhitekta sustava i menadžera). Također, istražuje se doprinos temeljnih praksi XP-a na status i napredak *HC Agent* razvojnog projekta.

Zatim, upoređuje se navedeni projekt koji koristi neke od praksi XP metodologije razvoja softvera sa "klasičnim" (ne XP, niti agilnim) projektnom razvoja softvera. Uočavaju se glavne prednosti i nedostaci obaju pristupa. Poglavlje sadrži i osvrt na budući rad na području efikasne implementacije XP metodologije razvoja softvera.

Sedmo poglavlje sadrži zaključak magistarskog rada.

Poglavlje 2

Pregled metodologija razvoja softvera

Povijesni nastanak metodologija razvoja softvera prati se još od kasnih 60-ih i ranih 70-ih godina 20. stoljeća. Naglim razvojem sklopovlja kontinuirano sve više raste složenost softverskih sustava.

Ovo poglavlje u svojoj uvodnoj riječi (odjeljak 2.1) daje osnovne definicije metodologija i procesa razvoja softvera.
Prikazuju se povijesni aspekti razvoja softvera i nastajanje različitih softverskih metodologija (odjeljak 2.2). Naglašena je problematika s kojom se razvoj softvera susretao sve do današnjih dana. Opisuju se okolnosti koje su dovele do nastanka metodologija razvoja softvera koje su objedinjene pod zajedničkim nazivom agilne metode (engl. *Agile Methods*) (poglavlje 3).
Opisuju se značajne metodologije razvoja softvera i prikazuju se njihove temeljne karakteristike (odjeljak 2.3).

2.1 Uvod u metodologije razvoja softvera

Proces razvoja softvera (engl. *software development process*) je korišteni slijed akcija, odnosno postupak koji se koristi kako bi se proizveo računalni program (ili programski sustav), odnosno programski proizvod [2].

U širem smislu, softver je cjelokupan skup programa, procedura i pripadne dokumentacije povezane sa sustavom, a naročito računalnim sustavom. Po svojoj naravi, proces razvoja softvera može biti i *ad hoc* proces, vođen od tima ljudi za jedan specifičan softverski razvojni projekt.

No, pojam se češće odnosi na standardiziranu i dokumentiranu metodologiju koja je korištena već prije na sličnim projektima razvoja softvera ili koja sa koristi unutar neke organizacije.

Iako ima mnogo različitih definicija metodologija, na kraju se one svode na isto. Metodologija razvoja softvera je postupak, odnosno kodificiran skup preporučenih praksi koje pokazuju kako organizacija odabire sustav ljudi i potrebnih resursa kako bi kreirala i održavala softverski proizvod [3].

Metodologija je prilagođavana prema različitim korisnicima koji će je koristiti [4]. Početnici traže detaljan slijed koraka koje treba pratiti kako bi se garantirao uspjeh. Oni

srednjeg znanja i iskustva traže rang strategija koje će koristiti u različitim trenucima u projektu. Eksperti su primarno zainteresirani na otkrivanje artefakata i kontrolnih točaka koje metodologija zahtijeva, kako bi se efikasno ostvarili zadani ciljevi i praćenje projekta.

U softverskom inženjerstvu (engl. *software engineering*), metodologija je popisan skup preporučenih praksi i teorije, ponekad popraćen s materijalima za treniranje, programima formalne edukacije, dijagramima tôka i slično [5].

Metodologije u softverskom inženjerstvu premošćuju mnoge discipline, uključujući upravljanje projektom (engl. *project management*), analizu, specifikaciju, dizajn, kodiranje, testiranje te osiguravanje kvalitete. Sve metodologije su zapravo svojevrsne kolekcija svake od tih disciplina.

Metodologije razvoja softvera se mogu općenito podijeliti u dvije glavne skupine [5] [6].

1. *Teške* i opsežne metodologije razvoja softvera (engl. *heavyweight methodologies*) sadrže mnogo pravila, načina postupanja i dokumentacije. Traže vrijeme i disciplinu za ispravno slijeđenje.
   Nazivaju se i "debelim" metodama (engl. *thick methods*).

2. *Lakše* i manje opsežne metodologije razvoja softvera (engl. *lightweight methodologies*) obično sadrže tek nekoliko pravila i načina postupanja koji su lagani za slijeđenje.
   Nazivaju se i "tankim" metodama (engl. *thin methods*).

Cilj razvoja softvera je proizvodnja kvalitetnog softvera, u zadanom vremenu, unutar predviđenog budžeta i uz zadovoljavanje stvarnih potreba naručitelja [7].

Uspjeh projekta ovisi o dobrom upravljanju zahtjevima (engl. *requirement management*).

Greške u zahtjevima i nepravilno specificirani zahtjevi su jedni od najčešćih problema u razvoju sustava i najskuplji su za ispravljanje. Neke ključne vještine mogu značajno smanjiti greške u određivanju zahtjeva softvera te poboljšati kvalitetu.

2.2 Povijesni aspekti nastanka i razvoja metodologija

U kasnim 60-tim te ranim 70-tim godinama 20. stoljeća zajednička praksa računalnih programera u stvaranju softvera je bilo kreiranje softvera na bilo koji način, tj. neorganizirano i nesređeno, na onaj način koji se trenutno mogao koristiti i koji je vodio cilju.
U to vrijeme se isticao velik broj računalnih programera koji su proizvodili softver koji je bio težak za razumijevanje ikome te jako kompleksan. Tada je naprosto bilo čudo ako se program izvršavao bez i jedne evidentirane neispravnosti. Držanje računala radno uspješnim je smatrano golemim naporom te svojevrsnim vrijednim istraživanjem.

Softversko inženjerstvo (engl. *Software Engineering*) je disciplina dizajniranja i proizvodnje softvera te uključuje sve potrebne akcije da bi se proizveo softver. Utemeljeno je na konferencijama sponzoriranim od NATO organizacije 1968. godine u *Garmisch-Partenkirchenu* i 1970. godine u *Rimu*. Programiranje je, dakle, samo dio discipline softverskog inženjerstva.

U to vrijeme je bilo uvriježena praksa da veće i discipliniranije metodologije mogu pomoći u razvoju softvera dosljedne kvalitete i predvidljive cijene. Time se zaustavila vladavina "razuzdanih" tzv. *cowboy* programera.

U 80-tim godinama 20. stoljeća nastupilo je bolje vrijeme za računalne programere. U to vrijeme je postojalo nekoliko pravila i praksi za kreiranje softvera koji je bio daleko superiorniji u kvaliteti od onog što je bio desetljeće ranije [8].

S vremenom se sve više povećava broj pravila i načina postupanja za pokrivanje potencijalnih problema u razvoju softvera. Značajno se povećava i kvaliteta softvera kao posljedica neprestanog usavršavanja metodologijâ razvoja softvera.

Konačno, u 21. stoljeću pravila te načini postupanja i djelovanja često postaju još teži za praćenje, procedure su katkad kompleksne i nerazumljive, a time i konačni softverski produkti. Količina dokumentacije u nekim apstraktnim notacijama lagano izmiče kontroli.

Programeri često odbacuju pravila i prakse složenih metodologija koje koriste (tzv. *cowboy* programeri); instinktivno se odriču teških i opsežnih metodologija razvoja softvera i vraćaju prema ranijim, jednostavnijim, lakšim i manje opsežnim metodologijama koje je lakše pratiti, gdje je tek nekoliko pravila (engl. *rules*) dovoljno. Kao suprotnost složenih i opsežnih metodologija, javljaju se lakše i manje opsežne metodologije.

Problematika koja se ovdje javlja je da računalni programeri i softverski inženjeri ne žele potpuno odbaciti pravila i praksu koja je pomagala razvoj kvalitetnog softvera (engl. *quality software*) kako ne bi nastao kaos. To znači, dakle, zadržavanje određenih pravila i praksi koje omogućuju razvoj kvalitetnog softvera. Rješenje je, dakako, jedino u pojednostavljenju složenih (previše kompleksnih) pravila i praksi, u onim situacijama i okolnostima gdje je to moguće.

Istodobno, ne želi se povratak ranim danima anarhičnog (engl. *cowboy*) kodiranja koje je prevladavalo u 70-im godinama 20. stoljeća. Umjesto toga, želi se zadržati dovoljno (ali ne previše) pravila i praksi kako bi se zadržala pouzdanost, kvaliteta i cijena softvera.

Rezultat tih napora su nove, lakše i manje opsežne metodologije razvoja softvera koje su objedinjene pod zajedničkim terminom agilne metode (poglavlje 3). Neke od metodologija razvoja softvera unutar agilnih metoda su ekstremno programiranje (poglavlje 4), *Lean Software Development* ("Mršavi" razvoj softvera) i druge.

2.3 Primjeri metodologija razvoja softvera

Slijedi opis značajnijih metodologija razvoja softvera. Težište je usmjereno na agilnim metodama (poglavlje 3) i na ekstremnom programiranju (poglavlje 4).

2.3.1 *Top-down* i *Bottom-up* model

Top-down [9] i *Bottom-up* [10] su dva pristupa korištena u razvojnoj metodi ili dvije kategorije razvojnih metodologija. Oba pristupa se mogu promatrati kao nadogradnja na druge, postojeće razvojne metodologije.

U *Top-down* kategoriji formuliran je pregled sustava bez ulaženja u detalje bilo kojeg njegovog dijela. Svaki pojedini dio sustava se tada dalje razvija dizajniranjem u više detalja. Svaki novi dio sustava može biti ponovno poboljšan i detaljnije definiran, sve dok cjelokupna specifikacija nije dovoljno detaljna kako bi se započeo razvoj, tj. implementacija.

U suprotnosti, u *Bottom-up* modelu individualni dijelovi sustava su detaljno specificirani te mogu mirno biti kodirani, tj. implementirani. Ti individualni kodirani dijelovi se tada spajaju zajedno kako bi tvorili veće komponente, koje su pak međusobno spajane sve dok nije spojen kompletan sustav.

Top-down pristup ističe planiranje te potpuno razumijevanje sustava. Svojstveno je da kodiranje ne može započeti dok nije dostignut dovoljan nivo detalja na barem nekim dijelovima sustava.
Pristup je 1970. godine promovirala IBM kompanija.

Bottom-up pristup ističe kodiranje koje može započeti već onda čim je specificiran prvi modul. Dakako, kodiranje u *Bottom-up* pristupu nosi rizik da moduli, koji su u fazi kodiranja, nemaju čvrstu ideju kako će se spojiti s ostalim dijelovima sustava te da to spajanje neće biti lagano kako na prvi pogled obično izgleda.

Moderni pristup dizajnu softvera obično kombinira metode od oba navedena pristupa. Iako se razumijevanje cijelog sustava obično smatra nužnim za dobar dizajn, veliki broj softverskih projekata pokušava iskoristiti već postojeći kôd (engl. *reuse*). Time postojeći moduli daju dizajnu *Bottom-up* smjer.

2.3.2 Vodopadni model

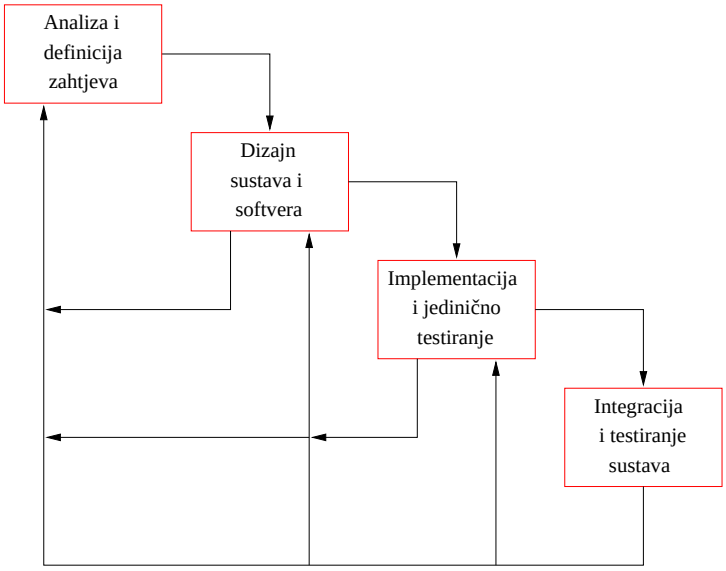
Vodopadni model (engl. *Waterfall model*) [5] je model, odnosno metodologija razvoja softvera koju je 1970. godine predložio *Winston W. Royce* u članku "*Managing Development of Large Scale Software*" [11].

Model je baziran na empiričkim opservacijama da cijena promjene raste eksponencijalno sa stupnjem promjene. Zaključak je da velike odluke potrebno donijeti čim je moguće više ranije jer je kasnija promjena skupa.

U tom modelu, razvoj slijedi linearno kroz faze analize zahtjeva (engl. *requirements analysis*), dizajna (engl. *design*), implementacije (kodiranja) (engl. *implementation*), testiranja (validacije) (engl. *testing (validation)*), integracije (engl. *integration*) i održavanja (engl. *maintenance*).
Proces se zove "vodopadni" jer svaka faza, tj. ciklus u modelu logički slijedi za prethodnim ciklusom (tj. fazom).

Slika (2.1) prikazuje glavne faze u vodopadnom modelu koje logički slijede za prethodnom fazom:

1. Analiza i definicija zahtjeva (engl. *Requirements Analysis and Definition*)
2. Dizajn sustava i softvera (engl. *System and Software Design*)



Slika 2.1: Vodopadni model (engl. *Waterfall model*)

3. Implementacija i jedinično testiranje (engl. *Implementation and Unit Testing*)
4. Integracija i testiranje sustava (engl. *Integration and System Testing*).

Iz bilo koje faze vodopadnog modela moguć je povratak u bilo koju fazu što je i prikazano strelicama na slici.

U praksi, vodopadni model rijetko slijedi čisto linearni stil već je čest iterativni pristup razvoja (odjeljak 2.3.6). *Winston W. Royce* je u svom originalnom članku naglasio ponovljivu upotrebu modela u iterativnom načinu.

Iteracija je karakteristika stanovitih algoritama ili računalnih programa u kojoj se sekvenca jednog ili više algoritamskih koraka obavlja u programskoj petlji. Iteracija se razlikuje se od ponavljajuće tehnike koja se zove rekurzija (engl. *recursion*).

Mnogi softverski inženjeri su u novije vrijeme diskreditirali vodopadni model u korist agilnih metoda razvoja (poglavlje 3). Glavni razlozi tome su težak i skup povratak u prethodnu fazu te dugo trajanje projekata koji koriste ovu metodologiju. Vodopadni model je prvenstveno orijentiran velikim i tehnički zahtjevnim projektima. Faza analize i definicije zahtjeva je kritična faza za propast projekta.

Vodopadni model je zapravo tradicionalni model razvoja softvera. Iako diskreditiran u korist agilnih metoda, i dalje je često korišten u praksi [12].
Glavne prednosti modela su: preglednost, vidljivost i dobra organiziranost. Moraju postojati dobro definirani zahtjevi kako bi se uspješno provodile slijedeće faze. Vodopadni model je orijentiran prema opsežnoj dokumentaciji.
Glavni nedostaci modela su: nefleksibilno particioniranje projekta u točno određene dijelove što ga čini teškim za odgovor na promjene zahtjeva kupca te težak i skup povratak u prethodnu fazu.

2.3.3 Spiralni model

Spiralni model [5] je model, odnosno metodologija razvoja softvera koja ima načelo u kombiniranju elemenata dizajna i stadija (etapa) iz prototipa. Zapravo, to je svojevrsna mješavina *Top-down* i *Bottom-up* koncepata (odjeljak 2.3.1).

Spiralni model je definirao *Barry Boehm* člankom "*A Spiral Model of Software Development and Enhancement*" 1986. godine [13]. Najveća značajka modela je eksplicitno prepoznavanje i uklanjanje rizika.

Grafički prikaz modela je u obliku spirale u koordinatnom sustavu. Svaka petlja spirale predstavlja jednu fazu procesa razvoja softvera. Petlja počinje iz kvadranta problema gdje se vrši pregled izvedivosti. U slijedećem kvadrantu se obavlja analiza rizika i odabir prototipa. Slijedi kvadrant s programskim dijelom (simulacije, modeli i procjena) i sa provjerom (validacija). Zadnji kvadrant predstavlja integraciju i provjeru.

No, taj model nije bio prvi model za raspravu o iteracijama već je bio prvi model koji je raspravljao o važnosti iterativnog dizajna. Originalno je zamišljeno da je dužina iteracija tipično 6 mjeseci do dvije godine.

Prednosti spiralnog modela su:

- Eksplicitno prepoznavanje rizika koje omogućuje njegovo efikasno uklanjanje.
- Procjene (budžet i plan) su realističnije kako projekt napreduje jer se povećava broj pitanja.
- Moguće je uhvatiti se u koštac s (gotovo neizbježnim) promjenama koje razvoj softvera općenito nameće.
- Softverski inženjeri mogu ranije početi djelovati na projektu.

Glavni nedostatak spiralnog modela je što su procjene koje se tiču budžeta i plana grube na početku jer neke analize nisu završene sve dok te etape nisu prošle fazu dizajna.

Spiralni model kao takav u današnje vrijeme (2005. godina) nije korišten [14] u većoj mjeri. Dakako, utjecao je na moderne koncepte razvoja softvera današnjice, naročito na tzv. agilne metode (poglavlje 3) koje nastoje biti ekstremnije u svojem pristupu nego što je spiralni model.

2.3.4 Model kaosa

U svijetu razvoja softvera, model kaosa (engl. *Chaos model*) je struktura razvoja softvera koji proširuje spiralni model (odjeljak 2.3.3) i vodopadni model (odjeljak 2.3.2) razvoja softvera. Model kaosa je definirao *L.B.S. Raccoon* radom "*The Chaos Model and the Chaos Life Cycle*" [15].

Model kaosa pretpostavlja da su faze životnog ciklusa programskog proizvoda primjenjive na sve nivoe projekta, od cjelokupnog projekta do individualnih linija kôda.

U matematici i fizici, teorija kaosa postupa s ponašanjem određenih nelinearnih dinamičkih sustava koji, pod određenim okolnostima, predočavaju fenomen poznatiji kao kaos, izvanredno karakteriziran osjećajnošću za početna stanja. Primjeri takvog sustava uključuju atmosferu, sunčev sustav, turbulentne fluide itd.

Nelinearni dinamički sustav općenito može pokazati slijedeće tipove ponašanja:

- zauvijek u miru
- zauvijek ekspandira (samo za neomeđene, tj. neograničene sustave)
- periodička kretanja
- kvaziperiodička kretanja
- kaotična kretanja

Tip ponašanja može ovisiti o početnom (inicijalnom) stanju sustava i vrijednostima parametara ako oni postoje.

Postoji nekoliko doprinosa modela kaosa:

- Model kaosa može pomoći dati odgovor na pitanje zašto je softver toliko nepredvidiv.
- Model objašnjava zašto koncepti visokog nivoa kao što je arhitektura ne mogu biti tretirani neovisno od linija kôda koje su nižeg nivoa.
- Model omogućuje objašnjenje što napraviti slijedeće (slijedeći korak) sukladno sa strategijom kaosa.

Slijede najvažnije pretpostavke za ostvarenje modela kaosa:

- Cijeli projekt treba biti definiran, implementiran te integriran.
- Sustav koji se razvija mora biti definiran, implementiran te integriran.
- Moduli sustava moraju biti definirani, implementirani te integrirani.
- Funkcije modula sustava trebaju biti definirane, implementirane te integrirane.
- Linije kôda trebaju biti definirane, implementirane te integrirane.

2.3.5 Model protipa

Model prototipa (engl. *Prototyping model*) [16] je proces razvoja softvera koji počinje s kolekcijom zahtjeva (engl. *requirements collection*) iza čega slijedi izrada prototipa i evaluacija od strane korisnika (tj. naručitelja).

Često, krajnji korisnik nije u mogućnosti pribaviti kompletan skup građe za sustav koji se implementira, detaljne informacije ili zahtjeve za inicijalni stadij. Nakon korisničke

evaluacije prototipa, obično se gradi drugi prototip koji je baziran na povratnoj sprezi dobivenoj od kupca. Ponovno se dobiva povratna informacija, tj. evaluacija od kupca.

Ciklus započinje slušanjem kupca (ili naručitelja) iza čega slijedi izgradnja ili revizija modela, te kasnije kupac (naručitelj) vrši testiranje modela. Ciklus se nakon toga ponavlja.

Razlikuju se dva osnovna tipa prototipa:

- evolucionarni prototip (prototip se odabire za daljnje korištenje)
- odbacujući prototip (prototip se odbacuje).

Pojam "prototip" ne odnosi se isključivo na softver. Kompanije mogu proizvesti prototip produkata kako bi pokazali određene karakteristike ili kako bi imale model koji radi prije poboljšavanja ostalih dijelova iz dizajna.

U elektronici, izrada prototipova znači izradu električnog kruga prema teoretskom dizajnu za verifikaciju valjanosti dizajna, te za pripremu fizičke platforme za otkrivanje neispravnosti dizajna.

2.3.6 Iterativni i inkrementalni razvoj

Iterativni i inkrementalni razvoj (engl. *Iterative and Incremental development*) [16] je proces razvoja softvera, te također jedna od praksi koja se koristi u metodologiji ekstremnog programiranja (poglavlje 4) kao predstavniku agilnih metoda (poglavlje 3).

Inkrement je definiran kao podsustav sustava. Završeni inkrement je podsustav za koji vrijedi da su faze analize, dizajna i kodiranja sustava završene, revidirane i testirane.

Iteracija je definirana kao prolaz preko određenog skupa aktivnosti. Iterativni proces je revizija i nastavak rada na prijašnjem poslu.

Osnovna ideja ovog modela je inkrementalan razvoj softverskog sustava koji omogućuje programerima da iskoriste prednosti od onoga što je bilo naučeno tijekom razvoja ranijih, inkrementalnih verzija sustava koje su često bile i isporučene kupcu (naručitelju). Učenje dolazi i iz razvoja i iz korištenja sustava, gdje je to moguće [17].

Ključni koraci u procesu su započeti sa jednostavnom implementacijom podskupa softverskih zahtjeva te iterativno povećavati skup softverskih zahtjeva (isporučujući pri tom verzije) sve dok svi programski zahtjevi nisu realizirani, što znači da je sustav implementiran u potpunosti. U svakoj iteraciji čine se modifikacije dizajna zajedno sa implementacijom novih funkcionalnosti (sukladno softverskim zahtjevima).

Discipline unutar svake iteracije su: prikupljanje zahtjeva, dizajn, implementacija i test. Svaka iteracija je zapravo mini-projekt.

Iterativna i inkrementalna metoda razvoja se sastoji od inicijalizacijskog koraka (engl. *Initialization step*), iteracijskog koraka (engl. *Iteration step*) i projektne kontrolne liste (engl. *Project Control List*).

Inicijalizacijski korak služi kreiranju osnovne (bazne) verzije sustava. Cilj te inicijalne implementacije je kreiranje proizvoda na kojeg korisnik može djelovati.
Taj korak treba ponuditi primjer ključnih aspekata problema i omogućiti rješenje koje je dovoljno jednostavno za razumijevanje i za implementaciju.

Za vođenje iterativnog procesa, kreirana je projektna kontrolna lista koja sadrži zapise svih zadataka koji trebaju biti izvršeni. Sadrži i pojedine stavke kao nove značajke koje treba implementirati te područja redizajna postojeće solucije.
Projektna kontrolna lista je kontinuirano pregledavana i revidirana kao rezultat faze analize.

Iteracijski korak povlači za sobom redizajn i implementaciju zadataka iz projektne kontrolne liste te analizu trenutne verzije sustava. Cilj dizajna i implementacije bilo koje iteracije je da bude čim jednostavnija, izravna i modularna, podržavajući redizajn u trenutnom stadiju ili na mjestu gdje se zadaci dodaju u projektnu kontrolnu listu.

Kôd reprezentira glavni izvor dokumentacije sustava. Analiza iteracije je bazirana na korisničkoj povratnoj informaciji i mogućnosti analize sustava koji je u implementaciji. To obuhvaća analizu strukture kôda, modularnost, upotrebljivost, pouzdanost, efikasnost te postignuće ciljeva. Projektna kontrolna lista je modificirana prema rezultatima analize.

Linija vodilja implementacije i analize sadrži:

- Bilo koja poteškoća u dizajnu, kodiranju i testiranju treba signalizirati modifikaciju (redizajn ili promjene u kôdu).
- Modifikacije trebaju biti prilagodljive postojećim izoliranim modulima. Ako to nije slučaj, potreban je redizajn.
- Modificiranje tablica baze podataka treba biti izvedivo posebno lagano. U slučaju da su modifikacije tablica zahtjevne, potreban je redizajn.
- Kako iteracije napreduju, modifikacije bi trebale biti lakše izvedive. Ako to nije slučaj, znači da postoji osnovni problem kao što je propust u dizajnu ili problem s *patch*-evima.

  U svijetu računala, softverska zakrpa (engl. *patch*) je softverska nadogradnja (engl. *update*) za ispravljanje problema, neispravnosti ili upotrebljivosti prethodne verzije aplikacije. To može uključiti bilo koji računalni program u rangu od tekstualnih editora, računalnih igara do operativnih sustava i Web servisa. Termin *patch* potječe od Unix *patch* komande *Larry Wall*-a.
- Preporučljivo je postojanje *patch*-eva za samo jednu ili maksimalno dvije iteracije. Postojanje *patch*-eva je nužno kako bi se izbjegao redizajn u fazi implementacije.
- Postojeća implementacija treba biti često analizirana kako bi se odredilo udovoljavanje prema ciljevima projekta.
- Treba biti korištena mogućnost programske analize kad god je to moguće, kako bi se pomogla analiza djelomične implementacije.
- Reakcija treba biti tražena i analizirana od strane korisnika (kupca ili naručitelja) za indikaciju nedostataka trenutne implementacije.

Iterativni pristup je uspješno korišten za razvoj familije prevodioca za familiju programskih jezika koji su primjenjivi na mnoštvu sklopovskih arhitektura.

Korištenje analize i mjerenja kao goniča iterativnog poboljšavajućeg procesa je glavna razlika između iterativnog procesa i postojećih agilnih metoda (poglavlje 3). Omogućuje podršku za određivanje djelotvornosti procesa i kvalitete proizvoda. Također, omogućuje proučavanje, poboljšavanje i prilagodbu procesa za određeni okoliš. Te aktivnosti mjerenja i analize mogu biti aktivno dodane postojećim agilnim razvojnim metodama.

Sklop višestrukih iteracija osigurava prednosti korištenja mjerenja. Mjerenja su katkad teška za razumijevanje u potpunosti, ali relativne promjene u mjerenjima preko evolucije sustava mogu biti vrlo informativne jer time osiguravaju osnovu za usporedbu. Na primjer, vektor mjerenja m_1, m_2, ..., m_n može biti definiran da bi karakterizirao različite aspekte proizvoda u nekom vremenskom trenutku; npr. vrijeme, promjene, neispravnosti te logičke, fizičke i dinamičke atribute. Analizator može odrediti kako se karakteristike proizvoda, kao što su veličina, kompleksnost, spajanje i kohezija povećaju i smanjuju u vremenu. Mjeriti se mogu i relativne promjene različitih aspekata produkta ili predvidjeti granice za mjerenja za signaliziranje potencijalnih problema i anomalija

2.3.7 *Rational Unified Process*

Rational Unified Process (RUP) su razvili *Philippe Kruchten, Ivar Jacobsen* i ostali u *Rational Corporation* kompaniji kako bi upotpunili UML (*Unified Modelling Language*), koji je industrijski standardizirana metoda modeliranja softvera [18].

RUP je interaktivni i inkrementalni pristup za objektno-orijentirane sustave, strogo prihvaća slučajeve uporabe (engl. *Use Cases*) kao zahtjeve u modeliranju. Iako implicitno ne isključuje ostale metode, naglašava UML metodu i objektno-orijentirani (OO) razvoj (*Jacobsen*, 1994. godine).

RUP naglašava razvoj softvera u fazama [19]. Svaka faza se sastoji od jedne ili više iteracija. RUP opisuje četiri faze kroz koje prolazi projekt: inspekcija, elaboracija, konstrukcija i tranzicija.
Sadržaj faze inspekcije je kreiranje vizije, razvoj *business case*-a, stvaranje softverskog prototipa ili djelomičnog rješenja. U fazi elaboracije se donose ključni arhitekturalni zaključci i eliminiraju rizici. Izvršna arhitektura proizvodi softver na temelju ključnih arhitekturalnih zaključaka. U fazi konstrukcije se vrši implementacija funkcionalnosti koja je identificirana u arhitekturi. U fazi tranzicije je fokus na isporuci softvera kupcu.

Faze su podijeljene na iteracije. Iteracije su vremenski određene te imaju specifične ciljeve. Iteracije se drže vremenski što je moguće kraćima ali dovoljno dugima kako bi se implementirali potpuni slučajevi uporabe ili određeni scenariji koji predstavljaju određenu vrijednost za korisnika. Na kraju svake iteracije se prilagođavaju planovi za buduće iteracije na temelju rezultata trenutne iteracije.

RUP predstavlja kreiranje vizije onoga što se želi, kreiranje okružja (engl. *framework*) kako bi se to postiglo, te procjene na zadanim točkama da li se nalazi na putu gdje se namjeravalo biti.

Poglavlje 3

Agilne metode (engl. *Agile Methods*) razvoja softvera

Agilno označava spremno na pokret, aktivnost, žustrost, hitrost. Agilne metode razvoja softvera pokušavaju ponuditi odgovor na želju poslovne zajednice za manje opsežnim metodologijama razvoja softvera, koje sa sobom donose brže, žustre procese razvoja softvera [18].

U softverskom inženjerstvu, agilne metode razvoja softvera su manje opsežne metode koje prihvaćaju činjenicu da je softver teško kontrolirati [18]. One minimiziraju rizik time što osiguravaju da su inženjeri koji razvijaju softver fokusirani na male jedinice posla. To je naročito važno sa današnjim rapidnim rastom industrije vezane uz Internet, sa okolinom vezanom uz mobilne aplikacije te s distribuiranim razvojem općenito.

Način na koji je agilni razvoj softvera općenito odijeljen od "težih", više procesno-orijentiranih metodologija, npr. vodopadnog modela (odjeljak 2.3.2), predstavlja naglasak na vrijednostima i principima umjesto na procesu.

Tipični ciklusi su jedan tjedan ili jedan mjesec. Na kraju svakog ciklusa vrednuju se projektni prioriteti što je karakteristika koju agilne metode dijele s modelom iterativnog razvoja (odjeljak 2.3.6) i modernim teorijama upravljanja projektima.

3.1 Povijesni aspekti nastanka agilnih metoda

U posljednjih 25 godina isproban je velik dio različitih pristupa u razvoju softvera, od kojih je samo nekoliko istinski zaživjelo.

Agilne metode su se razvijale sredinom 90-ih godina 20. stoljeća kao dio reakcije na tzv. visoko formalne metode (engl. *high ceremony methods*), kao što su CMM (engl. *Capability Maturity Model*), *Prince* i *Rational Unified Process*. Proces razvoja koji vuče porijeklo od ovih metoda je u određenim primjenama viđen kao birokratski i spor.

Agilni pokret (engl. *Agile Movement*) u industriji softvera je započeo 2001. godine kada je grupa softverskih praktičara i konzultanta (*Kent Beck*, *Alstair Cockburn* i ostali) objavila "*Agile Software Development Manifesto*" [20] [17].

Uvođenje metodologije ekstremnog programiranja (engl. *Extreme Programming*) 1999 godine, poznatije kao XP, je široko prihvaćena kao startna točka različitih pristupa agilnog razvoja softvera. Postoji, također, mnoštvo drugih metoda koje pripadaju zajed-

ničkoj porodici agilnih metoda. Neke od tih metoda, odnosno metodologija su: *Crystal Methods* (*Cockburn*, 2000. godine), *Feature-Driven Development* (*Palmer* i *Felsing*, 2002. godine), *Adaptive Software Development* (*Highsmith*, 2000. godine) i druge.

3.2 Svojstva agilnih metoda

Glavni aspekt agilnih metoda je jednostavnost i brzina. U razvojnom radu, tim je koncentriran samo na funkcije koje su potrebne u prvoj ruci i na njihovu implementaciju, zatim na brzu isporuku, dobivanje povratne informacije od naručitelja te reakcije na primljene informacije.

Glavno pitanje je što čini razvojnu metodu agilnom? To je slučaj kada je razvoj softvera:

- inkrementalan (malene isporuke, s brzim ciklusima)
- kooperativan (naručitelj i razvojni tim rade neprestano zajedno u bliskoj komunikaciji)
- izravan (metoda je jednostavna za učenje i modificiranje te dostatno dokumentirana)
- prilagodljiv (u mogućnosti da se čine promjene u posljednjem trenutku).

Tipični ciklus projekta [21] je u trajanju od jednog tjedna ili jednog mjeseca i na kraju svakog ciklusa se vrši ocjena projektnih prioriteta, što je karakteristika i inkrementalnih metodologija razvoja softvera (odjeljak 2.3.6) i modernih teorija projektnog vođenja.

Općenito, agilne metode nameću korištenje nepotrebnih troškova što je manje moguće, u formi principa, opravdanosti, izvještavanja i dopuštenja.

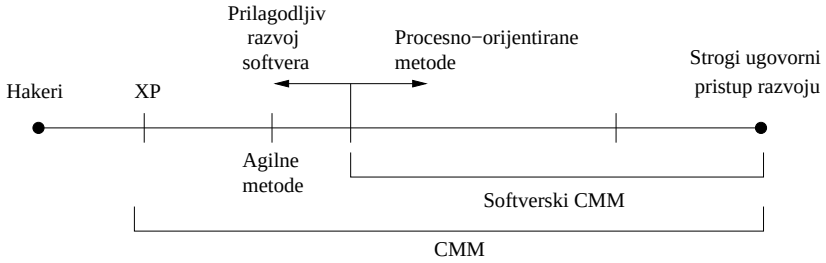
3.2.1 Pregled i definicije

Glavni principi agilnog pokreta objavljenih u manifestu su [17]:

- Individualnost i interakcije iznad procesa i alatâ
  Agilni pokret naglašava vezu i zajedništvo računalnih programera i ljudskih uloga u suprotnosti s institucionaliziranim procesima i razvojnim alatima. U postojećim agilnim praksama, to se manifestira u bliskosti projektnog tima i dobrom timskom duhu.
- Softver koji radi iznad opsežne dokumentacije
  Vitalni zadatak softverskog tima je kontinuirana isporuka testiranog softvera koji radi. Nove verzije se isporučuju u pravilnim razmacima (npr. nekoliko puta mjesečno). Programeri su dužni održavati kôd jednostavnim i tehnički dorađenim čim je više moguće. Naglasak nije na količini proizvedene dokumentacije.
- Suradnja s kupcem (engl. *customer*) iznad striktnih ugovora
  Veza i suradnja između programera i naručitelja je u prednosti nad striktnim

ugovorima, iako važnost dobro sročenih ugovora raste istim tempom kao i veličina softverskih projekata. Iz poslovnog kuta gledanja, agilni razvoj je fokusiran na brzu isporuku, ubrzo nakon što je projekt startan, čime se smanjuju brojni rizici (promjene zahtjeva, testiranost, odustajanje od projekta i drugi).

- Odgovor na promjene iznad slijeđenja plana
  Razvojna grupa koja sadrži i programere i naručitelja trebala bi biti dobro informirana, kompetentna i autoritativna kako bi uzela u obzir moguće prilagodbe za vrijeme razvojnog puta projekta. To znači da bi sudionici tima trebali biti pripremljeni da čine promjene.



Slika 3.1: Odnos agilnih metoda i ostalih pristupa razvoja softvera

Slika (3.1) prikazuje spektar različitih pristupa razvoju softvera, gdje su *ad-hoc* (*hacker*ski pristup) smješteni na jednom kraju a drugi, klasični, procesno-orijentirani pristup na drugom, suprotnom kraju. Agilne metode su bliže *ad-hoc* pristupu.

Cockburn [4] je definirao jezgru agilnih metodologija razvoja softvera koristeći neopsežna ali dovoljna (engl. *light-but-sufficient*) pravila ponašanja u projektu korištenjem humanih i komunikacijski-orijentiranih pravila. Agilni proces je istovremeno i jednostavan i dovoljan. Predlaže se postojanje ovih praksi koje omogućuju prosperitet projekata:

- dva čovjeka do osmero ljudi u jednoj sobi
  (komunikacija, dijeljenje znanja)
- često korištenje eksperta
  (kratke i kontinuirane povratne informacije)
- kratke inkrementalne verzije
  (jedan do tri mjeseca, omogućeno je brzo testiranje i popravak grešaka)
- potpuno automatizirano regresijsko testiranje
  (jedinično i funkcijsko testiranje stabilizira kôd i omogućuje kontinuirana poboljšanja)
- iskusni programeri
  (iskustvo ubrzava vrijeme razvoja i to 2 do 10 puta u usporedbi sa sporijim članovima tima; obično rade uz bilo koju metodologiju).

3.2.2 Karakteristike

Miller [22] je prikazao slijedeće karakteristike agilnih softverskih procesa iz kuta gledanja brze isporuke, koje omogućuju skraćivanje životnog ciklusa projekata:

1. modularnost stupnja razvojnog procesa
2. kratke iteracije koje omogućuju brze verifikacije i korekcije
3. vremenski okvir iteracija je od jednog do šest tjedana
4. micanje svih nepotrebnih aktivnosti
5. prilagodljivost s mogućim nenadanim rizicima
6. inkrementalni pristup procesu koji omogućava funkcionalnu izgradnju softverskog produkta u malim koracima
7. konvergentni i inkrementalni pristup minimizira rizike
8. ljudski-orijentirani agilni proces favorizira ljude iznad procesa i tehnologija
9. kolaborativni i komunikativni radni stil.

Odabir odgovarajuće procedure nije toliko orijentiran kako bi zaustavio promjene rano u projektu, već kako se bolje nositi s neminovnim promjenama tijekom čitavog životnog ciklusa projekta. Agilne metode su zapravo dizajnirane kako bi:

- proizvele prvu isporuku u ranim tjednima projekta, kako bi se postigla "brza pobjeda" i rapidna povratna informacija od kupca
- osmislile jednostavno rješenje tako da je manje toga za mijenjati i izrada tih promjena je jednostavnija
- kontinuirano unaprijedile kvalitetu dizajna, čineći slijedeću iteraciju jeftinijom za implementaciju
- potakle kontinuirano testiranje za raniju i time manje skuplju detekciju neispravnosti.

Osnovni principi agilnih metoda uključuju čistoću kôda koji radi, efektivnost ljudi koji rade zajedno sa dobrom voljom te je fokus zapravo na timskom radu [23]. Skup pristupa koji izviru iz agilnih procesa razvoja softvera su slijedeći:

- ljudima je stalo da razvojni projekt uspije
- čim manje dokumentacije (ako je moguće)
- komunikacija o kritičnim stvarima
- alati za modeliranje nisu korisni kao što se obično misli

- veliki dizajn unaprijed nije poželjan (iako je ovo nepovoljno prilikom razvoja sigurnosnih sustava).

Budućnost informacijskog doba je i u agilnim metodologijama [24]. Softverske kompanije koje koriste agilnost imaju kapacitet da povećaju inovativnost i brzinu te kreiraju promjene kod konkurenata. Sporije i manje inovativne kompanije doživljavaju kaos koji su drugi pokrenuli. Sposobnosti kompanija da drže korak i kreiraju promjene leži u sposobnosti razvoja softvera. U svijetu neprestanih promjena, tradicionalne rigorozne metode razvoja softvera su nedovoljne za uspjeh.
Sada je trend ko-opeticije, što za procese znači kombinaciju tradicionalnih i agilnih metoda.

3.3 Primjeri agilnih metoda

Kao rezultat agilnog pristupa u razvoju softvera, mogu se izdvojiti slijedeće postojeće agilne metode koje su definirane na prethodnim načelima (odjeljak 3.2) jednostavnosti i brzine:

- *Extreme Programming* (*Beck*, 1999. godina)
- *Scrum* (*Schwaber* 1995. godine, *Schwaber* i *Beedle* 2002. godine)
- *Crystal* porodica metodologija (*Cockburn* 2002. godine)
- *Feature Driven Development* (*Palmer* i *Felsing* 2002. godine)
- *Rational Unified Process* (*Kruchten* 1996. godine, *Kruchten* 2002. godine)
  Po naravi, RUP nije tipičan predstavnik agilnih metoda. Orijentiran je na stvaranje i korištenje velike količine dokumentacije.
- *Dynamic System Development Method* (*Stapleteon* 1997. godine)
- *Adaptive Software Development* (*Highsmith* 2000. godine)
- *Open Source Software Development* (*O'Reilly* 1999. godine).

Poglavlje 4

Razvojna metodologija ekstremnog programiranja

Ekstremno programiranje (*Extreme Programming*, XP) je pristup, odnosno metoda razvoja softvera kuju su formulirali *Kent Beck*, *Ward Cunningham* i *Ron Jeffries* [3]. *Kent Beck* je napisao prvu knjigu na tu temu pod naslovom "*Extreme Programming Explained*" [25] koja je objavljena 1999. godine. Metoda ekstremnog programiranja je najpopularnija od nekoliko agilnih metoda razvoja softvera (poglavlje 3).

XP je predstavnik lakših i manje opsežnih metodologija razvoja softvera namijenjen prvenstveno malim timovima koji razvijaju softver, suočeni sa neodređenim zahtjevima ili sa zahtjevima koji se dinamički mijenjaju. Predstavnik je agilnih metoda razvoja.

XP je izniknulo iz problema prouzročenog dugim razvojnim ciklusom tradicionalnih razvojnih modela. Krenulo je jednostavno kao prilika kako bi se završio posao s praksama koje su smatrane dovoljno efikasnima u procesu razvoja softvera. Nakon dovoljno velikog broja pokušaja u praksi, XP metodologija je dokumentirana s ključnim principima i praksama.

Individualizirane prakse XP-a nisu same po sebi nove, u XP-u one su sakupljene kako bi zajedno funkcionirale i činile novu metodologiju razvoja softvera. Pojam ’ekstremno’ dolazi od tih zajedničkih praksi koje su dokazane kao dobro primjenjive i koje su dovedene do svojih ekstremnih granica [25].

U XP-u postoji proces kojeg se slijedi ali je zajedničko razmišljanje da je taj proces više nalik pravilima igre [24].

4.1 Uvod u ekstremno programiranje

Ekstremno programiranje je discipliniran pristup razvoju softvera [8]. Ovaj pristup je star desetak godina (dakle, počeo se koristiti u prvoj polovici devedesetih godina prošlog stoljeća) i dokazan je u mnogim kompanijama različitih veličina te u industriji u zemljama širom svijeta [24].

Pokretač metodologije je bio *Kent Beck* koji je razmišljao o boljem načinu razvoja softvera. Proveo je neko vrijeme radeći s *Ward Cunningham*om i zajedno su iskusili

pristup razvoju softvera koji svaku stvar čini jednostavnijom i učinkovitijom. *Kent Beck* je u tvrtki *DaimlerChrysler* započeo jedan specifičan razvojni projekt u proljeće 1996. godine koristeći nove koncepte razvoja softvera. Rezultat je bio metodologija ekstremnog programiranja razvoja softvera.

Dakle, po svojoj starosti, to je jedan od mlađih pristupa. Smatra se prvom uspostavljenom agilnom metodom (poglavlje 3) nakon nekih zajedničkih popularnih taktika među računalnim programerima.

Jedna od temeljnih vrijednost XP-a je zadovoljstvo naručitelja softvera. Zadovoljstvo naručitelja je konačni cilj u ciklusu razvoja softvera jer uključuje ispunjenje, tj. implementiranost ulaznih zahtjeva. Ti ulazni zahtjevi su ispunjeni ako je naručitelj softvera zadovoljan s implementiranom funkcionalnošću. Metodologija ekstremnog programiranja je dizajnirana za isporuku softvera koju naručitelj treba u trenutku kada je ona potrebna (tj. vrši se implementacija točno onoga što je prethodno dogovoreno). XP omogućuje programerima odgovor na promjene zahtjeva koje donosi naručitelj softvera, čak i kasno u procesu razvoja softvera.

Ova metodologija naglašava timski rad. Menadžeri, naručitelji softvera i programeri su zajedno dio tima koji je zadužen za isporuku kvalitetnog softvera. XP tim je time prošireni razvojni tim jer njega ne čine samo programeri, nego i menadžeri i naručitelj. XP implementira jednostavan ali efikasan način razvoja softvera koji se temelji na grupama (parovima) (odjeljak 5.3.4).

Fundamentalne, tj. temeljne karakteristike modela ekstremnog programiranja su [26]:

1. Igra planiranja (engl. *Planning game*)
   Korisnička interakcija u programerskom (implementacijskom) timu između programera i naručitelja oko procjena implementacije pojedine funkcionalnosti.

2. Malene česte isporuke (engl. *Small/short releases*)
   Sustav se brzo i često isporučuje, najmanje svaka 2 do 3 mjeseca. Ovaj pristup se temelji na praksi iterativnog i inkrementalnog razvoja (odjeljak 2.3.6).

3. Organizacija sustava s metaforama (engl. *Methaphor*)
   Metafora je pojednostavljena slika sustava u razvoju.

4. Jednostavan dizajn (engl. *Simple Design*)
   Naglasak je na dizajnu najjednostavnijeg rješenja koji je zahtijevan u tom trenutku, bez dodatnog kôda i viška funkcionalnosti.

5. Testiranje (engl. *Testing*)
   Kontinuirano, često ponavljajuće automatizirano jedinično (engl. *unit*) testiranje i regresijsko (engl. *regression*) testiranje.

6. Korištenje tehnike refaktoriranja (engl. *Refactoring*)
   Micanje dvostrukog (redundantnog) kôda i održavanje kôda jednostavnim.

7. Programiranje u pâru (engl. *Pair Programming*)
   Ovaj princip znači da uvijek dva čovjeka pišu određeni kôd.

8. Zajedničko dijeljenje kôda (pristup kôdu) (engl. *Collective Ownership*)
   Bilo tko iz tima smije mijenjati bilo čiji kôd.

9. Kontinuirana integracija (engl. *Continuous integration*)
   Novi kôd se integrira u sustav čim je spreman (implementiran i testiran).

10. 40 satni radni tjedan (engl. *40-hours week*)
    Maksimum je 40-satni radni tjedan. Nisu poželjni uzastopni prekovremeni tjedni zbog slamanja timskog duha.

11. Povratna informacija od kupca (naručitelja) (engl. *On-site customer*)
    Naručitelj je stalno na raspolaganju programerima.

12. Standardi kodiranja (engl. *Coding Standards*)
    Postoje standardi kodiranja i programeri ih slijede kako bi kôd na kojem se trebaju načiniti bilo kakve promjene, a napisao ga je netko drugi u timu, bio čim razumljiviji.

Programeri su prvenstveno motivirani željom kreiranja softvera koji radi i na koji su ponosni koliko je dobar. Motivacija je zapravo bazirana na ponosu i programeri se jednostavno žele istaknuti. To je razlog što žele koristiti XP metodologiju razvoja softvera (pored ostalih agilnih metoda) jer osjećaju da agilne metode omogućuju isporuku softvera koji radi [4].

XP poboljšava softverski projekt u četiri esencijalna točke: komunikaciji, jednostavnosti, povratnoj informaciji od naručitelja (kupca) te odvažnosti ili hrabrosti. XP programeri komuniciraju sa svojim naručiteljima softvera i svojim kolegama (računalnim programerima). Svoj dizajn softvera nastoje držati jednostavnim i čistim. Važna točka u razvoju softvera je provođenje testiranja (počevši već s prvim danom), čime se dobiva povratna informacija da li se softver ispravno ponaša. Poželjno je softver isporučiti naručitelju (kupcu) najranije što je moguće, implementirajući promjene sukladno dogovorima s naručiteljem (kupcem). S tim pristupom, XP programeri su u mogućnosti odgovoriti zahtjevima i tehnologiji koji su skloni čestom mijenjanju.

Ekstremno programiranje je po svojim temeljnim karakteristikama različita metodologija razvoja softvera od drugih, tradicionalnih pristupa razvoju softvera. Dobra usporedba metodologije ekstremnog programiranja je sa pazlama (engl. *puzzle*). Postoji mnogo malih komadića koji individualno ne znače mnogo ali kada se kombiniraju zajedno, može se dobiti potpuna slika.

4.2 Područja primjene metodologije ekstremnog programiranja

Ekstremno programiranje je kreirano kao odgovor na domenu problema čiji se zahtjevi mijenjaju. Naručitelj softvera (engl. *customer*) ne treba imati čvrstu i konačnu

ideju što sustav u konačnici treba raditi. Moguće je da programeri razvijaju sustav čija se funkcionalnost mijenja svakih nekoliko mjeseci. U mnogim softverskim okolinama dinamičko mijenjanje zahtjeva je zapravo konstanta. Ovo je svakako područje gdje XP može uspjeti dok ostale metodologije razvoja softvera mogu zakazati i čest slučaj u praksi je da zakazuju.

XP također dotiče i probleme vezane za rizike u projektu. U slučaju da naručitelj softvera treba novi sustav do određenog datuma, rizik je velik. Ako je taj sustav i novi izazov za grupu programera u timu, rizik je još i veći. U slučaju da je sustav koji se razvija nov izazov za cjelokupnu softversku industriju, rizik je time još veći. Metodologija ekstremnog programiranja svojim praksama omogućava da se ublaži rizik i poveća vjerojatnost uspjeha projekata.

Ekstremno programiranje je prvenstveno namijenjeno maloj grupi programera, obično između 2 i 12. No, i veći projekti od 30 i više programera su zabilježili uspjeh koristeći ovu metodologiju razvoja softvera ili neke njene prakse. Ipak, ekstremno programiranje nije preporučljivo na velikim projektima iako postoje i pozitivna iskustva [25] [27].

Dakako, treba naglasiti da na projektima s dinamičkim zahtjevima ili visokim rizicima mala grupica XP programera može biti (a obično i je) učinkovitija nego veliki tim programera.

Metodologija ekstremnog programiranja zahtijeva prošireni razvojni tim. Kao što je već rečeno, XP tim uključuje ne samo programere, nego još menadžere i naručitelje softvera, tvoreći tako prošireni razvojni tim. Svi sudionici tima rade zajedno i jedni su drugima pri ruci. Postavljanje pitanja, područje pregovaranja te vremenski rokovi vezani uz razvoj, testiranje i isporuku ne uključuju samo programere, već i menadžere i naručitelje.

Nakon sâmog razvoja, testabilnost je drugi najvažniji zahtjev u razvoju softvera kod ekstremnog programiranja. Važna je mogućnost kreiranja automatiziranih funkcijskih i jediničnih testova. Čest je slučaj da su pojedine domene testiranja diskvalificirane zbog specifičnih zahtjeva. Tada je moguće promijeniti dizajn sustava kako bi se olakšalo testiranje.

XP metodologija razvoja softvera nije primjenjiva u svim projektima. XP je teško implementirati u slučajevima kada:

- Postoji jak otpor prema XP načinu programiranja. Teško je prihvatiti XP metodologiju ako proces koji se koristi funkcionira dobro [3] i nije ga potrebno mijenjati.
- U nekim projektima postoji zahtjev za generiranjem velike količine dokumentacije. XP je proces razvoja softvera a ne proces razvoja dokumentacije.
- U nekim okruženjima je prekovremeni rad uobičajena praksa. XP nije primjenjiv u takvim okruženjima.
- Ljudi u timu ponekad ne mogu ili ne žele međusobno komunicirati iz različitih razloga (objektivnih ili subjektivnih). XP nije primjenjiv u takvim okruženjima.

- Veliki timovi nisu prihvatljivi za XP. Najbolje rezultate XP postiže s manjim brojem ljudi, obično od 2 do 12.
- Okolina u kojoj se povratne informacije dugo čekaju nije pogodna za primjenu XP-a. XP očekuje povratne informacije vrlo rano u dizajnu i implementaciji kako bi se imalo vremena djelovati na razvoj sustava.

XP nije pogodan za određene klase produkata:

- sigurnosne
- dugog života
- modularne

Poglavlje 5

Postupci metodologije ekstremnog programiranja

Ovo poglavlje pobliže opisuje glavne prakse, odnosno temeljne karakteristike XP metodologije razvoja softvera koje su razvrstane u četiri tipične discipline u razvoju softvera:

1. planiranje razvoja (odjeljak 5.1)
2. dizajn softvera (odjeljak 5.2)
3. kodiranje softvera (implementacija) (odjeljak 5.3)
4. testiranje softvera (odjeljak 5.4).

Također, poglavlje opisuje životni ciklus XP projekta (odjeljak 5.5), XP projektne uloge (odjeljak 5.6) i slijed dnevnih aktivnosti u XP projektu (odjeljak 5.7).

5.1 Planiranje razvoja softvera u ekstremnom programiranju

Planiranje razvoja softvera uključuje akcije prikupljanja zahtjeva koji se oblikuju u korisničke priče (engl. *User Stories*) (odjeljak 5.1.1), planiranje isporuke na nivou čitavog projekta (odjeljak 5.1.2) te kreiranje plana iteracijâ za svaku pojedinu iteraciju (odjeljak 5.1.6) u iterativnom razvoju softvera (odjeljak 5.1.5).

Ovdje je naglašena i uključena važnost čestih i malenih isporuka (odjeljak 5.1.3) te važnost mjerenja brzine projekta (engl. *Project Velocity*) na vršenje planiranja (odjeljak 5.1.4).

Također, ističe se važnost dnevnih stojećih sastanaka (engl. *Daily Stand Up Meeting*) (odjeljak 5.1.8) te spomenuta praksa kretanja ljudi (engl. *Move people around*) u projektu kao mehanizam sprečavanja gubitka znanja (odjeljak 5.1.7).

5.1.1 Korisničke priče

Korisničke priče (engl. *User Stories*) služe istom cilju kao i slučajevi uporabe (engl. *Use Cases*), no zapravo nisu isto. Korištene su, u prvom redu, kako bi se kreirale vremenske procjene za planiranje isporuke softvera (engl. *Release Planing*) (odjeljak 5.1.2). Također, njihovo značenje je i zamjena velikih dokumenata za zahtjevima koje softver treba zadovoljavati.

Slučajevi uporabe koriste se za modeliranje zahtjeva koje sustav mora ispunjavati. Zahtjevi koje sustav mora ispunjavati se mogu podijeliti u dvije glavne skupine: funkcijski zahtjevi (engl. *functional requirements*) i zahtjevi koji se odnose na kvalitetu usluge (engl. *quality of service requirements*). Slučajevi uporabe definiraju ponašanje sustava ili dijela sustava te predstavljaju skup scenarija koji sustav izvodi kako bi postigao neki cilj. Scenarij je skup koraka koji se izvode za vrijeme interakcije između korisnika i sustava.

Funkcijski zahtjevi definiraju što će sustav raditi za korisnika. Kada se definiraju, sustav se obično predstavlja kao crna kutija (engl. *black box*) jer se samo gleda ponašanje sustava izvana.
Zahtjevi koji se odnose na kvalitetu usluge definiraju performanse, pouzdanost i sigurnost sustava. Nikad nisu samostalni, već se odnose na jedan ili više funkcijskih zahtjeva.

Primjena slučajeva uporabe ima za cilj modeliranje željenog ponašanja sustava, bez potrebe specificiranja načina implementacije takvog ponašanja. Tipičan proces modeliranja nekog sustava počinje detaljnom specifikacijom slučaja uporabe, a nakon toga se prelazi na realizaciju (definiranje klasâ od kojih se sastoji te potrebnih interakcijskih dijagrama).

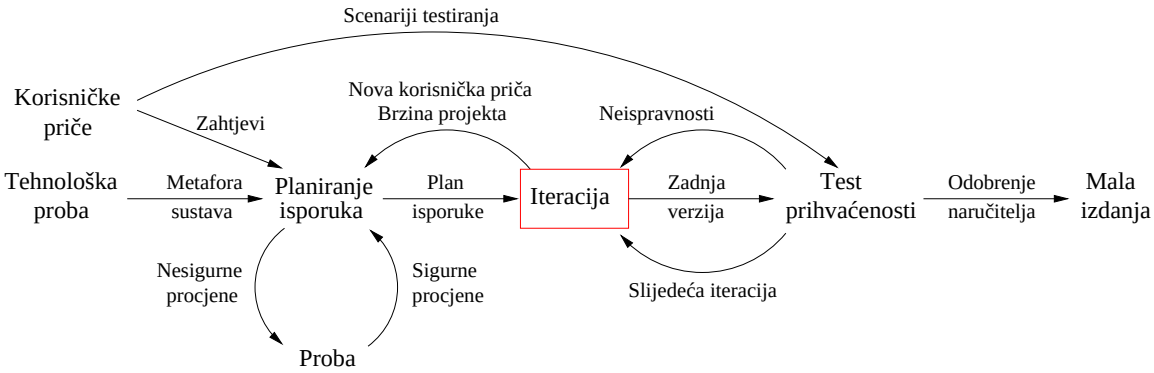
Korisničke priče piše naručitelj (engl. *customer*) softvera kao zahtjeve koje sustav (softverski sustav) treba zadovoljiti. Slične su korisničkom scenariju, jedino što nisu limitirane na opis korisničkog sučelja (engl. *user interface*). Obično su u formatu od oko tri rečenice teksta u terminologiji naručitelja bez tehničke sintakse. Dakle, po svojim karakteristikama, drugačije su od slučajeva uporabe.

Korisničke priče vode kreiranju tzv. testa prihvaćenosti softvera kojeg potvrđuje naručitelj (engl. *Acceptance Test*) (odjeljak 5.4.3) [28]. Potrebno je kreiranje jednog ili više automatiziranih testova prihvaćenosti softvera kako bi se verificirale korisničke priče, tj. provjerilo da li su korisničke priče ispravno implementirane.

Slika (5.1) prikazuje sudjelovanje korisničkih priča u obliku zahtjeva u izradi vremenske procjene za isporuku softvera (planiranje isporuke) te testa prihvaćenosti softvera kojeg potvrđuje naručitelj na temelju različitih scenarija testiranja.

Jedno od najvećih nerazumijevanja s korisničkim pričama je kako se priče razlikuju od tradicionalnih specifikacija zahtjeva, odnosno slučajeva uporabe. Najveća razlika je u stupnju detalja. Korisničke priče trebaju osigurati dovoljno detalja kako bi se načinila razumna procjena relativno niskog rizika koja prikazuje koliko dugo će trajati implementacija te korisničke priče. Kada dođe vrijeme za implementaciju priče, računalni programeri će doći do naručitelja i primiti detaljan opis zahtjeva.

Programer procjenjuje koliko dugo (vremenski) će trajati implementacija pojedine



Slika 5.1: Razvoj projekta u procesu ekstremnog programiranja

korisničke priče. Svaka korisnička priča će oduzeti jedan, dva ili tri tjedna procjene u "idealnom vremenu" razvoja. To idealno vrijeme razvoja zapravo znači koliko dugo treba za implementaciju korisničke priče u kôd ukoliko nema nekih prepreka, drugih obveza te je točno poznato što i kako treba činiti.
Razdoblje duže od tri tjedna znači da bi korisničku priču trebalo razdvojiti u više manjih dijelova. Razdoblje kraće od jednog tjedna znači premalu razinu detalja; treba kombinirati nekoliko korisničkih priča u jednu. Oko 80 korisničkih priča (plus/minus 20) je idealan broj za kreiranje plana isporuke softvera (odjeljak 5.1.2).
Ta procjena koju donosi programer je zapravo tehnička procjena. U kreiranju procjena sudjeluje i naručitelj.

Druga razlika između korisničkih priča (odjeljak 5.1.1) i dokumenta zahtjeva, odnosno slučajeva uporabe je fokus na potrebe naručitelja. Poželjno je pokušati izbjeći detalje specifične tehnologije, osnove baze podataka te algoritme. Poželjno je pokušati držati korisničke priče fokusirane na potrebama naručitelja kao suprotnost specifikaciji izgleda grafičkog sučelja.

5.1.2 Planiranje isporuke

Korisničke priče (odjeljak 5.1.1) služe kako bi se kreirao plan isporuke [29] koji vrijedi za čitav projekt. Plan isporuke se, kada je donešen, koristi za kreiranje plana iteracijâ za svaku pojedinu iteraciju (odjeljak 5.1.5) (slika 5.1).

Planiranje isporuke (engl. *Release Planning*) softvera sadrži skup pravila koje omogućuju svima uključenima u projekt da naprave vlastite odluke. Skup pravila definira metodu oko pregovora vremenskog plana kojeg svatko iz tima može ispuniti.

Bît planiranje isporuke za razvojni tim je idealna procjena trajanja svake korisničke priče u tjednima. Idealno, tjedan je vremensko trajanje implementacije korisničkih priča ako se apsolutno ništa drugo ne čini, jedino uključujući implementaciju jediničnih testova. Naručitelj tada odlučuje koje korisničke priče su najvažnije ili imaju najviši prioritet da bude završene, znajući koja je važnost pojedine priče za kupca.

Korisničke priče su obično napisane na kartama (komadima papira). Zajedno, računalni programeri i naručitelj kreiraju skupove priča koje će biti implementirane u prvoj (i slijedećim) isporukama. Programer donosi, kao što je već rečeno, tehničke procjene o potrebnom vremenu implementacije pojedine korisničke priče. Naručitelj određuje prioritet priča unutar iteracije (odjeljak 5.1.5).

Krajnji cilj je upotrebljiv, testabilan sustav koji će biti rano isporučen (čim je moguće ranije). Brzina projekta (odjeljak 5.1.4) je kreirana kako bi se odredilo koliko korisničkih priča može biti implementirano prije danog datuma (roka) ili koliko dugo traje implementacija skupa korisničkih priča (doseg). Kad se vrši vremensko planiranje, treba pomnožiti broj iteracija s brzinom projekta da bi se odredilo koliko korisničkih priča može biti završeno. Kad se vrši planiranje po dosegu, potrebno je podijeliti ukupan broj tjedana procijenjenih korisničkih priča s brzinom projekta kako bi se odredio ukupan broj iteracija do kraja.

Individualne iteracije su detaljno planirane prije početka svake iteracije i nikako ne unaprijed.

Kad je kreiran konačni plan isporuke (engl. *Final Release Plan*) i prosljeđen menadžmentu, često se pokušavaju promijeniti procjene korisničkih priča. Smanjenjivati procjene u ovom slučaju nije dobro niti poželjno. Procjene su valjane i smanjivanje trajanja procjena će vjerojatno prouzročiti probleme kasnije. Umjesto toga, može se pregovarati oko prihvatljivog plana isporuke sve dok se razvijatelji softvera, naručitelji i menadžeri slažu oko plana isporuke.

Slika (5.1) prikazuje postupak nastajanja planiranja isporuke (odjeljak 5.1.2). Na nastajanje planiranje isporuke utječu zahtjevi te metafora sustava (engl. *System Metaphor*) (odjeljak 5.2.2).
Nepouzdane i nesigurne procjene (engl. *Uncertain Estimates*) mogu biti popravljenje određenim tehnološkim probama (engl. *Spike*) (odjeljak 5.2.4). Sigurne, tj. pouzdane procjene (engl. *Confident Estimates*) ponovno utječu na planiranje isporuke.
Planiranje isporuke rezultira planom isporuke (engl. *Release Plan*) koji vrijedi za čitav projekt te se koristi za kreiranje plana iteracijâ (odjeljak 5.1.6) za svaku pojedinu iteraciju. Kreiranje novih korisničkih priča (engl. *New User Stories*) unutar pojedinih iteracija ponovno utječe na planiranje isporuke i donošenja novog plana isporuke.

5.1.2.1 Kvantificiranje projekta s četiri varijable

Osnovna filozofija planiranja isporuke je da projekt može biti kvantificiran s četiri varijable [29]: domet, resursi, vrijeme i kvaliteta.
Varijable predstavljaju dimenziju projekta.

Domet se odnosi na kvantitetu; tj. koliko toga treba načiniti. Resursi su u značenju broja dostupnih ljudi. Vrijeme se odnosi na trenutak kad su projekt ili isporuka gotovi. Kvaliteta je pokazatelj koliko je softver dobar i koliko dobro će biti testiran.

Gledajući općenito, potrebno je kontrolirati barem dvije od četiri varijable u bilo kojem projektu, kako bi se projekt odvijao pod nadzorom.
Menadžment može samo odabrati tri od četiri projektne varijable kako bi upravljao i

planirao, dok razvoj uzima preostalu četvrtu varijablu. Smanjenje kvalitete manje od izvrsno (engl. *excellent*) ima nepredvidljivi utjecaj na ostale tri varijable. Zbog toga, u osnovi, postoje samo tri varijable koje se zapravo žele mijenjati, izuzeći kvalitetu.

5.1.3 Male isporuke

Zadaća razvojnog tima je često isporučiti iterativne verzije sustava naručitelju. Kod planiranja isporuke je potrebno definirati malene jedinice funkcionalnosti koje su pogodne za isporuku i koje mogu biti isporučene u okružje naručitelja rano u projektu (u ranim fazama projekta).

Kritično je dobiti vrijednu i kvalitetnu povratnu informaciju od korisnika kako bi se na vrijeme imao bolji utjecaj na daljnji razvoj sustava. Cijena promjene softvera je veća u kasnijim fazama projekta (implementaciji, testiranju te isporuci) nego u početnim fazama projekta kada je malo toga implementirano. Što se više čeka isporuka važnih obilježja korisnicima, bit će manje vremena za njihov ispravak.

Slika (5.1) prikazuje nastajanje malenih isporuka (engl. *Small Releases*) nakon što je prošao test prihvaćenosti softvera.

5.1.4 Brzina projekta

Brzina projekta (engl. *Project Velocity*) je mjera koliko brzo posao može biti napravljen na projektu. Faktor radnog učinka (engl. *load factor*) je korišten kao mjera brzine u projektima sve do nedavno. No, brzina projekta je jednostavnija mjera za korištenje od faktora radnog učinka. Ako pomaže, može se koristiti faktor radnog učinka kako bi se kreirala početna procjena brzine projekta. Nakon toga se može koristiti brzina projekta umjesto faktora radnog učinka.

Za mjerenje brzine projekta, može se brojiti koliko korisničkih priča (odjeljak 5.1.1) ili koliko programerskih zadataka je završeno u iteraciji (odjeljak 5.1.5).

Za vrijeme kreiranja plana isporuke, brzina projekta u završenim korisničkim pričama može biti korištena kako bi se procijenilo koliko još korisničkih priča može biti dovršeno do nekog vremena. Za vrijeme kreiranja plana iteracijâ, dozvoljeno je da programeri potpisuju isti broj procijenjenih dana za obavljanje programerskih zadaća koji je jednak brzini projekta mjerenoj u prethodnoj iteraciji.

Brzina projekta se povećava tako što se dopušta programerima da pitaju naručitelja za neku drugu korisničku priču u slučaju da je njihov posao ranije završen.

Prilikom određivanja brzine projekta, očekuju se povećanja i smanjenja brzine projekta tijekom trajanja projekta (oscilacije u vrijednostima). Ako se brzina projekta dramatično mijenja od predviđene u više od jedne iteracije, potrebno je promijeniti projektni plan isporuke (odjeljak 5.1.2). Tijekom stavljanja sustava u produkciju, također se može očekivati mijenjanje brzine projekta zbog povećanih zadataka održavanja.

Dijeljenje brzine projekta s dužinom iteracije ili brojem programera nema baš smisla. Taj broj nije dobar pokazatelj pri uspoređivanju produktivnosti dvaju projekata jer svaki projektni tîm posjeduje različite sklonosti za procjenu priča i zadataka; svaki tîm procjenjuje različito (neki procjenjuju visoko a neki nisko) te svaki tîm ima ipak različit radni učinak. Praćenje cjelokupnog opsega posla obavljenog za vrijeme svake iteracije je ključ za ravnomjerno opterećen projekt.

Problem sa svakim projektom je kreiranje inicijalnih procjena. Sakupljanje mnoštva detalja ne čini inicijalnu promjenu ništa drugo nego pogađanjem. Briga oko korektne procjene cijelog projekta je iznad kreiranja opsežne dokumentacije.

Slika (5.1) prikazuje utjecaj brzine projekta na planiranje u projektu. Brzina projekta utječe na određivanje koliko će korisničkih priča biti implementirano u iteraciji. Kreiranje novih korisničkih priča mijenja planiranje isporuke na nivou čitavog projekta i utječe na donošenje novog plana isporuke.

Slika (5.2) prikazuje utjecaj brzine projekta iz slijedeće iteracije na planiranje iteracije. U razvoju na brzinu projekta utječu učenje i komunikacija. Kreiranje novih korisničkih priča u razvoju utječe na brzinu projekta.

5.1.5 Iterativni razvoj

Iterativni razvoj (engl. *Iterative Development*) povećava brzinu razvoja. Razvoj softvera kod XP-a se može podijeliti u raspored u oko desetak iteracija, a iteracija traje u prosjeku od jednog do tri tjedna. Poželjno je držati dužinu iteracija konstantnom tijekom čitavog projekta jer su iteracije zapravo žila kucavica projekta.

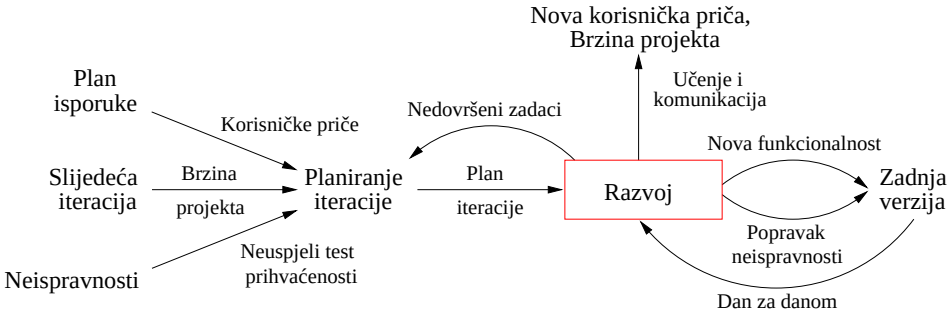
XP adresira i paralelni razvoj. Ako je razvojni projekt razmjerno velik, tj. u njemu sudjeluje veći broj osoba (npr. 30 do 40 programera), poželjno je podijeliti tim u dva ili više manjih timova. Svaki manji podtim (grupa) dobiva vlastite korisničke priče od naručitelja. Budući da je osnovna ideja podjela korisničkih priča na manje timove, u pravilu ne bi trebalo biti većih problema u sinkronizaciji među grupama [3] [29].

Nije dobro podijeliti programerske zadaće (kodiranje) unaprijed. Umjesto toga, potrebno je načiniti plan iteracijâ i planirati iteracije na početku svake od njih, kako bi se odredilo što će se činiti u trenutnoj iteraciji. Planiranje u trenutku (engl. *just-in-time planning*) je jednostavan način za praćenje promjena korisničkih zahtjeva u projektu.

Također, nije preporučeno gledanje unaprijed i implementiranje onoga što nije predviđeno u trenutnoj iteraciji. Bit će dosta vremena za implementiranje te funkcionalnosti kada ona postane dio jedne od slijedećih korisničkih priča u nekoj od slijedećih iteracija.

Važno je da se ozbiljno shvate rokovi predviđeni za implementaciju svake iteracije prateći napredak tijekom iteracije. U slučaju da je očito da se ne mogu završiti sve zadaće, potrebno je izvršiti novo planiranje iteracija, ponoviti procjene i smanjiti broj zadataka koji ulaze u iteracije.

Potrebno je koncentrirati napore na završavanje najvažnijih (najprioritetnijih) zadaća koje je odredio naručitelj, umjesto da je slučaj da postoji nekoliko nedovršenih zadataka koje su po svom nahođenju izabrali programeri.



Slika 5.2: Iteracija u ekstremnom programiranju

Ako se funkcionalnosti dodaju samo onda kada je određeno da budu implementirane, te ako se prakticira planiranje, tada se i lakše uhvatiti u koštac s promjenom korisničkih zahtjeva.

Slika (5.2) prikazuje razvoj iteracija u ekstremnom programiranju. U planiranje iteracija utječu korisničke priče koji su određene u planu isporuke, brzina projekta kao vrijednost koja se mijenja sa slijedećim iteracijama, a ovisi o prethodnim iteracijama te neispravnost testa prihvaćenosti softvera (engl. *Failed Acceptance Test*).
Usvojeni plan iteracija vodi u fazu razvoja. Problemi u fazi razvoja koji imaju za posljedicu nedovršene zadaće (engl. *Unfinished Tasks*) vode novom planiranju iteracija.

5.1.6 Planiranje iteracija

Planiranje iteracije (engl. *Iteration Planning*) se vrši na početku svake iteracije te ima cilj proizvesti plan izvršavanja zadaća koje će se izvršavati u toj iteraciji. Obično je svaka iteracija u dužini od jednog do tri tjedna. Korisničke priče (odjeljak 5.1.1) naručitelj odabire za trenutnu iteraciju sukladno planu isporuke (odjeljak 5.1.2) u poretku kojeg sâm vrednuje. Neispravnost testa prihvaćenosti softvera kojeg potvrđuje naručitelj, a koji treba biti popravljen, je također povezan s planiranjem iteracije [28]. Naručitelj odabire korisničke priče zajedno s procjenama o brzini projekta iz prethodne iteracije.

Korisničke priče i testovi koji nisu prošli provjeru ispravnosti se ubacuju kao programerske zadaće u obliku novih korisničkih priča. Dok su korisničke priče u jeziku naručitelja, zadaci su u jeziku programera. Dupliciranje zadaci mogu biti maknuti iz iteracije.

Razvijatelji potpisuju zadatke i procjenjuju koliko je potrebno da bi se ti zadaci kompletirali. Važno je za razvijatelje koji prihvaćaju zadatke da oni budu ti koji će procijeniti kraj zadatka. Unutar XP-a ne preporuča se zamjena ljudi koji sudjeluju na razvoju. Određena osoba koja će implementirati zadatak mora procijeniti koliko dugo će trajati implementacija.

Svaka zadaća treba biti u trajanju od jednog do tri idealna programerska dana. Idealni programerski dan znači koliko dugo je potrebno da bi se završio zadatak u slučaju da nema nikakvih smetnji (dodatnih zadataka) sa strane. Zadaci koji su kraći od jednog dana

mogu se zajedno grupirati. Zadaci koji su pak duži od tri dana trebaju biti razbijeni u više manjih zadataka.

Brzina projekta (odjeljak 5.1.4) je korištena kako bi se odredilo da li je iteracija preopterećena i rezervirana ili nije. Cjelokupna vremenska procjena u idealnim programerskim danima ne bi smjela premašiti brzinu projekta iz prethodne iteracije. Ako iteracija sadrži previše korisničkih priča, naručitelj treba odabrati priče (na osnovi prioriteta) koji moraju biti maknute sve do slijedeće iteracije.

U slučaju da iteracija sadrži premalo korisničkih priča, tada ih može biti prihvaćeno još. Brzina izražena u danima potrebnim za implementaciju zadataka je s vremenom vjerniji podatak od brzine izražene u tjednima potrebnim za implementaciju korisničkih priča.

Često je alarmantno vidjeti korisničke priče koje su propterećene time što su preopširne za implementaciju. Treba imati u vidu važnost jediničnog testiranja i tehnike refaktoriranja (engl. *Refactoring*) (odjeljak 5.2.6). Unutar XP-a se naglašava da nedovoljno posvećivanje pažnje tim područjima će uzrokovati usporavanje u projektu. Prema tome, poželjno je izbjegavati dodavanje novih funkcionalnosti prije nego su te funkcionalnosti po rasporedu za implementaciju, inače nastupaju zakašnjenja u projektu.

Kao što je već ranije rečeno, nije pametno mijenjati procjene zadataka i korisničkih priča. Proces planiranja leži na realnosti dosljednih procjena, mijenjanje (smanjivanje) tih procjena uzrokuje kasnije više problema.

Pažnju, dakle, treba posvetiti brzini projekta te mogućoj preopterećenosti korisničkih priča. Možda je potrebno načiniti ponovnu procjenu i pregovaranje oko plana isporuke svakih tri do pet iteracija, što je i uobičajeno.

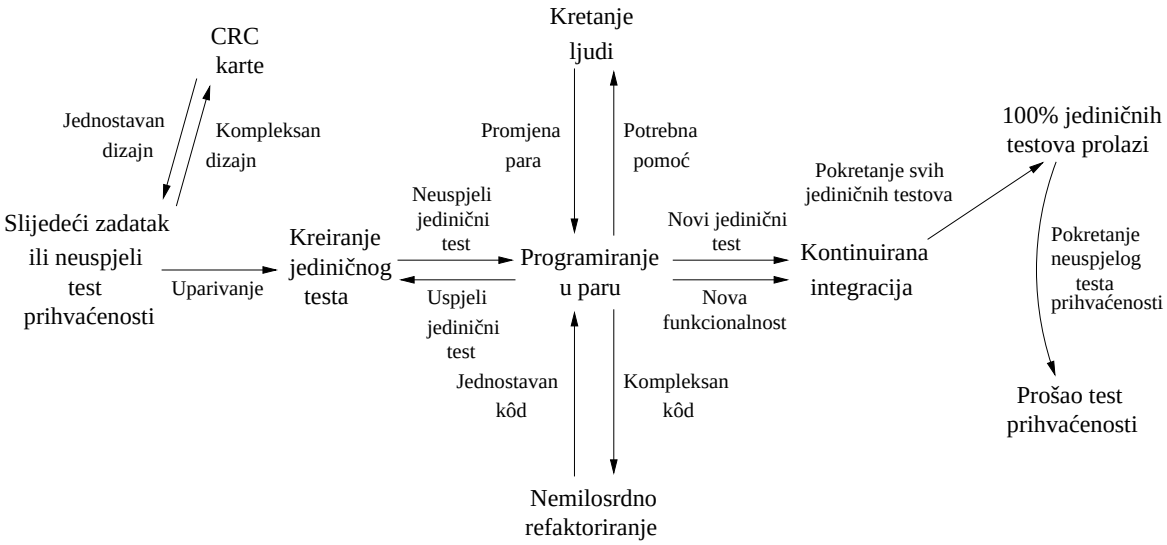
5.1.7 Kretanje ljudi

Kretanjem ljudi (engl. *Move people around*) se može spriječiti ozbiljan gubitak znanja i uska grla u kodiranju. Ako samo jedna osoba u projektnom razvojnom timu može raditi na zadanom području i ta osoba je nedostupna, napredak projekta će biti veoma spor.

Međusobno treniranje ljudi je često važna činjenica za razmatranje u kompanijama koje nastoje izbjeći tzv. "otoke znanja" koji su osjetljivi na gubljenje. Kretanje ljudskih resursa u obavljanju različitih zadataka u kombinaciji s programiranjem u pâru (engl. *Pair Programming*) (odjeljak 5.3.4) ima efekt dodatnog treninga. Umjesto jedne osobe koja znade sve oko određenog dijela kôda ili tehnologije, svi u timu znadu mnogo o svim dijelovima.

Tîm je fleksibilniji ako svatko zna dovoljno raditi na svakom dijelu sustava. To je naročito izraženo u malim grupama. Umjesto da postoji nekoliko ljudi prezauzetih s poslom dok ostali članovi tima imaju manje posla, cijeli tim može biti produktivan. Bilo koji broj programera može biti dodijeljen trenutno najvažnijim dijelovima sustava.

Dobra praksa je jednostavno ohrabrivati svakog pojedinca da pokušava djelovati na novom dijelu sustava i to na barem nekom dijelu svake iteracije. Tehnika programiranja u pâru to čini mogućim bez gubitka produktivnosti te omogućuje kontinuiranost. Jedna



Slika 5.3: Zajedničko vlasništvo kôda

osoba iz pâra može biti zamijenjena dok druga nastavlja s novim partnerom. Ovo je izvrstan način dijeljenja znanja u timu.

Slika (5.3) prikazuje kretanje ljudi koje je izravno povezano s tehnikom programiranja u pâru. U slučaju da je potrebna pomoć, dolazi do promjene pâra i stvaranja novog.

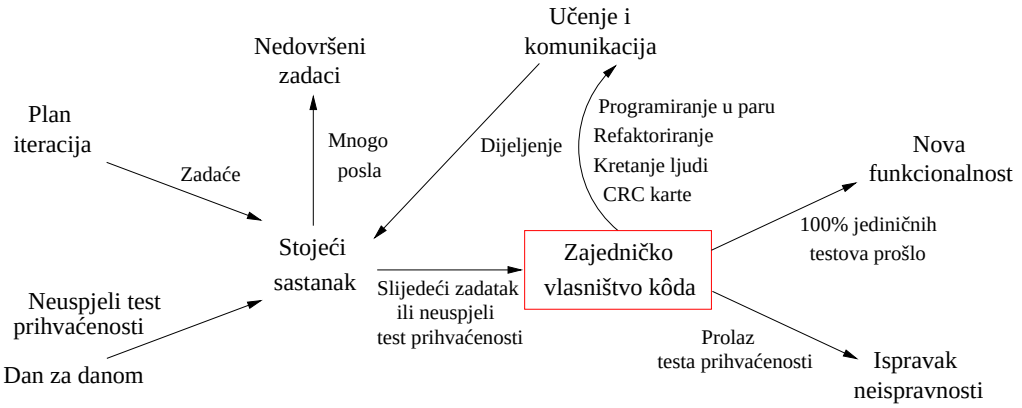
5.1.8 Dnevni stojeći sastanci

Na tipičnom projektnom sastanku većina osoba obično aktivno ne sudjeluje već samo sluša zaključke. Veliki dio vremena računalnih programera je potrošen na korištenje trivijalne količine komunikacije. U slučaju da mnogo osoba prisustvuje svim sastancima, to ima za uzrok iscrpljivanje projektnih resursa te stvaranje projektne "noćne more".

Komunikacija među cijelim timom je cilj stojećih sastanaka. Stojeći sastanci (engl. *Stand Up Meeting*) svakog jutra služe kako bi se komunicirali problemi i rješenja te unaprijedio timski fokus. Svi stoje "u krugu" kako bi se izbjegle duge diskusije. Sastanak se i zove "stojeći" kako bi se osigurala njegova kratkoća (desetak minuta) a da sastanak ne preraste u neku dugačku raspravu. Efikasnije je da postoji kratak sastanak kome su svi obvezni prisustvovati, nego mnogo sastanaka sa dijelom ljudi.

Kada se koriste dnevni stojeći sastanci, auditorij bilo kojeg drugog sastanka može biti odabran prema tome tko treba sudjelovati. Uz održavanje dnevnih stojećih sastanaka, moguće je ne koristiti čak većinu ostalih sastanaka.
S limitiranim auditorijem, većina sastanaka može biti spontano zamijenjena pred računalima gdje se može pregledavati kôd i rješavati probleme.

Dnevni stojeći sastanak može zamijeniti mnogo drugih sastanaka, pružajući čistu štednju nekoliko puta svojom vlastitom dužinom.



Slika 5.4: Razvoj (engl. *Development*) u Ekstremnom programiranju

Slika (5.4) prikazuje tipične aktivnosti dnevnih stojećih sastanaka. Prema planu iteracija, najavljuju se zadaci za tekući dan. Dobiva se izvještaj o testovima prihvaćenosti softvera, a koji nisu prošli. Također se izvještava ako je od prethodnog dana ostalo nezavršenih zadataka te ako je bilo previše toga za testirati, implementirati i refaktorirati. Tim dobiva nove zadatke ili popravak stvari iz prethodnog dana. Vrijedi pravilo zajedničkog vlasništva nad kôdom. Zajedničko vlasništvo nad kôdom obuhvaća programiranju u pâru, "nemilosrdno" refaktoriranje i kretanje ljudi unutar tima, čime se postiže učenje i komunikacija.

5.1.9 Popravak XP-a

Proces razvoja softvera koji koristi XP metodologiju treba biti prepravljen u slučaju kada se uočilo da napredak na projektu ne ide očekivanim tokom i da dosadašnje korištene prakse nisu dovoljno učinkovite. Ne kaže se izričito *ako* već *kada* jer je pretpostavka da će biti potrebno načiniti neke promjene kako bi se proces prilagodio specifičnosti trenutnog projekta.
Potrebno je slijediti temeljna pravila, odnosno postupke ekstremnog programiranja kako bi se započelo s promjenama. No, ne treba oklijevati s mijenjanjem onoga što ne radi.

Pravila trebaju biti slijeđena sve dok ih tîm ne odluči promijeniti. Svi razvijatelji (računalni programeri) trebaju znati točno što trebaju očekivati jedan od drugoga imajući skup pravila koji je jedini način za postavljanje tih očekivanja.

5.2 Dizajn softvera u ekstremnom programiranju

Dizajn softvera u XP-u uključuje korištenje jednostavnog dizajna (odjeljak 5.2.1), korištenje metafore kao pojednostavljene slike sustava koje je u razvoju (odjeljak 5.2.2), korištenje CRC karata za timski dizajn sustava (odjeljak 5.2.3), korištenje rješenja "šiljka" (tj. tehnoloških proba) (odjeljak 5.2.4).

Naglašena je važnost izbjegavanja dodavanja nepotrebne funkcionalnosti prije nego što je ta funkcionalnost zaista nužna ili dogovorena za implementaciju (odjeljak 5.2.5) te

važnost upotrebe prakse refaktoriranja kao metode koja obuhvaća akcije poboljšanja kôda (odjeljak 5.2.6).

5.2.1 Jednostavnost

Jednostavan dizajn uvijek zahtijeva manje vremena za završetak nego kompleksni. Tako je uvijek potrebno činiti najjednostavniju stvar (tj. najjednostavniju moguću) koja može funkcionirati. Ako se nađe neka kompleksna materija u implementaciji, poželjno je da se ona zamijeni s jednostavnijom. Jednostavan dizajn je jedan općeniti princip koji vrijedi i za ostale razvojne metodologije.

Uvijek je brže i jeftinije zamijeniti kompleksan kôd jednostavnijim prije nego što se izgubi previše vremena na njega. Razlog tome je lakše održavanje sâmog kôda, članovi projekta se lakše snalaze te je lakše vršiti implementaciju novih funkcionalnosti.

Stvari je potrebno držati čim jednostavnijima što je duže moguće, tako da se nova funkcionalnost ne dodaje dok to nije propisano iteracijom.

Potrebno je biti svjestan da je vođenje jednostavnog dizajna zapravo težak posao.

5.2.2 Metafora sustava

Metafora sustava (engl. *System Metaphor*) predstavlja pojednostavljenu sliku sustava koji je u razvoju, tj. u implementaciji. Važno je da ta slika sustava bude čim više pojednostavljena kako bi je svi razumjeli.

Glavna ideja ove pojednostavljene slike je da programeri koji sudjeluju u razvoju imaju jasnu sliku gdje se njihov dio nalazi u sustavu, tj. kako se njihov dio uklapa u sustav. Metafora pojednostavljenom slikom pomaže modeliranje sustava [24]. Korištenjem metafore, programeri imaju i pojednostavljen pregled čitavog sustava te im je moguće imati pojednostavljenu sliku cjelokupne funkcionalnosti.

Metafora definira zajednički jezik, tj. terminologiju koja se upotrebljava na projektu i koju svi razumiju.

Potrebno je odabrati metaforu sustava kako bi tîm konzistentno davao imena klasama i metodama. Davanje imena objektima je vrlo važno za razumijevanje cjelokupnog dizajna sustava i tehniku ponovnog iskorištavanja kôda. Velika ušteda vremena je znati kako bi nešto moglo biti imenovano u sustavu. Nužno je odabrati imena objektima koji svi u timu mogu povezati sa specifičnim dijelom sustava.

Slika (5.1) prikazuje utjecaj arhitekturalnog "šiljka" (tj. određene tehnološke probe) (engl. *Architectural Spike*) metaforom sustava (engl. *System Metaphor*) na planiranje isporuke (engl. *Release Planning*).

5.2.3 CRC karte

CRC karte (engl. *Class, Responsibilities, Collaboration cards*) (karte klasâ, odgovornosti i suradnje) je potrebno koristiti za timski dizajn sustava. CRC karte omogućuju

cjelokupnom projektnom timu da se usredotoče na dizajn. Što je više ljudi koji mogu pomoći u dizajnu sustava, bit će veći broj uključenih dobrih ideja.

Individualne CRC karte se koriste kako bi predstavljale objekte. Klasa objekta može biti predstavljena na vrhu karte, odgovornosti prikazane na donjoj lijevoj strani a suradnja među klasama desno od svake odgovornosti.

CRC sesija se odvija s nekom osobom koja simulira sustav, govoreći koji objekti šalju poruke kojim objektima. Prolazeći kroz sustav, rano se otkrivaju slabosti i problemi. Alternative dizajna mogu biti brzo istražene simulirajući predloženi dizajn.

Jedna od najvećih kritika CRC karti je nedostatak napisanog dizajna. To obično nije potrebno ako CRC karte čine dizajn očitim. Ako je potreban zapis, treba biti dokumentirana jedna karta za svaku klasu i sačuvati je kao dokumentaciju.

CRC karte mogu biti smatrane strateškim nivoom dizajna, programiranje u pâru (odjeljak 5.3.4) taktičnim nivoom dok refaktoriranje (odjeljak 5.2.6) služi i kao strateški i kao taktički nivo dizajna.

Slika (5.3) prikazuje utjecaj CRC karata koje od složenog dizajna čine jednostavan dizajn pomoću timskog dizajna sustava.

5.2.4 Rješenje tehnoloških probi

Rješenje tehnoloških probi (engl. *Spike Solution*) pomaže u pronalaženju odgovora na teške probleme iz tehnike ili dizajna.

To rješenje je zapravo vrlo jednostavan program koji pomaže u otkrivanju potencijalnog rješenja za problem s kojim se razvojni tim ili pâr susreće. Potrebno je izgraditi sustav, tj. skup jednostavnih programa koji samo adresiraju problem koji se istražuje te ignoriraju sve ostale poslove.

Većina tehnoloških proba nisu dovoljno dobre kako bi se zadržale u sustavu, tako da je opće očekivanje da se odbace [30].

Njihova osnovna namjena je samo proba da li određeno rješenje radi, a ne cjelokupno rješenje dizajna. Cilj je, dakle, ili smanjenje rizika tehničkog problema ili povećanje pouzdanosti procjena korisničkih priča (odjeljak 5.1.1).

Kada tehnička poteškoća prijeti razvoju sustava, poželjno je uključiti pâr programera da rješavaju problem.

Slika (5.1) prikazuje utjecaj procjena na planiranje isporuke i stvaranje "šiljaka". Nesigurne procjene u planiranju isporuka utječu na potrebu stvaranja proba. U stanju tehnološke probe provjerene procjene utječu na kreiranje plana isporuke.

5.2.5 Izbjegavanje dodavanja funkcionalnosti

Poželjno je držati sustav neopterećen s viškom funkcionalnosti za koju se pogađa da će se kasnije koristiti. Samo oko 10% od te cjelokupne dodatne funkcionalnosti će se

ikada koristiti, tako da se zapravo gubi 90% vremena.

Svi su pod pritiskom da dodaju funkcionalnost radije sada nego kasnije iz jednostavnog razloga da će to poboljšati sustav. Čini se da je brže da se ta funkcionalnost doda upravo sada. Potrebno je neprekidno se podsjećati da je ta dodatna funkcionalnost zapravo nepotrebna.

Dodatna funkcionalnost će uvijek usporiti implementaciju i nepotrebno tratiti resurse. Potrebno je da je tîm usmjeren prema sadašnjim i budućim zahtjevima i fleksibilnosti, te koncentriran na implementaciju trenutnih zadataka.

5.2.6 Refaktoriranje

Računalni programeri se zadržavaju na dizajnu softvera i dugo nakon što je softver stvoren. Praksa je da se nastavlja s korištenjem te s ponovnim korištenjem kôda (engl. *Code Reuse*) koji dugo nije održavan jer još uvijek taj kôd nekako radi i strah ga je mijenjati kako se ne bi nešto poremetilo. Postavlja se pitanje da li je ekonomski isplativo tako činiti. Ekstremno programiranje metodologija razvoja softvera daje odgovor na ovu problematiku.

Refaktoriranje (engl. *Refactoring*) je tehnika poboljšavanja kôda bez promjene funkcionalnosti, tj. vanjskog ponašanja kôda [31]. To je discipliniran način čišćenja kôda koji minimizira šanse uvođenja neispravnosti.

Tehnika refaktoriranja uključuje uklanjanje redundancija kôda, eliminiranje nekorištene funkcionalnosti te pomlađivanje zastarjelog dizajna što kôd čini lakšim za razumijevanje. Refaktoriranje tijekom čitavog životnog ciklusa razvojnog projekta štedi vrijeme i povećava kvalitetu.

Refaktoriranje nije pisanje kôda iz početka. Moguće su situacije kada je sigurnije i bolje početi iz početka; bolje je poboljšati postojeći kôd nego ući u rizik ponovnog pisanja kôda [32].

Refaktoriranje se provodi kako bi se dizajn održavao čistim te se izbjegla nepotrebna složenost. Održavanje čistog i konciznog kôda omogućuje lakše razumijevanje, modificiranje te proširivanje kôda. Treba biti siguran kako je sve izraženo jednom i samo jednom. U konačnici, potrebno je manje vremena za proizvodnju željenog sustava.

Tehnika refaktoriranja se provodi tijekom čitavog životnog ciklusa projekta: prije i nakon implementacije novog kôda. Prije implementacije nove funkcionalnosti, kôd se mijenja kako bi implementacija nove funkcionalnosti bila jednostavnija. Nakon implementacije nove funkcionalnosti, pokušava se pojednostavniti kôd koji je napisan. Nije poželjno istovremeno refaktorirati i pisati kôd za novu funkcionalnost. Kako bi se provjerilo da li je refaktoriranje promijenilo prethodno implementiranu i testiranu funkcionalnost, potrebno je neprekidno izvršavati jedinične testove.

Potreba za refaktoriranjem raste sa starošću arhitekture te s novim funkcionalnostima koje korisnici traže. Nove funkcionalnosti mogu biti dodane bilo kada u kôd. Općenito, te su nove funkcionalnosti povezane i često je bolje razviti arhitekturu koja ima mogućnost lakšeg prihvaćanja tih novih funkcionalnosti. Tu se često javlja potreba za refaktoriranjem kako bi se uklonio dvostruki kôd prilikom implementacije novih funkcionalnosti [33].

Upotreba refaktoriranja može u programski kôd unijeti nove greške i uzrokovati brojne

probleme.
Jedan od problema povezanih s refaktoriranjem je baza podataka. Mnoge poslovne aplikacije su povezane sa shemama baze podataka. Baze podataka je teško mijenjati. Drugi problem je migracija podataka. Iako je sustav pažljivo dizajniran kako bi se minimizarale ovisnosti između shema baze podataka i objektnog modela, promjena sheme znači migraciju podataka, što može biti dug i problematičan zadatak.
Slijedeći problem je promjena sučelja. Važna činjenica objekata je da je moguća promjena implementacije softverskog modula neovisno od promjene sučelja. Unutrašnjost objekta se može sigurno mijenjati, dok sa promjenom sučelja treba biti naročito oprezan, pogotovo ako je sučelje u upotrebi kod klijenata.
Također, neke dizajnerske i arhitekturalne odluke su toliko centralne da ih se ne može refaktorirati. [31]

Slika (5.3) prikazuje da se primjenjivanjem postupka programiranja u pâru i tehnike "nemilosrdnog" refaktoriranja složen kôd može zamijeniti s jednostavnim kôdom.

5.3 Kodiranje softvera u ekstremnom programiranju

Kodiranje softvera u XP-u naglašava važnost prisutnosti i dostupnosti naručitelja razvojnom timu (odjeljak 5.3.1), važnost postojanja standarda (tj. dogovora) pisanja kôda (odjeljak 5.3.2), važnost implementacije testova prije implementacije sâmog kôda (odjeljak 5.3.3), opis tehnike programiranja u pâru (odjeljak 5.3.4) te provođenje kontinuirane integracije kôda (odjeljak 5.3.5).

Ističe se politika zajedničkog vlasništva nad kôdom (odjeljak 5.3.6), provođenje optimizacije kôda na kraju (engl. *Optimize Last*) (odjeljak 5.3.7) te prakticiranje 40-satnog radnog tjedna, bez prekovremenog (engl. *40 Hours Week, No Overtime*) (odjeljak 5.3.8).

5.3.1 Stalna prisutnost naručitelja u razvoju

Jedan od nekoliko zahtjeva XP-a je prisutnost, odnosno dostupnost naručitelja, ne samo kako bi pomogao razvojnom timu u radu, već kako bi on bio dio razvojnog tima.

Sve faze XP projekta zahtijevaju komunikaciju s naručiteljem, često licem u lice. Najlakše je stoga stalna prisutnost, odnosno dostupnost naručitelja razvojnom timu, pogotovo ako je projekt zamjetne veličine i složenosti.

Postoje dva osnovna načina kako naručitelj može sudjelovati u timu:

- naručitelj nešto radi u projektu (ima dodijeljene zadatke)
- naručitelj pomaže timu u izradi softvera odgovarajući na nejasnoće u trenutku kada su nastupili problemi.

Stalna prisutnost naručitelja u projektu može rezultirati određenim problemima u projektu, naročito ako je naručitelj uključen u tehnički dio implementacije funkcionalnosti ili nije u mogućnosti vjerno opisati što je sve potrebno prije nego što je kôd napisan.

Korisničke priče (odjeljak 5.1.1) i njihov opis stvara naručitelj s pomoći razvojnog tima čime nastaju vremenske procjene i dodjeljuju se prioriteti. Svatko od njih se drži svojih obveza u planiranju:

- Programer procjenjuje vrijeme koje je potrebno za implementaciju pojedine korisničke priče, te određuje raspored implementacije pojedine korisničke priče unutar pojedine iteracije.
- Naručitelj određuje datum izdavanja verzije te važnost pojedine korisniče priče za kupca. Na osnovu važnosti pojedine korisničke priče određuje prioritet korisničkih priča unutar pojedine iteracije.

Planiranje verzija definira malene inkrementalne isporuke koje omogućuju ranu isporuku određene funkcionalnosti naručitelju. Konstantno isporučivanje verzija omogućuje korisnicima da čim prije mogu vidjeti kako sustav ili određena funkcionalnost sustava radi. Kritično je dobivanje povratne informacije od korisnika kako bi se na vrijeme moglo djelovati na razvoj sustava. Važno je da ta povratna informacija bude pravovremena.

Unatoč svemu, naručitelj može prekinuti dizajn u slučaju da je nezadovoljan s funkcionalnošću ili s napretkom sustava. U tom slučaju, nema veće štete jer tim nije izradio nikakav višak funkcionalnosti za koju nije bio plaćen.

Naručitelj je potreban kako bi pomogao u izradi funkcijskog testiranja, odnosno kreiranju testova prihvaćenosti softvera (odjeljak 5.4.3). Potrebno je kreirati testne podatke i provjeriti (verificirati) rezultate. Funkcijski testovi verificiraju da je sustav ili dio sustava spreman da bi bio isporučen. Naručitelj izravno sudjeluje u kreiranju testova prihvaćenosti softvera koji provjeravaju funkcionalnost čitavog sustava. U kreiranju testova mu u tehničkoj problematici pomažu programeri.

Može se desiti da sustav ne prolazi sve funkcijske testove prije isporuke. U tom slučaju, naručitelj treba na osnovu rezultata testova ili odobriti da sustav ide u isporuku ili zaustaviti isporuku sustava kako bi se ispravile greške.

5.3.2 Standardi pisanja kôda

Programeri u razvojnom timu su često u prilici da gledaju i proučavaju kôd koji su drugi iz tima napisali. Potencijalni problem je što svaki programer ima svoj vlastiti stil kodiranja.
Važno je da postoji sporazum, odnosno pravila ili standard kako pisati kôd (engl. *Coding Standars*). U protivnom, troši se previše vremena na razumijevanje tuđeg kôda.

Standardi kodiranja ne smiju oduzimati previše programerskog vremena. Ako se članovi XP tima ne mogu dogovoriti oko pravila pisanja kôda, najbolje je preuzeti neki standard propisan od neutralne organizacije.
Takav standard može biti u vidu dogovora ili u obliku pisanog dokumenta ako je većeg opsega.

5.3.3 Kodiranje testova za jedinice programa prije implementacije funkcionalnosti

Kada se prvo vrši implementacija testova (prije implementacije određene funkcionalnosti), kreiranje sâmog kôda će biti mnogo lakše i brže. Vrijeme koje je potrebno da se implementira prvo test a zatim kôd koji zadovoljava test je slično vremenu koje je potrebno za implementiranje funkcionalnosti bez implementacije testa. Implementacija testa, dakle, ne odnosi neko dodatno vrijeme, no čini implementaciju funkcionalnosti za koju se piše test dosta lakšom. Ako već postoji jedinični test, nije ga potrebno kreirati nakon implementacije funkcionalnosti, čime se štedi vrijeme.

Kreiranje jediničnog testa pomaže programerima da zaista razmotre što zapravo treba biti implementirano (što je zahtjev) bez implementacije onoga što nije nužno ili što nije izričiti zahtjev. Testovi prate zahtjeve sustava. Ne može biti nesporazuma u specifikaciji napisanoj u obliku izvršnog kôda.

Korištenjem jediničnih testova dobiva se odmah povratna informacija za vrijeme rada, tj. implementacije kôda. Često nije jasno kada je programer završio, tj. implementirao svu zahtijevanu funkcionalnost. Kada se kreiraju jedinični testovi, točno se zna trenutak kada je implementacija funkcionalnosti završena time što je pokretanje testa bilo uspješno, tj. svi jedinični testovi su prošli.

Pristupom implementacije testova prije sâmog kôda vrši se veliki utjecaj na dizajn sustava.
Često je vrlo teško vršiti testiranje nekih softverskih sustava. Kod takvih sustava se prvo čini klasična implementacija funkcionalnosti pa se tek onda implementiraju testovi u vidu jediničnih testova. Implementaciju testova ponekad vrši i drugi tîm ljudi.
Implementacijom testova prije kôda je diktirano da se testira sve što je kritično i od važnosti za naručitelja.

Postoji ritam u razvoju softvera koji nalaže da se vrši implementacija jediničnih testova prije implementacije sâmog kôda. Taj pristup je općenito poznat kao razvoj diktiran testiranjem (engl. *Test Driven Development*, TDD) [34].
Prvo se implementira jedan test (jedna ili više testnih metoda) koji definira jedan mali aspekt problema. Za taj test (ili testove) se implementira najjednostavniji kôd koji će omogućiti da prethodno implementirani test prođe bez grešaka. Prilikom dodavanja nove funkcionalnosti, prethodno se prema potrebi refaktorira prethodni kôd. Test se neprestano izvodi kako bi se provjerilo da ta nova funkcionalnost radi, te da se refaktoriranjem nije narušila neka prethodna funkcionalnost. Ciklički se ponavlja dodavanje testnih metoda, refaktoriranje kôda i dodavanje nove funkcionalnosti sve dok svi zahtjevi nisu pretočeni u testove i kôd.
Mogući su određeni problemi kada isti programeri koji pišu kôd pišu i jedinične testove za taj kôd. Moguća je situacija da programeri proizvedu određene neispravnosti i pri tome sa jediničnim testovima ne detektiraju te neispravnosti.

Korištenjem TDD tehnike obično brže nastaje kôd. Zabilježeno je da je takav kôd često jednostavniji nego kada se ne primjenjuje ova tehnika. Implementirano je samo ono što je nužno i specificirano zahtjevima. Pokretanjem testova se odmah dobiva povratna

informacija da novi kôd radi. Tim pristupom se povećava sigurnost programera u vlastiti kôd, te se omogućuje jednostavno razumijevanje i refaktoriranje. Ostali programeri mogu vidjeti kako se taj novi kôd koristi uvidom u testove. Testovi su ujedno i svojevrsna dokumentacija kôda: pokazuju kako se kôd ispravno koristi. Ovim pristupom se dobiva izvrsna pokrivenost kôda ako se kôd provjerava određenim alatima za testiranje.

Tipični alati, odnosno okružja kojima je moguće vršiti jedinično testiranje su *JUnit* [35] za J2SE [36] ili J2EE [37] platformu i *csUnit* [38] za *Microsoft .NET* platformu [39] [30]. Pomoću ovih alata, moguće je automatizirano izvršavati jedinične testove iz IDE razvojnog okružja.

Slika (5.3) prikazuje da se programiranjem u pâru (odjeljak 5.3.4) vrši implementacija testova. Ako je jedinični test prošao za određenu funkcionalnost, može se tehnikom programiranja u pâru kreirati novi jedinični test za novu funkcionalnost koja se želi implementirati. Ako jedinični test nije prošao, potrebno je ispraviti pogrešno ili nepotpuno implementiranu funkcionalnost. Prilikom implementacije novih testova i nove funkcionalnosti, potrebno je neprekidno izvršavati integraciju.

5.3.4 Programiranje u pâru

Sav kôd koji će biti uključen u produkciju je kreiran od dva programera koji rade u pâru za jednim računalom (engl. *Pair Programming*).

Programiranjem u pâru se povećava kvaliteta softvera bez utjecaja na vrijeme isporuke. Neka istraživanja su pokazala da kôd koji nastaje od pâra programera je znatno više kvalitete od kôda koji ne radi pâr, dakle nastaje od pojedinca [40]. Broj evidentiranih neispravnosti je obično manji kod programiranja u pâru jer dva programera obično brže uoče pogreške nego što ih uoči samo jedan programer.

Dva programera koji rade u pâru za jednim računalom mogu implementirati isto toliko (ili čak više) funkcionalnosti kao i dva programera koji rade svaki za sebe. Dakle, produktivnost pâra je ista ili veća nego produktivnost svakog programera pojedinačno.

Oba programera u pâru povećavaju svoju kompetenciju čime se omogućuje proces dijeljenja znanja. Oba programera su uključena u trenutnu problematiku, odlučivanje i jedinično testiranje.

Razlikujemo dvije uloge, odnosno role koje se međusobno izmjenjuju:

1. Programer koji koristi tipkovnicu računala i piše kôd (engl. *Driver*).
2. Programer koji provjerava napisani kôd (engl. *Navigator*).

Obje osobe sjede ispred monitora i dijele tipkovnicu računala i miša. Jedna osoba tipka (koristi tipkovnicu) i razmišlja taktički o kôdu (metodi ili klasi) koja nastaje. Druga osoba razmišlja strateški kako se ta metoda uklapa u klasu.

Dobra praksa je kruženje ljudi iz tima unutar parova kako bi se proširile kompetencije i iskustvo svih članova XP projektnog tima. Poželjno je da osobe u parovima budu slične razine kompetencija i iskustva kako bi se izbjeglo da se u pâru jedna osoba "odmara" dok druga radi [41].

Slika (5.3) prikazuje programiranje u pâru u XP metodologiji razvoja softvera.
U slučaju da pâr koji vrši implementaciju treba pomoć, dolazi do kretanja ljudi u XP timu te do promjene pâra.

5.3.5 Kontinuirana integracija kôda

Bez kontroliranja izvornog kôda, programeri koji vrše integraciju vjeruju da je sve u redu. Zbog paralelne integracije izvornog kôda modula, može postojati kombinacija kôda koja nije prije međusobno testirana.
Zbog toga često nastaje velik broj problema bez pravovremene detekcije.

Slijedeći problemi se pojavljuju kada ne postoji jasan rez posljednje verzije koja se isporučuje naručitelju. To se ne odnosi samo na izvorni kôd, već i na kolekciju jediničnih testova koji moraju verificirati ispravnost izvornog kôda. Ako se ne mogu osigurati kompletni, ispravni i konzistentni testovi, mogu se evidentirati lažne neispravnosti i propustiti prave neispravnosti u kôdu.

Neki projekti pokušavaju imati programere koji su zaduženi za određene dijelove kôda, npr. određene klase. Vlasnici klasâ se tada uvjeravaju da je kôd svake klase integriran i ispravno isporučen. To umanjuje probleme kod integracije, ali ovisnost među klasama uvijek može biti problem.
Takav pristup zaduženja programera po klasama također ne rješava problem.

Drugi način rješavanja problema integracije je uspostava integratora ili tima integratora u projektu. Integriranja kôda više programera često ne može obaviti samo jedna osoba. Tim ljudi je preveliki resurs za integriranje više od jednom na tjedan. U takvom okružju, programeri koriste starije verzije kôda koji je prošao integraciju.

Takva rješenja ne adresiraju korijenski problem. Želja je da programeri budu u mogućnosti raditi u paraleli, s odvažnošću činiti promjene na bilo kojem dijelu sustava (kôdu) bez grešaka u integraciji.

Striktno sekvencijalna ili jednonitna (engl. *Single Threaded*) integracija koju vrše sâmi programeri u kombinaciji s pristupom zajedničkog vlasništva nad kôdom rješenje je problema koje nudi XP metodologija. Sav izvorni kôd se isporučuje u zajednički kontrolni repozitorij izvornog kôda koji omogućuje da samo jedan pâr programera testira, kodira i integrira, tj. radi promjene na zajedničkom kontrolnom repozitoriju izvornog kôda. Zajednički kontrolni repozitorij izvornog kôda dopušta samo jednom pâru da radi promjene na određenim datotekama, tj. omogućuje mehanizam lokiranja. Jednonitna integracija omogućuje da se zadnja verzija može kontinuirano identificirati i koristiti.

5.3.5.1 Česta integracija

Programeri trebaju integrirati i isporučivati kôd u zajednički kontrolni repozitorij izvornog kôda kad god to ima smisla. Obično se integracija dešava nekoliko puta dnevno.

Kontinuiranom integracijom se izbjegavaju problemi ako su promjene u kôdu značajne.

Slučaj je da komunikacija među parovima nije dovoljno jaka o tome koji dijelovi kôda

se mogu dijeliti, a koji ponovno upotrijebiti. Svatko iz tima mora biti u mogućnosti raditi s posljednjom verzijom. Promjene na kôdu se ne bi smjele raditi na starom, već na zadnjem kôdu sa zajedničkog repozitorija.

Zadnja verzija kôda se stavlja i dohvaća sa zajedničkog kontrolnog repozitorija izvornog kôda. U tu se svrhu koriste repozitoriji, kao što su: *Microsoft Visual Source Safe* [42], *Rational ClearCase* [43], *Concurrent Versions System* (CVS) [44] i drugi.

Kontinuiranom integracijom se mogu izbjeći i rano detektirati problemi s kompatibilnosti.

Slika (5.3) prikazuje da je prilikom implementacije novih jediničnih testova i nove funkcionalnosti potrebno neprekidno izvršavati integraciju. Kontinuirana integracija nalaže izvršavanje jediničnih testova. U slučaju da svi testovi nisu prošli, potrebno je ispraviti pogreške u implementaciji ili u testovima ako testovi ne zadovoljavaju zahtjeve. U slučaju da su svi testovi prošli, pokreću se testovi prihvaćenosti softvera.

5.3.6 Zajedničko vlasništvo nad kôdom

Zajedničko vlasništvo nad kôdom (engl. *Collective Code Ownership*) ohrabruje sve koji sudjeluju na projektu da doprinose novim idejama u svim segmentima projekta. Bilo koji programer može promijeniti bilo koju liniju kôda kako bi dodao novu funkcionalnost, ispravio evidentirane neispravnosti ili refaktorirao (odjeljak 5.2.6).

Zajedničko vlasništvo nad kôdom znači da su i svi odgovorni za taj kôd. Cjelokupan tim je odgovoran za arhitekturu sustava i često se ne izdvaja posebna projektna uloga arhitekta. Arhitektura sustava je distribuirana u XP timu; svatko iz tima ima određene odgovornosti nad arhitekturalnim odlukama.

Način na koji funkcionira zajedničko vlasništvo nad kôdom nalaže da svaki programer kreira jedinične testove uz kôd koji je napisao. Testovi su velika pomoć drugim programerima koji će se susresti s tim kôdom jer:

- pokazuju kako se određeni dio kôda koristi
- kad se kôd mijenja (ispravljaju evidentirane neispravnosti ili se vrši refaktoriranje), izvršavanje testova ukazuje na promjenu funkcionalnosti.

Svi testovi i izvorni kôd trebaju biti pohranjeni na središnjem zajedničkom repozitoriju. Na zajednički repozitorij se isporučuje samo onaj kôd sa testovima koji radi, tj. kôd za koji potpuno prolaze testovi.

U kombinaciji sa čestom integracijom (odjeljak 5.3.5.1), originalni autor često jedva primjećuje da je originalna klasa ili metoda unutar klase proširena ili prepravljana.
U praksi, zajedničko vlasništvo nad kôdom je učinkovitije nego dodjeljivanje odgovornosti nad pojedinim klasama (dijelovima kôda) određenim osobama u timu.

5.3.7 Optimizacija kôda na kraju

Optimizaciju kôda treba ostaviti za kraj razvojnog projekta, nikako nije dobro raditi optimizaciju za vrijeme trajanja implementacije.

Proces optimizacije je različit od procesa refaktoriranja (odjeljak 5.2.6).
Refaktoriranjem se poboljšava kôd bez promjene funkcionalnosti, čime se eliminiraju viškovi i redundancija. Općenito, kôd se želi učiniti lakšim za razumijevanje i bolje strukturiranim.
Optimizacija kôda se vrši kako bi se određeni dijelovi kôda zamijenili s dijelovima koji su tehnički napredniji: manjeg su stupnja složenosti, jednostavniji su za korištenje te se vremenski brže izvode. Ponekad je potrebno zamijeniti korištene algoritme s bržim algoritmima ili primijeniti neki od poznatih obrazaca dizajna (engl. *design patterns*).

Najvažnije je da sustav ispravno radi. Tek nakon toga se može posvetiti optimizaciji kôda i brzini rada sustava.

5.3.8 40-satni radni tjedan

Prekovremeni rad slama timski duh i motivaciju. Razvojni projekti koji zahtijevaju prekovremeni rad kako bi bili završeni u zadanom vremenskom roku, najvjerojatnije će kasniti. Umjesto da se poseže za prekovremenim radom, potrebno je pažljivije činiti planiranje isporuke (odjeljak 5.1.2) kako bi se uskladili vremenski zahtjevi. Povećanje resursa dodavanjem više ljudi kako bi se zadovoljili vremenski rokovi je također loša ideja.

Kako bi XP tim uspješno obavljao timske zadatke, nužno je da članovi tima budu:

- odmorni ujutro, s puno novih i konstruktivnih ideja
- umorni nakon posla, ali sretni jer je njihov rad bio produktivan.

Da bi zadržao timski duh i motivaciju, menadžment treba:

- osigurati radnu okolinu bez stresa (npr. postavljanje gotovo nemogućih vremenskih rokova)
- ne zamarati tim s administrativnim problemima
- ne dopustiti prekovremeni rad više od jednog tjedna.

40-satni radni tjedan znači da je radni dan u trajanju od 8 sati. U pâru, gotovo je nemoguće raditi produktivno više od 6 sati na zahtjevnom poslu kao što je razvoj softvera.

5.4 Testiranje softvera u ekstremnom programiranju

Testovi se mogu podijeliti u dvije osnovne skupine:

1. Jedinični testovi
   Provjeravaju ispravnost rada sustava i uvijek moraju raditi. Daju sigurnost programerima u vlastiti kôd te ujedno objašnjavaju kako se kôd koristi. Moraju biti automatizirani kako bi se omogućila jednostavna ponovljivost.

2. Testovi prihvaćenosti
   Provjeravaju funkcionalnost čitavog sustava koji je opisan korisničkim pričama. Određuju da li sustav zadovoljava kriterij prihvaćenosti i omogućuju naručitelju da odluči da li može prihvatit sustav. Pišu ga naručitelj uz pomoć programera.

Testiranje je jedna od najvećih i najjačih značajki XP metodologije razvoja softvera. Svaki put kada programer promijeni kôd iz bilo kojeg razloga, potrebno je ponoviti jedinične testove kako bi se uvjerilo da promjene nisu narušile prethodno implementiranu i testiranu funkcionalnost.
Iz tog razloga je važno da testovi budu automatizirani kako bi se omogućila njihova ponovljivost. Programer je ujedno i jedinični tester svog kôda.

5.4.1 Testiranje jedinice programa

Jedinični testovi su jedna od najvažnijih stvari u ekstremnom programiranju [45]. Za uspješno korištenje jediničnih testova, potrebno je kreirati ili koristiti neki od gotovih test okruženja koja su dostupna na tržištu i kojima je moguće vršiti jedinično testiranje.
Kent Beck je napisao *SUnit* testno okružje za *Smalltalk* programski jezik [30]. Nakon toga su *Kent Beck* i *Erich Gamma* napravili *JUnit* [35] za J2SE [36] i J2EE [37] platformu. Pomoću ovih alata, moguće je automatizirano izvršavati jedinične testove iz IDE razvojnog okružja.

Potrebno je testirati većinu klasa i metoda u objektnom sustavu. Može se za svaku klasu koja se želi testirati generirati testna klasa i za svaku metodu testna metoda. Nije nužno potrebno testirati sve metode. Obično je nepotrebno testiranje metoda kojima se postavljaju ili dohvaćaju vrijednosti varijable ili objekta.
Poželjno je kreirati test prije pisanja kôda (odjeljak 5.3.3) ako je to moguće.

Jedinični testovi se stavljaju u zajednički repozitorij zajedno s datotekama izvornog kôda. Kôd bez pripadnog testa ne bi se uopće trebao stavljati u zajednički repozitorij. Ako je otkriveno da nedostaje jedinični test (npr. za neku metodu), test treba biti naknadno kreiran.

Najveći otpor odvajanju vremena na pisanje jediničnih testova su brzo dolazeći vremenski rokovi isporuke. Tijekom života projekta, automatizirani testovi mogu smanjiti troškove stotinu puta jer su najbolje oružje protiv evidentiranih neispravnosti. Čim je test teži za implementaciju, bit će potrebniji jer će donijeti i veće uštede. Automatizirani jedinični testovi u konačnici nude više nego što košta njihovo kreiranje.

Sljedeća pogrešna predodžba je da jedinični testovi mogu nastati u posljednja tri mjeseca projekta. Na nesreću, bez jediničnih testova će razvoj potpuno zaposjesti ta zadnja tri mjeseca. Čak i ako je vrijeme raspoloživo, razvoj testova zahtijeva određeno vrijeme za razvoj.
Otkrivanje svih problema koji se mogu pojaviti zahtijeva vrijeme. U slučaju da se žele

imati kompletni jedinični testovi, bitno je da se počnu implementirati zajedno s izvornim kôdom, što znači već prvog dana razvoja.

Jedinični testovi omogućuju zajedničko vlasništvo nad kôdom (odjeljak 5.3.6). Kada se kreiraju jedinični testovi, čuva se funkcionalnost da ne bude slučajno narušena. Zahtjev da kôd mora proći sve testove prije stavljanja testova i izvornog kôda u zajednički repozitorij osigurava da sva funkcionalnost uvijek radi. Vlasništvo nad kôdom nije potrebno ako su sve klase pokrivene s testovima.

Jedinični testovi također omogućuju refaktoriranje (odjeljak 5.2.6). Nakon svake male promjene kôda, jedinični testovi mogu verificirati da promjena u strukturi nije uvela promjenu u funkcionalnosti.

Kreiranje univerzalne garniture jediničnih testova (engl. *unit test suite*) za validaciju i regresijsko testiranje omogućuje čestu integraciju (odjeljak 5.3.5 i 5.3.5.1). Moguće je brzo integrirati bilo koju nedavnu promjenu pokrećući niz testova. Popravljanje malih problema svakih nekoliko sati oduzima manje vremena nego popravljanje velikih i kritičnih problema prije isporuke. Automatiziranjem jediničnih testova moguće je sjediniti skup promjena sa zadnjom verzijom u kratkom vremenu.

Često dodavanje nove funkcionalnosti zahtijeva mijenjanje jediničnih testova kako bi testovi odgovarali funkcionalnosti. Moguće je unijeti nepravilnosti istovremeno i u testove i u izvorni kôd, iako se to u praksi ne događa često. Povremeno se može dogoditi da je test pogrešan, a da je kôd ispravan no to se lako otkriva pokretanjem testa. Kreiranje testova koji nisu zavisni o kôdu prije nastajanja kôda uvelike omogućuju da će kôd raditi kada bude kreiran.

5.4.2 Pronalaženje neispravnosti

Kad se pronađe neispravnost, kreiraju se testovi kako bi se spriječilo da se ta neispravnost neprimijećeno opet pojavi. Evidentirana neispravnost zahtijeva da bude napisan test prihvaćenosti softvera kojeg potvrđuje naručitelj. Kreiranje testova prihvaćenosti prije suočavanja programerâ s kôdom pomaže naručitelju koncizno definiranje problema i komuniciranje problema razvojnom timu. Programeri u timu imaju test koji nije prošao i mogu fokusirati nastojanja da čim prije otkriju i riješe problem.

Kada je dobiven test prihvaćenosti koji nije prošao, programeri mogu kreirati jedinični test da pokažu neispravnost iz kuta gledanja koji je specifičan izvornom kôdu. Jedinični testovi koji nisu prošli daju odmah povratnu informaciju programerima da je neispravnost popravljena (prije su prolazili testovi koji nisu otkrivali neispravnost). Kada svi jedinični testovi opet prolaze, može se opet pokrenuti test prihvaćenosti (koji nije prolazio) koji pokazuje da je neispravnost popravljena.

Slika (5.1) prikazuje utjecaj, tj. sudjelovanje korisničkih priča na kreiranje testa prihvaćanja softvera kojeg potvrđuje naručitelj ovisno o mogućim scenarijima testiranja. Neispravnost u sustavu također zahtijeva kreiranje specifičnog testa prihvaćenosti. Kada se dogodi greška u isporuci, tj. otkrije se neispravnost, programeri popravljaju grešku te isporučuju zadnju i ispravljenu verziju sustava. Testom prihvaćenosti se može provjeriti

da li je greška ispravljena.
Uspjeli test prihvaćenosti vodi u slijedeću iteraciju. Odobravanjem naručitelja nastaju male inkrementalne isporuke.

5.4.3 Često izvršavanje testova prihvaćenosti

Testovi prihvaćenosti softvera koje potvrđuje naručitelj su kreirani iz korisničkih priča.

Za vrijeme iteracije, korisničke priče koje su odabrane tijekom planiranja isporuke će biti prevedene u testove prihvaćenosti. Naručitelj specificira scenarije testiranja kada su korisničke priče uspješno implementirane. Korisnička priča može imati jedan ili više testova prihvaćenosti, ovisno o tome koliko ih je potrebno da bi se osiguralo da funkcionalnost radi.

Testovi prihvaćenosti testiraju sustav kao crnu kutiju. Apstrakcija crne kutije odnosi se na vidljivost implementacije sustava iza sučelja sustava. Idealno, klijenti sustava predočenog apstrakcijom crne kutije ne trebaju znati nikakve detalje koji se nalaze iza sučelja ili specifikacije. Svaki test prihvaćenosti reprezentira neki očekivani rezultat sustava. Naručitelj je odgovoran za verifikaciju i ispravnost testova prihvaćenosti te recenziju rezultata testiranja kako bi odlučio koji testovi koji nisu prošli imaju najviši prioritet.

Korisnička priča se ne smatra završenom sve dok nije uspješno prošla test prihvaćenosti. To znači da novi test prihvaćenosti mora biti kreiran u svakoj iteraciji ili se može smatrati da u određenoj iteraciji nije bilo napretka.

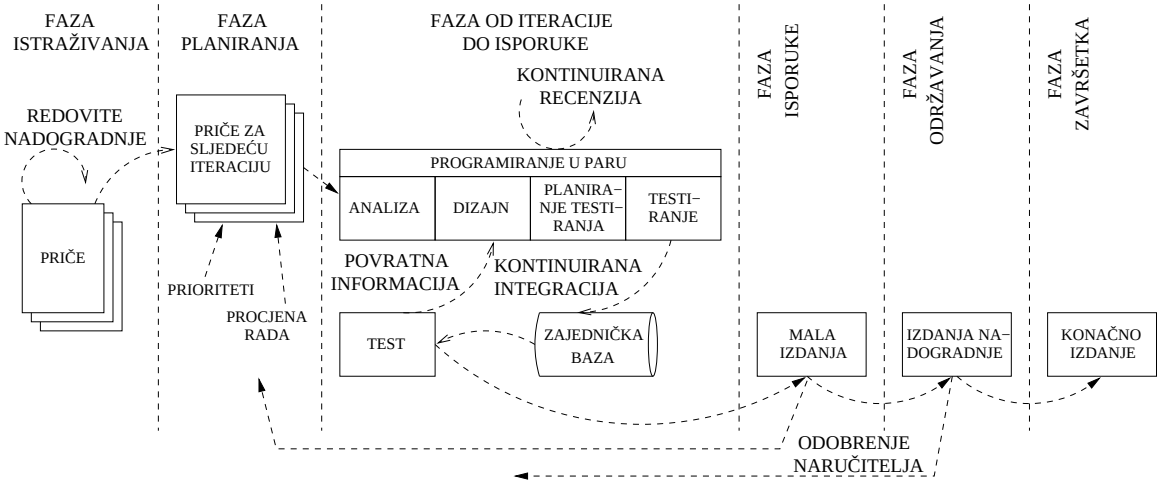
Testovi prihvaćenosti trebaju biti automatizirani tako da mogu biti često pokretani. Rezultati testova se prenose razvojnom timu. Odgovornost tima je da napravi vremenski raspored u svakoj iteraciji kako bi se popravile sve moguće greške.

5.5 Životni ciklus XP projekta

Životni ciklus XP projekta se sastoji od slijedećih faza [18] [25]:

1. Faza istraživanja (engl. *Exploration Phase*) (odjeljak 5.5.1)
2. Faza planiranja (engl. *Planning Phase*) (odjeljak 5.5.2)
3. Faza od iteracija do isporuke (engl. *Iterations to Release Phase*) (odjeljak 5.5.3)
4. Faza isporuke (engl. *Productionizing Phase*) (odjeljak 5.5.4)
5. Faza održavanja (engl. *Maintenance Phase*) (odjeljak 5.5.5)
6. Faza završetka (engl. *Death Phase*). (odjeljak 5.5.6)

Slika (5.5) prikazuje životni ciklus XP projekta sa glavnim fazama.
Faza istraživanja prikazuje korisničke priče koje će biti uključene u prvu isporuku.



Slika 5.5: Životni ciklus XP procesa

Naglašeno je da se priče redovno obnavljaju u slučaju da nisu dobro napisane.
Faza planiranja prikazuje specifikaciju korisničkih priča za slijedeću iteraciju. Priče koje ulaze u iteracije rezultat su prioriteta i procjena implementacije.
Faza od iteracije do isporuke prikazuje akcije u iteracijama sustava do prve isporuke. Prilikom programiranja u pâru, pâr čini analizu, dizajn, planiranje testiranja i testiranje. Sve akcije (analiza, dizajn, planiranje testiranja i testiranje) se kontinuirano revidiraju. Test daje povratnu informaciju koja utjeće na dizajn. Prilikom implementacije novih testova i nove funkcionalnosti, potrebno je neprekidno izvršavati integraciju. Kôd se treba nalaziti na zajedničkom repozitoriju.
Faza isporuke prikazuje nastajanje malih ali čestih isporuka. Prije isporuke, važno je da prolaze svi testovi iz faze od iteracije do isporuke. Odobrenje naručitelja dovodi do faze planiranja gdje se kreiraju i planiraju korisničke priče. Male isporuke tiču se i isporuka nadogradnje u fazi održavanja.
Faza održavanja prikazuje da se sustav drži u radu isporukama nadogradnje. Isporuke nadogradnje su također male isporuke iz faze isporuke. Odobrenje naručitelja dovodi do faze od iteracija do isporuke gdje se kroz iteracije kreira nova funkcionalnost.
Faza završetka prikazuje da je u radu konačna verzija i da nema više nove funkcionalnosti koja treba biti implementirana. Do konačne verzije dolazi se iz faze održavanja isporukama nadogradnje.

5.5.1 Istraživačka faza

U istraživačkoj fazi (engl. *Exploration Phase*) naručitelj ispisuje korisničke priče za koje želi da budu uključene u prvu isporuku. Istovremeno, programeri su sigurni da ne mogu dati bolje procjene za korisničke priče koje će ući u iteraciju dok ne počnu implementaciju.

U isto vrijeme, razvojni tim se aktivno upoznaje s alatima, tehnologijom i praksama koje će biti korištene u razvoju. Tehnologija koja se želi koristiti će biti testirana u određenoj mjeri kako bi se vidjelo da svojim mogućnostima zadovoljava. Arhitekturalne

mogućnosti sustava će biti istražene izradom prototipa sustava.

Dok tim radi tehnološke probe (odjeljak 5.2.4), naručitelj vježba pisanje korisničkih priča. Prve korisničke priče vjerojatno neće biti dobro napisane. Cilj je da naručitelj dobije kvalitetne povratne informacije od razvojnog tima kako bi mogao brzo naučiti specificirati što programeri trebaju i što ne trebaju. Ključno pitanje je: "*Mogu li programeri sigurno procijeniti vrijeme potrebno za implementaciju te korisničke priče?*".

Istraživačka faza traje od nekoliko tjedana do nekoliko mjeseci, najviše oviseći o tome koliko je tehnologija poznata programerima.

5.5.2 Faza planiranja

Uloga faze planiranja (engl. *Planning Phase*) za naručitelja i programere je da se slože oko datuma kada će najmanji i najprioritetniji skup korisničkih priča biti gotov.

U fazi planiranja se postavljaju prioriteti na korisničke priče. Programeri prvo procjenjuju koje je vrijeme potrebno za implementaciju pojedine priče te određuju raspored implementacije pojedine priče unutar iteracije. Naručitelj u dogovoru s projektnim timom određuje datum izdavanja verzije, važnost pojedine priče za kupca te prioritet pojedine korisničke priče unutar iteracije. Faza planiranja obično traje nekoliko dana.

5.5.3 Faza od iteracija do isporuke

Faza od iteracija do isporuke (engl. *Iterations to Release Phase*) uključuje nekoliko iteracija sustava prije prve isporuke. Implementacija svake iteracije obično traje od jednog do četiri tjedna.

Prva iteracija kreira sustav s arhitekturom cijelog budućeg sustava. To je postignuto odabirom onih korisničkih priča koje će omogućiti izgradnju strukture cjelokupnog sustava. Naručitelj odlučuje koje korisničke priče će ući u koju iteraciju. Pitanje koje si postavlja je: "*Koja je najvažnija stvar da radi u toj iteraciji?*".
Funkcijski test (obično test prihvaćenosti) koji kreira naručitelj se pokreće na kraju svake iteracije. Na kraju posljednje iteracije sustav je spreman za isporuku.

5.5.4 Faza isporuke

Faza isporuke (engl. *Productionizing Phase*) zahtijeva posebno testiranje i provjeru performansi sustava prije nego što sustav može biti isporučen kupcu. U toj fazi se mogu dodavati nove promjene uz odluku da li će biti uključene u trenutnu isporuku ili u neku od slijedećih isporuka.

Za vrijeme faze isporuke, iteracije trebaju biti skraćene od tri tjedna na jedan tjedan uz obavezno održavanje dnevnih stojećih sastanaka. Odložene, tj. odgođene predložene ideje su dokumentirane za kasniju implementaciju (npr. za vrijeme faze održavanja, odjeljak 5.5.5).

Također, u fazi isporuke se može pozabaviti sa pitanjima performansi sustava ali tek nakon što sustav provjereno radi.

5.5.5 Faza održavanja

Nakon prve isporuke, XP tim mora održavati sustav u radu zajedno s implementacijom novih iteracija (nove funkcionalnosti). Kako bi se to postiglo, faza održavanja (engl. *Maintenance Phase*) zahtijeva napor za pružanje podrške kupcu.

Razvoj sustava koji je isporučen te je u radu nije isti kao i razvoj sustava koji još nije došao u fazu isporuke. Treba biti oprezniji s promjenama koje se rade te biti spreman prekinuti trenutni daljnji razvoj kako bi se riješili problemi u radu jer su oni najvišeg prioriteta.

Razvojna brzina (odjeljak 5.1.4) se može deklarirati nakon što je sustav isporučen i u radu. Faza održavanja može zahtijevati inkorporiranje novih ljudi u projektni tim i određene promjene strukture tima.

5.5.6 Faza završetka

Faza završetka (engl. *Death Phase*) je onaj trenutak kada naručitelj više nema korisničkih priča (odjeljak 5.1.1) koje trebaju biti implementirane. Naručitelj je zadovoljan sa sustavom i trenutno ne zahtijeva nikakva poboljšanja. To također znači da sustav zadovoljava potrebe naručitelja također i u ostalim aspektima (npr. performanse, pouzdanost, skalabilnost i drugo).

Faza završetka je onaj trenutak XP procesa kada je potrebna dodatna dokumentacija sustava konačno napisana i kada više nisu potrebne promjene na arhitekturi, dizajnu i kôdu.

Završetak projekta se također može dogoditi kada sustav ne isporučuje željene rezultate prije planiranog kraja. Kraj je i kada sustav postane preskup za daljnji razvoj ili dodavanje novih funkcionalnosti.

5.5.7 Rizici u životnom ciklusu XP projekta

Osnovni rizici (problemi) koji se mogu javiti u životnom ciklusu XP projekta po fazama su:

1. Faza istraživanja
    - tim nema dovoljno znanja niti iskustva kako bi započeo s implementacijom i uspješno proizveo sustav (programski proizvod)
    - nedovoljno su istražene mogućnosti arhitekture sustava
    - nedovoljno je poznavanje razvojnih tehnologija
2. Faza planiranja
    - nerealno je određen datum izdavanja najprioritetnijeg skupa korisničkih priča
    - pogrešno su određeni prioriteti implementacije korisničkih priča
3. Faza od iteracije do isporuke

- mogući su problemi s razvojnim alatima koji se koriste
- nedovoljno je dobro izgrađena arhitektura sustava

4. Faza isporuke
   - mogući su problemi u testiranju sustava
   - mogući su problemi s performansama sustava
5. Faza održavanja
   - mogući su prekobrojni problemi u radu sustava koji ometaju razvoj novih funkcionalnosti
   - mogući su problemi u integraciji novih funkcionalnosti sa sustavom koji je u radu
6. Faza završetka
   - moguće je da sustav ne ispunjava željene rezultate prije planiranog kraja
   - sustav je pre skup za daljnji razvoj ili dodavanje novih funkcionalnosti

5.6 XP projektne uloge

Postoje različite projektne uloge u XP-u koje su se profilirale za različite zadaće i svrhe tijekom procesa i praksi. Mogu se izdvojiti slijedeće projektne uloge [25]:

- programer (engl. *Programmer*) (odjeljak 5.6.1)
- naručitelj (engl. *Customer*) (odjeljak 5.6.2)
- tester (engl. *Tester*) (odjeljak 5.6.3)
- tragač (engl. *Tracker*) (odjeljak 5.6.4)
- trener (engl. *Coach*) (odjeljak 5.6.5)
- savjetnik (engl. *Consultant*) (odjeljak 5.6.6)
- menedžer (engl. *Manager*). (odjeljak 5.6.7)

Uloge menadžera, tragača i trenera može fizički obavljati jedna osoba. Te tri uloge su menedžerske uloge.

5.6.1 Programer

Programer je srce XP-a. Kada bi programer uvijek mogao donositi odluke koje pažljivo balansiraju između kratkoročnih i dugoročnih prioriteta, ne bi bilo potrebe za nekom drugom tehničkom osobom osim programera. Programeri su, dakle, uz naručitelje (odjeljak 5.6.2), druga polovica osnovnog tima u XP-u.

XP programer je zapravo isto kao i programer bilo kojeg druge discipline razvoja softvera. Glavni zadatak programera je rad s programima, odnosno kôdom; čineći ih većim, jednostavnijim i bržim. Zadatke kodiranja XP programera se mogu rezimirati ovako:

- testiranje - kodiranje - refaktoriranje

što je u suprotnosti s tradicionalnim pristupom:

- dizajn - kodiranje - testiranje.

Zadaća programera, osim razvoja kôda, je i komunikacija s ostalim osobama koje sudjeluju u projektu, dakle projektnog tima. Programer je autor testova za svoj kôd kojima pokazuje da njegov kôd radi i kako radi (odjeljak 5.4.1).

Postoje vještine koje XP programer mora posjedovati (steći treningom). Najvažnija među njima je programiranje u pâru.

Sljedeća vještina je osjećaj za jednostavnost. Važno je održavati kôd čim više jednostavnim kako bi promjene na kôdu bile lakše (npr. dodavanje nove funkcionalnosti, ispravljanje evidentiranih neispravnosti, refaktoriranje i slično) te kako bi bila veća razumljivost tog kôda.

Ostale vještine koje treba posjedovati XP programer su tehnički orijentirane i tiču se znanja dobrog programiranja. Uključuju znanje solidnog programiranja, refaktoriranja te pisanja jediničnih testova.

5.6.2 Naručitelj

Naručitelj je, uz programera, druga polovica osnovnog tima u XP-u. Dok programer zna *kako* programirati, naručitelj zna *što* programirati.

Osnovna vještina koju treba posjedovati naručitelj je da piše dobre, programerima razumljive korisničke priče (odjeljak 5.1.1). Potreban je stav koji potiče opću uspješnost projekta.

Naručitelj određuje prioritet priča, odnosno zahtjeva unutar iteracije, znajući koja je važnost pojedine priče za kupca.

Pisanje testova prihvaćenosti (odjeljak 5.4.3) je također jedna od zadaća naručitelja. Na osnovu rezultata testova, naručitelj određuje koja funkcionalnost sustava je zadovoljena.

5.6.3 Tester

Budući da je mnogo testnih odgovornosti na leđima programera (odjeljak 5.6.1) [28], uloga testera u XP timu je fokusiranost na naručitelja. Prvenstveno, pomaže naručitelju

u pisanju funkcijskih testova. Ako funkcijski testovi nisu dio integracijske garniture, odgovornost testera je da redovito pokreće funkcijske testove i rezultate objavljuje na vidljivom mjestu.

XP tester nije odijeljena osoba posvećena 'probijanju' sustava. Zadaci testera su redovno pokretanje testova, obrađivanje rezultata testiranja i provjera da programska pomagala za testiranje rade u redu.

5.6.4 Tragač

Tragač je osoba koja vodi računa o dogovorenim i završenim zadacima u pojedinoj iteraciji. On mjeri količinu obavljenog posla XP tima.

XP propisuje praćenje nekoliko metrika. Najvažnija metrika je brzina projekta. Važan podatak je promjena brzine projekta, količina prekovremenog rada te odnos rezultata testova (postotak evidentiranih neispravnosti u odnosu na ispravnu funkcionalnost).
Svi ti podaci mjere napredak i pomažu odrediti da li se projekt odvija prema dogovorenom rasporedu u trenutnoj iteraciji.

Tragač može upozoriti ako je ugrožen dogovoreni raspored. Kako bi odredio brzinu projekta unutar iteracije, tragač obično svaki dan ili svaka dva dana prati završene zadaće unutar projektnog tima.
Kada XP tim bude određivao trajanje iteracije na osnovu procijenjenog trajanja korisničkih priča, moći će se koristit podatak o brzini projekta iz prethodnih iteracija za donošenje realnijih procjena.

Redovito praćenje napretka može pomoći XP timu da bolje uoči i popravi svoje nedostatke, znajući u svakom trenutku da li je na dobrom putu da izvrši dogovoreno.

5.6.5 Trener

Trener je odgovorna osoba za cjelokupan proces. Njegova zadaća je pratiti da li se članovi tima pridržavaju primijenjene razvojne metodologije.

Svatko u XP timu je odgovoran razumjeti vlastitu ulogu u XP-u. Trener je treba 'dublje' razumjeti, znati koje alternativne prakse mogu pomoći riješiti određene probleme, kako drugi timovi primjenjuju XP, koje su ideje iza XP-a i kako su povezane s trenutnom situacijom.

Trener u XP-u vodi tim i njegov je mentor. Njegova pozicija je da vodi XP tim svojim primjerom. Glavna vrlina mu je posjedovanje golemog iskustva i tehničkog znanja.
Trener vodi tim kako bi tim razumio XP i razvoj softvera općenito.

Općenito, vještine koje razvija XP su jednostavan dizajn (odjeljak 5.2.1), refaktoriranje (odjeljak 5.2.6) i testiranje (odjeljak 5.4). U slučaju da je tim tehnički jak a treba znanja o procesu, trener može voditi tim bez posebnih tehničkih znanja. Ako tim ima i tehnička znanja i znanja procesa, trener mora konstantno podsjećati tim na procedure kojih se trebaju pridržavati u određenim situacijama. Posao se trenera smanjuje sa starošću tima.

5.6.6 Savjetnik

XP tim s vremena na vrijeme treba tehničkog savjetnika u vidu konzultanta, unatoč fleksibilnosti i znanju koje posjeduje.

Uloga konzultanta je pomoći timu riješiti određeni tehnički problem ili mu razjasniti nejasnoće iz domene kojom se XP tim bavi.
Tim treba kreirati posebne testove kako bi ukazao na problem te kako bi bilo vidljivo da je problem riješen.

Ideja je poučiti tim kako da ubuduće rješava tehničke probleme.

5.6.7 Menadžer

Menadžer je osoba izvan tima i predstavlja tim prema vanjskom svijetu. On oformljuje, tj. slaže tim te nabavlja potrebne resurse. Također, upravlja timom (organizira sastanke, bilježi napredak i slično). Vlasnik je tima, ali i njihovih problema [26].

5.7 Slijed dnevnih aktivnosti u XP-u

Jedan tipičan dan XP programiranja može se ovako sumirati [26]:

1. Dnevni 'stojeći' sastanak na početku dana (odjeljak 5.1.8)
   Identificiraju se problemi te se određuje tko će ih riješiti (problemi se ne rješavaju na sastanku). Donosi se izvještaj (pregled) aktivnosti od prethodnog dana.

2. Formiranje parova programera (odjeljak 5.3.4)
   Sâv kôd koji će biti uključen u isporuku je kreiran od dva programera koji rade u pâru za jednim računalom. Oba programera iz pâra su uključena u problematiku, odlučivanje i jedinično testiranje.

3. Testiranje (odjeljak 5.4)
   Implementacija testova prethodi kodiranju (odjeljak 5.3.3). Važno je pokriti testovima čim veći dio kôda (sve što može krenuti loše), po mogućnosti kompletan kôd (odjeljak 5.4.1). Za sve nepoznanice i probleme, potrebno se je obratiti naručitelju (odjeljak 5.6.2).

4. Kodiranje (odjeljak 5.3)
   Nakon testiranja slijedi kodiranje (odjeljak 5.3.3). Važno je implementirati najjednostavnije stvari koje bi mogle funkcionirati (odjeljak 5.2.1).

5. Refaktoriranje (odjeljak 5.2.6)
   Refaktoriranje je tehnika poboljšavanja kôda bez promjene funkcionalnosti.

6. Pitanja i odgovori
   Za sva moguća pitanja i nejasnoće, na raspolaganju je naručitelj (odjeljak 5.6.2).

7. Integriraj ili odbaci (engl. *Integrate or Toss*) (odjeljak 5.3.5.1)
   Programeri trebaju integrirati i isporučivati kôd u zajednički kontrolni repozitorij

izvornog kôda i prilikom toga ponoviti testove. Ako neki testovi ne prolaze, potrebno je otkriti i popraviti moguće probleme tako da svi testovi na kraju prođu. Ako nešto nije dobro, treba to odbaciti te pokušati napraviti iz početka (isti dan ako još ima vremena ili slijedeći dan).

8. Kraj posla u 17h
   Potrebno je držati se 40-satnog radnog tjedna, bez prekovremenog rada (odjeljak 5.3.8). Sve što se radilo tog dana je integrirano ili odbačeno.

Poglavlje 6

Primjene razvojnih postupaka metode ekstremnog programiranja

Ovo poglavlje opisuje iskustvo uvođenja određenih postupaka metodologije ekstremnog programiranja razvoja softvera u projekt razvoja javne elektroničke usluge. Razvojni projekt se vršio unutar organizacije kojoj je osnovna djelatnost razvoj softvera.

Istražuje se mogućnost primjene razvojnih postupaka definiranih unutar metode ekstremnog programiranja, s posebnim naglaskom na konačni programski proizvod. Ispituje se primjena određenih praksi XP-a sa stajališta osoba uključenih u realizaciju razvojnog projekta (programera, testera, arhitekta sustava i menadžera).

Period u kojem je vršeno istraživanje je faza implementacije programskog proizvoda, dakle faza razvoja. Trajanje implementacije programskog proizvoda na kojem je vršeno promatranje je bilo nešto kraće od pola godine.

Osnovno područje istraživanja kojeg opisuje ovo poglavlje su performanse procesa na razvojnom projektu. Promatra se vrijeme potrebno za implementaciju zadane funkcionalnosti, korištenje resursa u sâmoj implementaciji i kvaliteta kôda, tj. kakvoća programskog proizvoda. Posebna važnost je usmjerena na postupke testiranja, mogućnostima poboljšanja programskog kôda te korištenju i integraciji postojećih programskih pomagala. U analizi su korištene metode mjerenja i intervjua sa članovima projektnog tima.

Uspoređuje se projekt razvoja javne elektroničke usluge koji je u fazi razvoja implementirao neke postupke XP razvojne metodologije sa "klasičnim" (ne XP) razvojnim projektom uobičajenim u istoj organizaciji.
Uočavaju se glavne prednosti i nedostaci obaju pristupa. U istraživanju je osigurana nepristranost i pouzdanost podataka jer je osnovna ideja ocijeniti mogućnost uvođenja prikladnih praksi XP razvojne metodologije u organizaciju.

6.1 Opis projekta razvoja javne elektroničke usluge

Razvojni projekt na kojem je izvršeno ovo istraživanje je *HealthCare Agent* (skraćeno: *HC Agent*) [46]. Radi se o implementaciji programskog sustava u sklopu projekta implementacije *Informacijskog sustava Primarne zdravstvene zaštite* (ISPZZ) u Republici

Hrvatskoj.

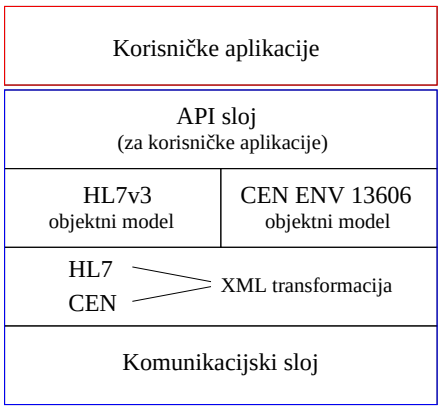
Ovaj i slični projekti informatizacije zdravstva trebaju omogućiti elektroničko poslovanje različitim poslovnim subjektima, ali i donijeti brojne koristi korisnicima zdravstvenih usluga, kao što su primjena komunikacijskih mreža za naručivanje na specijalističke preglede, korištenje elektroničkog zdravstvenog kartona koji omogućuje pristup medicinskim podacima bez obzira na lokaciju liječnika.

Uvođenje informacijskog sustava zdravstvene zaštite ima za cilj poboljšati cjelokupan sustav zdravstva Republike Hrvatske.

HC Agent je programski sustav, odnosno komponenta koja je namijenjena osobama koje se bave razvojem aplikacija u području zdravstvene djelatnosti. Komponenta je zasnovana na znanjima o specifičnim protokolima potrebnim za usluge središnjeg informacijskog sustava zdravstvene skrbi. *HC Agent* pojednostavljuje razvoj klijentskih aplikacija za zdravstvo pružajući jednostavno sučelje nužno za pristup složenim uslugama središnjeg informacijskog sustava koristeći složeni skup protokola.

HC Agent povezuje različite korisničke aplikacije u domeni zdravstva pomoću *Web* usluge središnjeg informacijskog sustava zdravstvene skrbi korištenjem HL7v3 [47] (engl. *Health Level Seven Version 3*) i CEN ENV 13606 normi [48]. HL7 je ANSI organizacija fokusirana na područje zdravstvene zaštite. HL7 se odnosi na najviši, sedmi nivo OSI referentnog modela što odgovara aplikacijskom sučelju. Komunikacija *HC Agent*a i centralnog sustava zdravstva se odvija razmjenom XML [49] poruka.

HC Agent je realiziran kao *Microsoft .NET* komponenta za *Microsoft .NET* [39] platformu s primarnom zadaćom da omogući integraciju širokog spektra klijentskih aplikacija iz domene zdravstva sa središnjim sustavom zdravstva. Komponenta je dizajnirana da lako služi kao osnovni gradivni blok malih i srednjih sustava u domeni zdravstva i da omogući brz i jednostavan razvoj klijentskih aplikacija. Glavna karakteristika komponente je visoka konfigurabilnost te programsko sučelje koje je jednostavno za korištenje i koje omogućuje integraciju aplikacija sa integriranim ICT (informacijska i komunikacijska tehnologija, engl. *Information and Communications Technology*) okružjem baziranim na HL7v3 [50] i CEN ENV 13606 [48] standardima.



Slika 6.1: *HalthCare Agent*

Slika (6.1) prikazuje arhitekturu *HC Agent* komponentnog programskog sustava. *HC Agent* programski sustav se sastoji od nekoliko različitih programskih podsustava. Ti programski podsustavi, odnosno programske komponente su:

- *HC Agent* komponenta (koja je i *Microsoft .NET* komponenta) je API (aplikacijsko programsko sučelje, engl. *Application Program Interface*) sučelje između klijentskih aplikacija zdravstvenog sustava i aktualnih HL7v3, odnosno CEN ENV 13606 poruka prema središnjem informacijskom sustavu. Na slici je komponenta označena kao API sloj prema korisničkim aplikacijama.

- HL7 verzije 3 (na slici HL7v3) objektni model je komponenta koja sadrži poslovnu logiku, automate stanja te RIM i D-MIM implementaciju [47] informacijskih modela (čini HL7 API).
  RIM (engl. *Reference Information Model*) je statički model koji je izvor za podatkovni sadržaj svih HL7 poruka koje se šalju i primaju kao XML poruke. D-MIM (engl. *Domain Message Information Model*) je pročišćen podskup RIM modela koji sadrži skup klasa, atributa i relacija koje mogu biti korištene kako bi se kreirale HL7 poruke za određenu domenu (određeno područje u zdravstvu).

- CEN ENV 13606 objektni model je komponenta prema CEN/TC 251/N99-040 [48] standardu te sadrži poslovnu logiku navedenog protokola, odnosno standarda.

- HL7 i CEN XML transformacija je komponenta koja kreira XML komunikacijske poruke bazirane na HL7v3 i CEN ENV 13606 standardima te transformira primljene XML poruke prema navedenim standardima.

- HL7 verzije 3 *Web Services Client* [51] (na slici označeno kao komunikacijski sloj) je komunikacijski sloj zadužen za transmisiju XML poruka prema središnjem sustavu zdravstva.

- HL7 verzije 3 podatkovni tipovi (*HL7v3 Data Types*) [47] je komponenta koja sadrži implementirane podatkovne tipove koje koristi HL7 protokol te je koriste sve komponente *HC Agent* sustava. Na slici komponenta nije posebno označena.

Jedna od važnijih komponenti *HC Agent* komponentnog programskog sustava su HL7 podatkovni tipovi (engl. *HL7 Data Types*). Budući da se isti podatkovni tipovi koriste i u središnjem sustavu zdravstva na koji se *HC Agent* programski sustav spaja i s njim komunicira, koriste ga i sve komponente *HC Agent*-a, ta komponenta je implementirana u *Java* programskom jeziku za J2SE [36] i J2EE [37] platformu.
S implementacijom HL7 podatkovnih tipova je zapravo i započeo razvoj *HC Agent* sustava jer je ta komponenta njegova okosnica.

Nakon završene implementacije HL7 podatkovnih tipova, kôd te komponente je modificiran da bude kompatibilan s *Microsoft Visual J#* [52] programskim alatom jer je u kasnijem periodu razvoja bio zahtjev da ta komponenta, kao i ostale komponente *HC Agent* sustava, bude dio *Microsoft Windows .NET* platforme. *Microsoft Visual J#* je alat koji Java kôd može koristiti kako bi izgradio aplikacije i servise na *Microsoft .NET* platformi. Ostale komponente *HC Agent* programskog sustava su implementirane u *Microsoft Visual C#* [53] programskom jeziku na *Microsoft .NET* platformi.

HC Agent sustav se sastoji od nekoliko osnovnih funkcionalnosti koje se mogu prikazati u formi slučajeva uporabe. Osnovni slučajevi uporabe *HC Agent* komponentnog programskog sustava su:

1. Dobavljanje dozvole za rad (engl. *Get Work Permission*)
   Aktivnosti ovog slučaja uporabe zadovoljavaju dva cilja:
   (a) Identificiranje određenog korisnika (liječnika, medicinske sestre ili nekog drugog djelatnika u zdravstvu) koji traži uslugu od središnjeg sustava zdravstva.
   (b) Omogućavanje korištenja usluga središnjeg sustava zdravstva potencijalnom korisniku.

2. Dohvat osiguranja pacijenta (engl. *Retrieve Patient Eligibility*)
   Ovaj slučaj uporabe opisuje akcije dohvata statusa zdravstvenog osiguranja pacijenta od središnjeg sustava zdravstva. Procedura je obavezni dio procesa zaprimanja pacijenta. Predstavlja vrlo važnu administrativnu zadaću koja treba biti obavljena prije nekih drugih akcija koje se tiču postupka zaprimanja i obrade pacijenta.

3. Dohvat administrativnih podataka pacijenta (engl. *Retrieve Patient Administrative Data*)
   Ovaj slučaj uporabe opisuje aktivnosti dohvaćanja administrativnih podataka pacijenta. Ova aktivnost je obvezni dio postupka zaprimanja pacijenta. Prije dolaska pacijenta, medicinska sestra mora poslati upit centralnom sustavu zdravstva kako bi provjerila kojem davatelju usluge zdravstva pacijent pripada i kako bi dobila enkriptirani indeks pacijenta za buduće pristupe medicinskim podacima pacijenta.

4. Dohvat medicinskih podataka pacijenta (engl. *Retrieve Patient Medical Data*)
   Ovaj slučaj uporabe opisuje akcije dohvaćanja medicinskih podataka (elektroničkog zdravstvenog kartona) za određenog pacijenta sa centralnog sustava zdravstva.

5. Ažuriranje elektroničkog zdravstvenog kartona (engl. *Update Patient Medical Record*)
   Ovaj slučaj uporabe opisuje akciju ažuriranja elektroničkog zdravstvenog kartona pacijenta sa medicinskim podacima dobivenim za vrijeme zdravstvene obrade.

6. Uspostavljanje računa (engl. *Deliver Billing Information*)
   Kako bi ispostavio račun za pruženu zdravstvenu uslugu, davatelj usluge mora poslati određene informacije zdravstvenom osiguranju. To opisuje ovaj slučaj uporabe.

7. Slanje javnih zdravstvenih izvješća (engl. *Send Public Health Report*)
   Zbog različitih razloga (kontrole zaraznih bolesti, prikupljanja podataka u statističke svrhe i ostalo), nacionalni zakon nalaže pružateljima usluga da šalju različite informacije ustanovi Javnog zdravstva. To opisuje ovaj slučaj uporabe.

8. Isporuka izvješća o zdravstvenom osiguranju (engl. *Deliver Health Insurance Report*)
   U svojoj svakodnevnoj praksi, davatelj zdravstvene usluge mora poslati različita izvješća (dnevno izvješće, pregled pruženih usluga, mjesečno izvješće itd.). To opisuje ovaj slučaj uporabe.

HC Agent potprojekt je bio prikladan za primjenu nekih postupaka XP metodologije razvoja softvera zbog više razloga. U kasnijem izlaganju će biti podrobnije navedeni ti razlozi.

6.2 Primjene razvojnih postupaka metode ekstremnog programiranja

U *HC Agent* razvojnom projektu na kojem je vršeno istraživanje, implementirane su neke od temeljnih karakteristika, odnosno osnovnih postupaka metode ekstremnog programiranja. Naglasak postupaka je bio na oblikovanju programskog proizvoda.

Neke prakse nije bilo moguće provesti zbog specifičnosti radne organizacije u kojoj se obavljao projekt, tj. postojanja određenih organizacijskih standarda.

XP projekt (sa svim navedenim praksama i projektnim ulogama) u pravilu u organizaciji nije moguće provoditi zbog slijedećih razloga:

- Karakteristike sâme organizacije koja obavlja razvoj specifičnih programskih i telekomunikacijskih programskih i sklopovskih sustava, namijenjenih AXE komutacijskim sustavima [54] (digitalnim komutacijskim centralama u javnoj telekomunikacijskoj mreži za sve tipove javnih telefonskih aplikacija) i CELLO paketskim platformama (paketska platforma za pristup i transport produkata u fiksnoj i mobilnoj mreži namijenjena velikom broju aplikacija, kao što su RBS (engl. *Radio Base Station*), RNC (engl. *Radio Network Controller*) i druge). Takvi projekti koriste prilično učinkovit PROPS [55] opći model vođenja i organizacije projekta. Zbog svoje složenosti, takvi projekti pripisuju golemu važnost pisanim, unaprijed utvrđenim i u kasnijim fazama projekta nepromjenjivim zahtjevima. Organizaciju projekta temelje na opsežnoj ali dostatnoj dokumentacije.
  Stav je organizacije da razvojni projekt koji koristi nove razvojne tehnologije mora zadržati slične korporativne standarde, vezane uz dokumentaciju i zahtjeve te da nije preporučljivo niti pogodno da bude baziran na osnovi programskog kôda, što je jedna od praksi XP metodologije razvoja softvera.

- U projektima obično sudjeluje veći broj osoba, obično mnogo više od 12 za koji XP pripisuje da je maksimum za projekte koji žele efikasno koristiti XP.

- Mnogi projekti su distribuirani (dijelovi projekata se obavljaju u različitim organizacijama u različitim zemljama). Tu nisu primjenjive neke prakse XP-a, kao što je npr. programiranje u pâru.

- Tehničke specifičnosti projekata za koje nije pogodan brzi razvoj koji je osnova XP-a.

i još drugih razloga.

Zbog relativno malog broja osoba koje su sudjelovale u projektu, sâme specifičnosti projekta te duboke zainteresiranosti voditelja projekta i članova razvojnog tima za XP-

om, bilo je moguće provoditi neke određene postupke XP razvojne metodologije na *HC Agent* projektu.

U nastavku slijedi opis primijenjenih postupaka XP-a u projektnom timu i doprinos postupaka na status i napredak opisanog *HC Agent* razvojnog projekta.

6.2.1 Programiranje u pâru

U razvojnom projektu implementacije *HC Agent* softverske komponente (odjeljak 6.1) isprobana je tehnika programiranja u pâru (odjeljak 5.3.4).

U *HC Agent* projektu je sudjelovalo osmero ljudi, od čega su dvojica bila uključena u testiranje, a četvorica u programiranje. Bilo je moguće grupirati ljude u tri pâra, kako bi se isprobala i koristila ova tehnika.

Jedan pâr se bavio isključivo testiranjem. Testiranje koje je provodio pâr je bilo funkcijsko testiranje komponente u cijelosti, što je uključilo implementaciju i izvođenje testova prihvaćenosti koji su bili u obliku posebno razvijenih testnih aplikacija s grafičkim sučeljem za klijente koji koriste *Microsoft .NET* platformu i klijente koji koriste COM tehnologiju [56].

Dva pâra razvojnog tima su se bavila implementacijom kôda (programiranjem). Prema praksi XP-a, to je uključivalo implementaciju i izvođenje jediničnih testova.

Od preostale dvojice u timu, zadaća jedne osobe je bila tehničko vođenje projekta dok je druga osoba vršila ulogu voditelja projekta.

6.2.2 Tehnika refaktoriranja

Isprobana je tehnika refaktoriranja (odjeljak 5.2.6) u razvojnom projektu implementacije *HealthCare Agent* softverske komponente (odjeljak 6.1).

Slijedi opis najvažnijih tehnika refaktoriranja [31] koje su bile u određenoj primjeni u *HC Agent* razvojnom projektu.

Izdvojeni su najčešći problemi i navedene tehnike refaktoriranja koje mogu biti primijenjene na kôdu kako bi se riješili takvi problemi te kôd postao tehnički naprednijim.

Najčešći prisutni problemi u *HC Agent* razvojnom projektu gdje se mogla uspješno koristiti tehnika refaktoriranja su bili slijedeći:

- Dupli cirani kôd (engl. *Duplicated Code*) (odjeljak 6.2.2.1)
- Dugačka metoda (engl. *Long Method*) (odjeljak 6.2.2.2)
- Golema klasa (engl. *Large Class*) (odjeljak 6.2.2.3)
- Dugačka lista parametara (engl. *Long Parameter List*) (odjeljak 6.2.2.4).

6.2.2.1 Dupliciran kôd

Fragmenti kôda mogu biti grupirani zajedno, čime se uklanja dupliciran kôd.

Najjednostavniji problem dupliciranja kôda je višestruko postojanje istog izraza. Tehnikom izdvajanje metode (odjeljak 6.2.2.1) se isti kôd poziva sa različitih mjesta.

Slijedeći problem je izdvajanje dvije iste metode u klasu u višoj hijerarhije koju obije klase nasljeđuju (tehnika područja zaustavljanja i tehnika predloška metode).

Primjer izdvajanja klase pokriva slučaj sadržajnog rasta klasa.

Izdvajanje metode

Tehnika izdvajanja metode (engl. *Extract Method*) je primjer grupiranja kôda što i pokazuje slijedeći jednostavni primjer.

Ovaj kôd:

```
void printOwing(double amount) {
      printBanner();

      // print details
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```

se zamjenjuje s ovim kôdom:

```
void printOwing(double amount) {
      printBanner();
      printDetails(amount);
}

void printDetails (double amount) {
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```

Ovo je jedna od najčešćih tehnika refaktoriranja koja je korištena u *HC Agent* razvojnom projektu. Traži se metoda koja je preduga ili dio kôda koji treba komentar kako bi se razumjela njegova svrha. Tada se od tog fragmenta kôda kreira posebna metoda, kao što je prikazano na prethodnom jednostavnom primjeru.

Na *HC Agent* projektu je dogovoreno da je poželjno korištenje kratkih metoda i to iz nekoliko razloga:

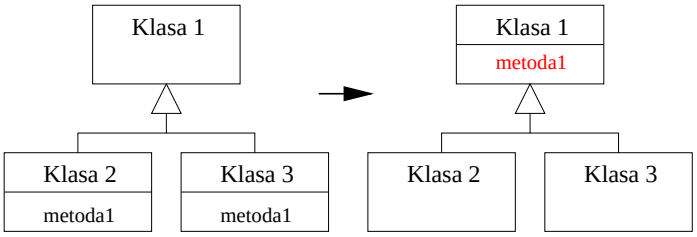
1. Povećavaju se šanse da druge metode koriste tu izdvojenu metodu.
2. Metode više razine zbog ove prakse izgledaju "čisto" i pregledno, kao "serija komentara" (tj. poziva izdvojenih metoda). To se pokazalo naročito praktično u slučajevima kada se vršila serijalizacija implementiranog objektnog modela u XML dokumente, odnosno komunikacijske poruke.

Problem koji se javio prilikom nastojanja korištenja kratkih, jezgrovitih metoda je bilo njihovo smisleno imenovanje. Smisleno imenovanje metoda je vršeno unutar provođenog postupka standardiziranog pisanja kôda. Taj problem je s vremenom i iskustvom tima riješen.

Uočeno je da sâma dužina metode nije problem, već semantička udaljenost između imena metode i njenog područja rada.

Područje zaustavljanja

Slijedeći primjer izbjegavanja dupliciranja kôda prikazuje tehnika područja zaustavljanja (engl. *Pull Up Field*).



Slika 6.2: Područje zaustavljanja (engl. *PullUpField*)

Ako su dvije klase razvijene neovisno ili kombinirane refaktoriranjem, često se mogu u njima naći dvostruke karakteristike, odnosno slični dijelovi kôda. Ti dijelovi, odnosno metode ponekad imaju slična imena, no to nije uvijek slučaj. Ako su ti dijelovi isti ili dovoljno slični, mogu se generalizirati.

Slika 6.2 prikazuje primjer korištenja tehnike područja zaustavljanja.

Ovaj postupak smanjuje dvostruki kôd omogučujući pomicanje podataka u hijerarhiji klasâ u višu klasu (engl. *parent class*) koju druge klase nasljeđuju (potklase, engl. *child class*).

Ova tehnika je bila često primjenjivana u *HC Agent* razvojnom projektu, što se može pokazati slijedećim primjerom.
Zbog čestog korištenja provjere vrijednosti objekata, oblikovane su i izdvojene posebne metode čija je funkcija provjeravanje vrijednosti objekata. Ove metode su često korištene kod provjere vrijednosti objekata koji su proslijeđeni kao argumenti u pozivima raznih metoda, što pokazuje slijedeći kôd:

```
Class Args... {
    public static void checkNullPointerArgument(Object o, String message)
      throws IllegalArgumentException {
        if (o == null) throw new IllegalArgumentException(message);
    }

    public static void checkNullPointerArgument(Object o)
      throws IllegalArgumentException {
        checkNullPointerArgument(o, "Passed parameter is null.");
```

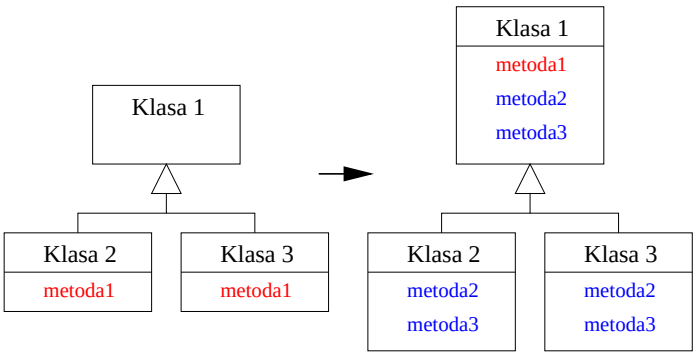
```
        return;
    }

    public static void checkNullPointerArgument(Object o1, Object o2)
      throws IllegalArgumentException {
        checkNullPointerArgument(o1);
        checkNullPointerArgument(o2);
        return;
    }
    ...
}
...

Class A... {
    public example (String string1, int integer1) {
        Args.checkNullPointerArgument(string1, integer1);
    }
    ...
}
```

Oblik predloška metode

U slučaju da su metode potklasa (engl. *subclasses*) slične ali nedovoljno slične da bi se mogle koristiti nasljeđivanjem od više klase u hijerarhiji, potrebno je izvući zajedničke dijelove kôda u posebnu klasu, a različite dijelove prepustiti donjim klasama (slika 6.3). Ovaj princip je poznatiji kao oblik predloška metode (engl. *Form Template Method*).

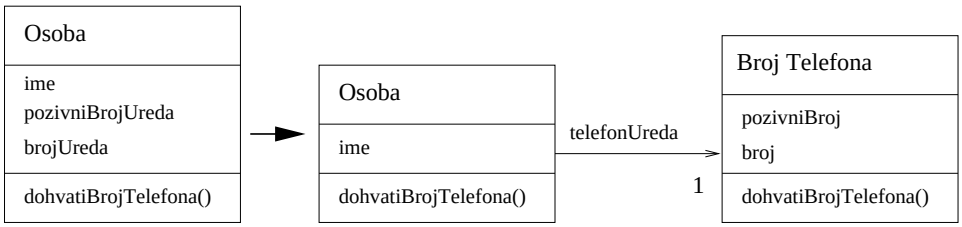


Slika 6.3: Oblik predloška metode (engl. *Form Template Method*)

Nasljeđivanje (engl. *inheritance*) je efikasno za eliminaciju dupliciranog ponašanja. Kad god su se uočile dvije slične metode u klasama, ono zajedničko se izdvajalo u posebnu metodu u klasi koja je u višoj hijerarhiji i koju obje klase iz kojih se izdvajala metoda nasljeđuju.

Izdvajanje klase

Slijedeći primjer je tehnika izdvajanja klase (engl. *Extract Class*). Jedna klasa obavlja posao koji logički trebaju obavljati dvije klase.



Slika 6.4: Izdvajanje klase (engl. *Extract Class*)

Potrebno je kreirati novu klasu i relevantni dio kôda preseliti iz stare klase u novu klasu (slika 6.4).

Općenito, poželjno je da klasa bude jasna apstrakcija koja rukuje s jasnim odgovornostima. U praksi, klase neprestano rastu. Kada se dodaje nova funkcionalnost u klasu, treba se pitati da li je potrebno kreirati novu klasu ili tu funkcionalnost dodati u već postojeću klasu, ali da postojeća klasa ostane smislena, tj. zaokružena cjelina.

6.2.2.2 Dugačka metoda

Naročitu vrijednost u objektno-orijentiranom razvoju softvera treba dati kratkim metodama.

Glavni princip korištenja kratkih metoda naglašava da je potrebno biti mnogo agresivniji kod postupka rastavljanja metoda, nego što se obično koristi u praksi. Heuristički, treba slijedi načelo da kad god treba komentirati nešto u kôdu, može se umjesto toga napraviti metoda. Takva metoda sadrži kôd koji je komentiran te se postupak rastavljanja metoda i kreiranja novih metoda može činiti na grupi linija kôda ili na samo jednoj liniji kôda, koliko je već potrebno.

Ključna nije dužina metode, već njena semantika, što metoda radi i kako radi. Kako bi se to omogućilo, u *HC Agent* projektu su primijenjene različite tehnike. Najveći dio vremena se provodila tehnika izdvajanja metode (engl. *Extract Method*) (odjeljak 6.2.2.1). Također, koristile su se i druge tehnike, kao što su: zamjena privremenih vrijednosti s upitom, uvođenje parametriziranog objekta, sačuvanje cijelog objekta, zamjena metode s metodom objekta i rastavljanje uvjeta u kôdu.

Zamjena privremenih vrijednosti s upitom (engl. *Replace Temp With Query*)

Često se može koristiti tehnika zamjene privremenih vrijednosti s upitom (engl. *Replace Temp with Query*) kojom se eliminira korištenje privremenih vrijednosti u programskom kôdu.

Slijedeći primjer pokazuje izdvajanje izraza u metodu. Sve reference na privremene vrijednosti u kôdu se mogu zamijeniti s pozivima metoda tako da te nove metode mogu biti korištene i u drugim metodama.

Ovaj kôd:

```
    double basePrice = _quantity * _itemPrice;
    if (basePrice > 1000)
        return basePrice * 0.95;
    else
        return basePrice * 0.98;
```

se zamjenjuje s ovim kôdom:

```
    if (basePrice() > 1000)
        return basePrice() * 0.95;
    else
        return basePrice() * 0.98;
    ...
double basePrice() {
    return _quantity * _itemPrice;
}
```

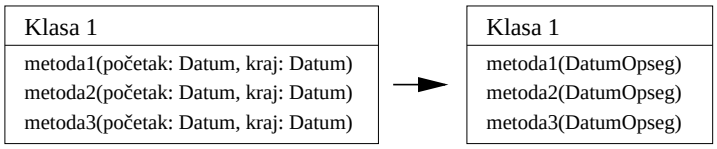
Problem s privremenim vrijednostima je da su one lokalne i privremene. Budući da egzistiraju samo u kontekstu metode u kojoj se koriste, obično su prisutne u duljim metodama jer su jedini način za pohranu privremenih vrijednosti. Zamjenom privremenih vrijednosti s metodama upita (engl. *query methods*), bilo koja metoda u klasi može dobiti tražene informacije. To pomaže čistoći kôda u klasi.

Ova tehnika je često korištena prije tehnike izdvajanja metode (odjeljak 6.2.2.1). Upotrebom lokalnih varijabli obično je otežano ekstrahiranje. Zbog toga ova tehnika često prethodi tehnici izdvajanja metode.

Uvođenje parametriziranog objekta

Dugačka lista parametara u pozivima metoda može biti smanjena uporabom tehnike uvođenja parametriziranog objekta.

Grupa parametara prirodno ide zajedno, kao što je prikazano (slika 6.5).



Slika 6.5: Uvođenje parametriziranog objekta (engl. *Introduce Parameter Object*)

Često se grupa parametara mogla grupirati zajedno čime se omogućuje da više različitih klasa može koristiti tu grupu. Time se smanjuje veličina liste parametara koji se prenosi. Kôd je konzistentan, lakše je razumijevanje i kasnije modificiranje.

Sačuvanje cijelog objekta

Dugačka lista parametara također može biti zamijenjena s tehnikom sačuvanja cijelog objekta (engl. *Preserve Whole Object*).

Dobivaju se nekoliko vrijednosti objekata i prosljeđuju te vrijednosti kao parametri u pozivi metode.

Ovaj kôd:

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

se zamjenjuje s ovim kôdom:

```
withinPlan = plan.withinRange(daysTempRange());
```

Ovaj tip situacije nastaje kada objekt prosljeđuje nekoliko podatkovnih vrijednosti jednog objekta kao parametre u pozivu metode. Problem je ako pozvani objekt kasnije zahtjeva nove podatkovne vrijednosti, potrebno je pronaći i promijeniti sve pozive prema toj metodi. Taj problem se može izbjeći tako da se prosljeđuje cijeli objekt koji sadrži podatke. Pozivani objekt tada može tražiti koje podatke želi iz proslijeđenog objekta.

Primjena ove tehnike u *HC Agent* projektu je omogućila veću čitljivost kôda. Pozivi metoda su puno robusniji i jednostavniji. Izbjegnut je problem dupliciranja parametara u pozivu metoda.
Uočeni su problemi oko ovisnosti (engl. *dependency*) kod korištenja ove tehnike. Prosljeđivanjem vrijednosti, pozvani objekt ima ovisnosti samo na varijable. Prosljeđivanjem objekata, pozvani objekt ima ovisnosti na prosljeđeni objekt čime je u nekim slučajevima pretjerano narušena struktura međuovisnosti objekata.
Zbog ovih razloga, ova tehnika je korištena u slučajevima kada se prosljeđivalo više parametara u pozivu metoda.

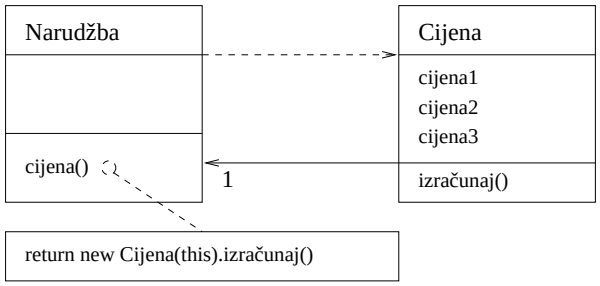
Zamjena metode s metodom objekta

Ponekad je slučaj da postoji dugačka metoda koja koristi lokalne varijable i ne može se koristiti tehnika izdvajanja metode (odjeljak 6.2.2.1).

Metoda se stavlja u poseban, vlastiti objekt tako da su sve lokalne varijable polja tog objekta. Nakon toga, može se rastaviti metoda u više metoda unutar istog objekta, kao što pokazuje slijedeći kôd i slika 6.6:

```
class Narudzba... {
    double cijena() {
        double cijena1;
        double cijena2;
        double cijena3;
        // long izracunaj;
        ...
    }
```

Problematika rastavljanja metode leži u njenim lokalnim varijablama. Ako su varijable brojne, postupak rastavljanja može biti složen. Korištenje tehnike zamjene privremenih vrijednosti s upitom (odjeljak 6.2.2.2) može biti pomoć u reduciranju privremenih varijabli.



Slika 6.6: Zamjena metode s metodom objekta (engl. *Replace Method with Method Object*)

Primjenom ove tehnike lokalne varijable se zamjenjuju poljima metode objekta. Može se upotrijebiti tehnika izdvajanja metode (odjeljak 6.2.2.1) na tom objektu kako bi se kreirale dodatne metode koje smanjuju originalnu metodu.

Rastavljanje uvjeta u kôdu

Sva stanja uvjeta u kôdu (engl. *conditionals*) te programske petlje (engl. *loops*) također su mogući kandidati za rastavljanje.

U slučaju da postoji kompliciran *if-then-else* izraz, može se izdvojiti metoda iz uvjeta, *then* dijela i *else* dijela.

Ovaj kôd:

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
      charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

se zamjenjuje s ovim kôdom:

```
if (notSummer(date))
      charge = winterCharge(quantity);
else charge = summerCharge (quantity);

private boolean notSummer(Date date) {
      return date.before (SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
      return quantity * _summerRate;
}

private double winterCharge(int quantity) {
      return quantity * _winterRate + _winterServiceCharge;
}
```

Jedan od najčešćih područja kompleksnosti softvera je upravo složenost *if-then-else* izraza i programskih petlji. Kada se piše kôd i testiraju takvi izrazi, obično se brzo završava s dugim metodama. Dužina metode je faktor koji ju čini težim za čitanje, a

time svakako doprinose i složeni *if-then-else* izrazi te programske petlje. Problem obično leži u činjenici da kôd pokazuje što se dogodilo, ali je obično nedovoljno poznato i zašto se to dogodilo. Sa ovakvim složenim izrazima i programskim petljama, može se lakše razjasniti korištenje pojedinih grana takvih izraza.

Kao s bilo kojim blokom kôda, tehnikom rastavljanja uvjeta u kôdu na posebne metode kôd je postao razumljiviji u *HC Agent* projektu.

6.2.2.3 Golema klasa

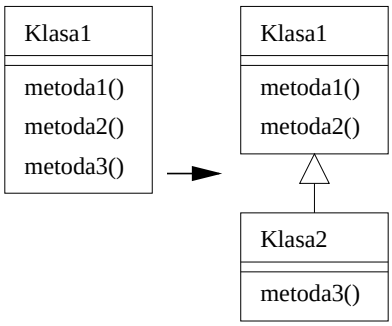
Kada klasa pokušava načiniti previše implementiranih zadataka, obično se ta preopterećenost s funkcionalnostima manifestira kao prisutnost previše instanci varijabli.

Može se koristiti tehnika izdvajanja metode (odjeljak 6.2.2.1) kako bi se eleminirao broj varijabli.

Često je jednostavnija upotreba tehnike izdvajanja potklase.

Izdvajanje potklase

Klasa ima karakteristike koje se koriste samo u nekim instancama. Može se kreirati potklasa za taj podskup karakteristika (slika 6.7).



Slika 6.7: Izdvajanje potklase (engl. *Extract Subclass*)

Glavni motiv za korištenje ove tehnike je realizacija klase da sadrži implementaciju koja se koristi za neku instancu klase, a ne za ostale. Obično ne treba pisati dodatni kôd prilikom izdvajanja klase u potklasu.

Glavna alternativa tehnike izdvajanja potklase je tehnika izdvajanja klase (odjeljak 6.2.2.1). Često se radi o svojevrsnoj odluci između delegacije i nasljeđivanja. Ova tehnika se pokazala jednostavnijom za uporabu nego tehnika izdvajanja klase, iako ima i određenih ograničenja. Ne može se mijenjati ponašanje klase jednom kada je kreiran objekt. Ponašanje klase se može promijeniti s tehnikom izdvajanja klase.

6.2.2.4 Dugačka lista parametara

Dio kôda u *HC Agent* softverskoj komponenti u pozivu drugih metoda je prosljeđivao kao parametre sve što je bilo potrebno kako bi pozvane metode uspješno obavljale funkcionalnost.

Problem koji se mogao uočiti kod upotrebe složenih metoda je dugačka lista parametara. Alternativa je prosljeđivanje globalnih podataka što bi trebala biti praksa objektno-orijentiranog programiranja; lista parametara je mnogo manja.

Dugačka lista parametara (engl. *Long Parameter List*) kompliciranа je za razumijevanje i teška za održavanje, naročito kada se mijenjaju (dodaju i/ili uklanjaju) parametri. Pokazalo se da se najviše promjena može eliminirati kada se, umjesto liste parametara, prosljeđuju objekti.

Ovaj problem se može riješiti s tehnikom zamjene parametra s metodom (odjeljak 6.2.2.4). Također može se koristiti i opisana tehnika sačuvanja cijelog objekta (odjeljak 6.2.2.2) te tehnika uvođenja parametriziranog objekta (odjeljak 6.2.2.2).

Preveliki broj parametara je, nakon primjene ove tehnike, zamijenjen s jednim ili nekoliko objekata koji se prosljeđuju kao argumenti poziva metoda u *HC Agent*-u.

Zamjena parametara s metodom

Tehnika zamjene parametara s metodom (engl. *Replace Parameter with Method*) se koristi kada se žele dobiti podaci u jednom parametru čineći poziv, odnosno prosljeđujući poznati objekt. Taj objekt može biti polje (engl. *field*), struktura (engl. *structure*) ili neki drugi parametar.

Ovaj kôd:

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

se zamjenjuje s ovim kôdom:

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Dugačke liste parametara su teške za razumijevanje i trebaju biti reducirane čim je moguće više.

6.2.3 Jedinično testiranje

Implementacija kôda koju preporučuje XP razvojna metodologija naglašava važnost implementacije jediničnih testova prije implementacije sâmog kôda (engl. *Code the Unit Test First*) (odjeljak 5.3.3).

Stoga je u razvojnom projektu implementacije *HC Agent* softverske komponente isprobana tehnika jediničnog testiranja.

Tipični alati, odnosno okružja kojima je bilo moguće vršiti jedinično testiranje su bili *JUnit* [35] za J2SE [36] ili J2EE [37] platformu i *csUnit* [38] za *Microsoft .NET* platformu [39] [30]. Korištenjem ovih alata za testiranje bilo je moguće automatizirano izvršavati jedinične testove iz IDE razvojnog okružja.

Oba alata su našla primjenu u *HC Agent* razvojnom projektu, iako su namijenjeni za različite platforme. Na početku razvoja *HC Agent*-a vršila se implementacija HL7

podatkovnih tipova. Ti podatkovni tipovi su implementirani na J2SE [37] platformi pomoću *IntelliJ IDEA* [57] IDE razvojnog okružja. Za jedinično testiranje se koristio *JUnit* [35] koji se može pokretati iz sâmog IDE-a.
U kasnijoj fazi projekta, kada se vršila implementacija komponenata na *Microsoft .NET* platformi [39], koristio se *csUnit* [38] kao programski dodatak koji je integriran sa *Microsoft Visual Studio .NET* razvojnim okružjem.
Ostali podsustavi *HC Agent* sustava su u kasnijem slijedu projekta također implementirani u *Microsoft .NET* tehnologiji. Za jedinično testiranje tih podsustava se također koristio *csUnit*.

Korištenje *csUnit*-a i *JUnit*-a je vrlo slično, tj. ti alati koriste gotovo identičan način implementacije jediničnih testova i izgledom su slični.

6.2.3.1 Implementacija testova

Razvoj *HC Agent* sustava je tekao paralelno s razvojem središnjeg sustava zdravstva. Glavni zadatak u testiranju je bio osmišljavanje i razvoj testnog okružja (engl. *testing framework*) za testiranje HL7 podatkovnih tipova koji su zajednički *HC Agent* komponenti i središnjem sustavu zdravstva. HL7 podatkovni tipovi su okosnica tih dvaju sustava.

Imajući u vidu ograničenja *JUnit* programskog alata za testiranje, odluka je bila koristiti *Rational TestManager* [58] zajedno s *JUnit* [35] alatom.
Rational TestManager je alat za planiranje testova, izvršavanje testova, analizu rezultata testova, statistiku i izvještavanje o rezultatima testova. Sadrži vrlo važnu karakteristiku povezivanja zahtjeva sustava i pojedinih testova.
Funkcijski plan testiranja je bio baziran na: zahtjevima slučajeva uporabe (zahtjevi koje sustav mora ispunjavati), dodatnim specifikacijama te HL7v3 *ballot* 6 [47] specifikaciji.

Testovi ostalih podsustava u *HC Agent* razvojnom projektu su vršeni u *csUnit* testnom okružju, budući da je za razvoj korištena *Microsoft .NET* tehnologija (*Microsoft Visual C#* programski jezik).
U kasnijem dijelu projekta, nakon razvoja HL7 podatkovnih tipova i ostalih komponenata, zbog potrebe integracije *HC Agent* sustava te cjelokupnog testiranja i prve isporuke, HL7 podatkovni tipovi su prebačeni iz Java programskog jezika na *Microsoft Windows .NET* platformu pomoću *Microsoft Visual J#* alata. Od tada se za testiranje koristilo isključivo *csUnit* testno okružje.

Imajući u vidu potrebu za dokumentiranjem testova, testne klase su u pravilu dokumentirane u vidu komentara koji su pisani prije implementacije svake pojedine testne metode.

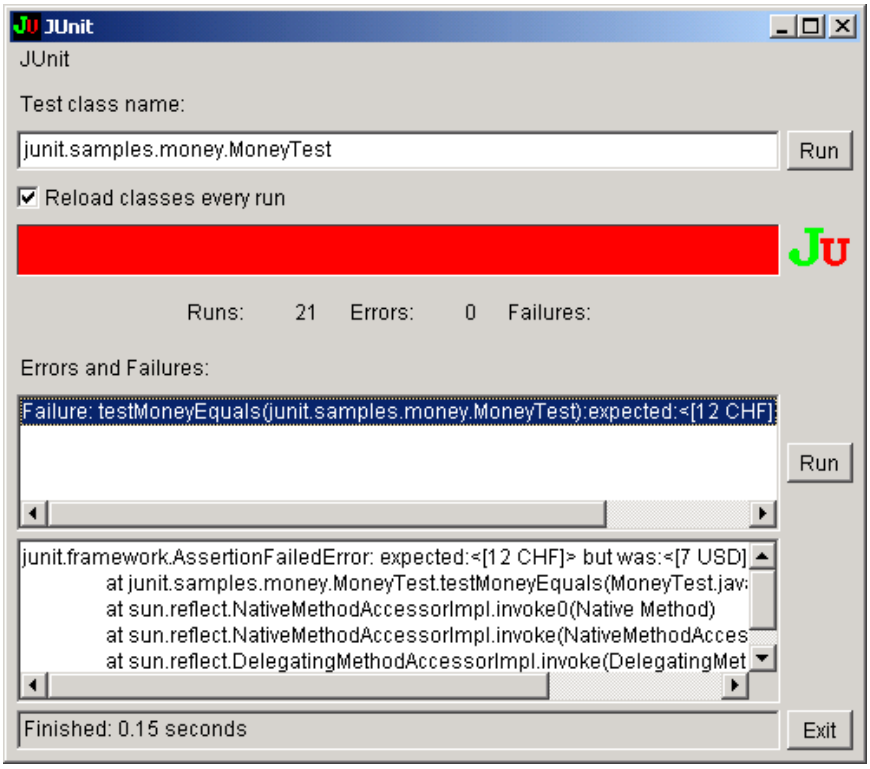
6.2.3.2 *JUnit* okružje za testiranje

Odlučeno je koristiti *JUnit* okružje za implementaciju ponovljivih testova u Java [36] [37] programskom jeziku koje ovisi o Java SDK (engl. *Software Development Kit*, alat za razvoj softvera).

U razvoju softvera, jedinica (engl. *unit*) se može odnositi na klasu, paket (engl. *package*), podsustav (engl. *subsystem*) ili sustav (engl. *system*).

Prvi zadatak u korištenju *JUnit*-a je bilo imati tračak nade da će razvojni tim *HC Agent* projekta implementirati testove i kontinuirano provjeravati vlastiti kôd.

Drugi zadatak u korištenju *JUnit*-a je bilo kreiranje testova koji zadržavaju svoje vrijednosti kroz vrijeme. Netko drugi, tko nije originalni autor testa, morao je biti u mogućnosti izvršiti testove i interpretirati rezultate. To je obično bio posao testera. Također, tester je trebao proširivati osnovne testove koje vrše razvijatelji kako bi provjerio kompleksniju funkcionalnost modulâ i integraciju različitih modula u jedan sustav. Bilo je moguće kombinirati testove različitih autora i pokretati ih zajedno bez straha od interferencije.



Slika 6.8: *JUnit* testno okružje

Prema XP metodologiji razvoja softvera, posao programera nije završen dok nije napisan i provjeren kôd. Programeri trebaju napisati testove kojima dokazuju da njihov kôd radi ali i dokumentirati kako radi. Ulazni podaci za testove su zahtjevi sustava.

Slika (6.8) prikazuje grafičko sučelje *JUnit* testnog alata koji pokreće testove. U gornjem dijelu se vidi pokrenuta testna klasa (engl. *Test class name*, *junit.samples.money.MoneyTest*) koja sadrži testove. Ispod nje se može nalaziti crvena ili zelena traka. Crvena traka na slici označava da neki od testova (jedan ili više) iz testne klase nisu prošli. Zelena traka označuje da su svi testovi iz testne klase prošli. U donjem dijelu nazale se podaci o greškama ukoliko ih ima (engl. *Error and Failures*).

U implementaciji HL7 sučelja programeri su vršili implementaciju kôda te osnovne jedinične testove kojima su dokazali da kôd radi. Jedinični testovi su ujedno pokazivali i kako kôd radi, tj. kako se kôd ispravno koristi. Testovi su, u pravilu, pokrivali samo

osnovnu funkcionalnost, kritične i kompleksne dijelove kôda, te pomogli i ubrzali implementaciju. Testeri su, nakon završene početne implementacije dijela funkcionalnosti HL7 podatkovnih tipova, proširivali početni skup testova s novim testovima i provjeravali sve ono što u osnovnim testovima nije bilo pokriveno ili je bilo nedovoljno provjereno.

JUnit okružje za testiranje je besplatno i bazirano na opće prihvaćenim obrascima dizajna (engl. *design patterns*): *Command*, *Template Method*, *Collecting Parameter*, *Adapter*, *Pluggable Selector* te *Composite pattern*. Korištenjem *JUnit* okružja razvjatelji eksplicitno koriste obrasce dizajna što vjerojatno nije slučaj kada koriste vlastita testna okružja.

U *HC Agent* projektu, *JUnit* testovi su organizirani u posebne Java pakete tako da su odvojeni od izvornog kôda sâme aplikacije. Na taj način, mogao se isporučiti konačni produkt (originalne klase) bez testova. Ako je kupac želio provjeriti implementaciju proizvoda prema zahtjevima, mogli su mu se demonstrirati testovi kako bi se uvjerio da je implementacija bila u skladu sa zahtjevima. Obično, kupac provjerava softver prilikom isporuke pokretanjem testova prihvaćenosti koji su testovi visoke razine i demonstriraju traženu funkcionalnost.

Testovi su dizajnirani na slijedeći način. Za svaku klasu izvornog kôda (engl. *source class*) implementirana je testna klasa (engl. *test class*) koja provjerava funkcionalnost izvorne klase. Svaka *JUnit* klasa se sastoji od statičke *Suite* metode, što pokazuje slijedeći primjer:

```
/**
 * Description: Suite Method that uses reflection to dynamically
 * create a test suite containing all the testXXX() methods.
 * @return TestSuite
 */
 public static Test suite() {
     return new TestSuite(TC_BN.class);
 }
```

Suite metoda je slična *main* metodi koja je specijalizirana za pokretanje testova u tekstualnom môdu, što pokazuje slijedeći primjer:

```
/**
 * Description: Main Method for running the test with the textual test
 * runner.
 * @param args String[]
 */
 public static void main (String[] args) {
     junit.textui.TestRunner.run(suite());
 }
```

Suite metoda dinamički kreira slijed (garnituru) testova (engl. *test suite*) koja sadrži kolekciju svih testova iz testnih metoda u pridruženoj testnoj klasi. Isto tako, *suite* metoda može pozivati i više testnih klasa u slučaju kad se želi pokrenuti kolekcija više testnih klasa koje sadrže testne metode, što pokazuje slijedeći primjer:

```
import junit.framework.*;
public class TCrun {
```

```
    public static Test suite() {
        suite = new TestSuite();

        suite.addTest(TC_CLASS_A.suite());
        suite.addTest(TC_CLASS_B.suite());
        suite.addTest(TC_CLASS_C.suite());

        return suite;
    }

    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Slijedi primjer kôda (originalne klase) (dodatak A.1) i testne klase za taj kôd (dodatak A.2).

Testna klasa treba naslijediti klasu *TestCase* koja se nalazi unutar *junit.framework* Java paketa koji se isporučuje s *JUnit* alatom. Imena testnih metoda trebaju započinjati sa riječi *test*, npr: *testCreateObject()*, *testEQUAL()* i slično. Na *Assert* objektu su definirane različite *assert()* metode i to su metode koje se najviše koriste prilikom testiranja.
Na primjer, pomoću *Assert.assertEquals()* metode uspoređuje se vrijednost dva objekta. Ukoliko je vrijednost objekata jednaka, test prolazi. Ako je vrijednost objekata različita, test ne prolazi što se može detektirati sa crvenom linijom na grafičkom sučelju.
Slijedeći primjer *assert()* metode je *Assert.assertNotNull()* metoda koja provjerava da li objekt ima *NULL* vrijednost (tj. da li objekt postoji), čime se uspješno provjeravaju konstruktori klase koja se testira.
Postoje i specijalne *setUp()* i *tearDown()* metode koje prvenstveno služe za inicijalizaciju objekata te za oslobađanje tih objekata. Važno je naglasiti da se te metode pozivaju u slijedećem redoslijedu, dakle prije i poslije svake testne metode u testnoj klasi:

```
setUp()
testXX()
tearDown()

...

setUp()
testYY()
tearDown()
```

6.2.3.3 Integracija *Rational TestManger*-a i *JUnit*-a

Jedan od glavnih zadataka testiranja je izrada i izvršavanje testova koji su ponovljivi u vremenu. Kada se uvodi nova funkcionalnost u postojeći kôd, nužno je ponovno pokrenuti postojeće testove kako bi se uvjerilo da nove promjene u kôdu nisu uvele nove greške i neispravnosti u već implementiranu i testiranu funkcionalnost. *JUnit* testno okružje to omogućuje.

U *HC Agent* razvojnom projektu, HL7 podatkovni tipovi, koji su ujedno i okosnica čitavog sustava, napisani su u Java programskom jeziku. Testovi za komponentu su

napisani u *JUnit* testnom okružju. Glavna ideja je bila organizirati testove kako bi se mogli koristiti u *Rational TestManager* programskom alatu.
Rational TestManager je otvoreno i proširivo okružje koje ujedinjuje sve ono što je vezano uz testiranje. Podržava glavne testne aktivnosti: planiranje testova, dizajn testova, implementaciju testova, izvršavanje testova i ocjenjivanje zahtjeva koje softver treba zadovoljavati.

Osnovna karakteristika *JUnit*-a je implementacija i pokretanje testova. Glavni nedostatak *JUnit* okružja je nemogućnost spremanja rezultata prethodno izvršenih testova i prikaz statističkih podataka (npr. spremanje rezultata prethodnih testova po datumima).

Zbog navedenih ograničenja *JUnit*-a, bilo je potrebno uvesti novi alat za testiranje koji omogućuje mjerenje testnih rezultata i praćenje promjena [59]. Rješenje je bio *Rational TestManager* koji ima mogućnost spremanja rezultata prije izvršenih testova i tako imati statistiku (pratiti poboljšanja) u razvoju softvera. Također, *Rational TestManager* ima mogućnost izvršavanja testova na različitim računalima u lokalnoj mreži. To je važno zbog prirode softvera koji je u razvoju.

Prva aktivnost u pripremi testova je bila "učenje" *Rational TestManager*-a da koristi *JUnit* testne klase, budući da *Rational TestManager* ne može izravno koristiti *JUnit* klase. Za svaku *JUnit* testnu klasu koja sadrži testne metode i slijed testova (engl. *test suite*), razvijena je posebna prilagodna klasa koja učitava slijed testova (dodatak A.3). Budući da slijed testova dinamički pokreće sve testne metode iz dotične klase, rezultat je kolekcija svih testova u *Rational TestManager* testnom okružju.

Nakon izvršavanja testova, *Rational TestManager* prikazuje rezultate testiranja i sprema rezultate. Testovi su vizualno označeni crvenom (testovi nisu prošli) i zelenom bojom (testovi su prošli), isto kao i sučelje *JUnit*-a. Na taj način, kada se mijenjaju testovi i/ili originalni kôd, testovi mogu biti izvršavani bez promjene kôda u *Rational TestManager* prilagodnoj klasi. Ovaj postupak je znatno olakšao i ubrzao postupak testiranja.

Provjereno je da povezivanje, tj. integracija *csUnit*-a i *Rational TestManager* testnog okruženja trenutno nije moguća. Korištena verzija *Rational TestManager* nije podržavala *Microsoft .NET* platformu.

6.2.3.4 *csUnit* okružje za testiranje

Nakon prelaska na *Microsoft .NET* platformu, u projektu je korišteno *csUnit* [38] testno okružje. Već sa prvim susretom, vidjelo se da je način korištenja (sintaksa) i izgled (grafičko sučelje) vrlo sličan *JUnit* testnom okružju.

Nakon završene implementacije HL7 podatkovnih tipova u Java programskom jeziku, ostale komponente su rađene na *Microsoft .NET* platformi. HL7 podatkovni tipovi prebačeni su na *Microsoft .NET* platformu pomoću *Microsoft Visual J#* programskog alata.

Budući da nije bilo moguće integrirati *Rational TestManager* i *csUnit*, nije bilo moguće niti koristiti planiranje testiranja te statističku obradu rezultata nakon početnih uspjeha u implementaciji HL7 podatkovnih tipova.

Paralelno s razvojem ostalih komponenata *HC Agent* sustava, testeri su, umjesto

proširivanja skupa testova sa dodatnim testovima koji testiraju podsustave *HC Agent*-a, implementirali jednostavne testne aplikacije s grafičkim sučeljem. Te aplikacije su svojom funkcionalnošću omogućavale provjeru funkcionalnosti *HC Agent* komponente (i ostalih implementiranih podsustava) te središnjeg sustava zdravstva s kojim klijenti korištenjem *HC Agent* komponente komuniciraju. Te aplikacije predstavljaju klijentske aplikacije koje koristite komponentu. Aplikacije predstavljaju i svojevrsnu kolekciju funkcijskih testova za *HC Agent* komponentu i ostale podsustave *HC Agent* programskog sustava. Također, aplikacije služe i za izvođenje testova prihvaćenosti kod kupca.
Budući da je *HC Agent* programska *Microsoft .NET* komponenta, odlučeno je da jedna testna aplikacija koja koristi komponentu bude razvijena kao *Microsoft .NET* aplikacija (dakle, u *Microsoft .NET* razvojnom okružju). Zbog lakoće korištenja koristio se *Microsoft Visual Basic .NET* [60] programski jezik za razvoj testne aplikacije.
Iz razloga što neki klijenti ne koriste *Microsoft .NET* platformu za razvoj klijentskih aplikacija, *HC Agent* može biti korišten s COM tehnologijama koje omogućuju pristup *HC Agent*-u s COM *Collable Wrapper* (CCW) sučeljem. Iz tog razloga je razvijena i posebna testna aplikacija pomoću *Microsoft Visual Basic 6.0* programskog jezika kako bi se provjerio rad *HC Agent*-a s CCW sučeljem i klijentima koji koriste COM tehnologiju.
Izgledu grafičkog sučelja aplikacija za testiranje je pridodana minimalna pažnja. Namjena grafičkih aplikacija je prvenstveno mogućnost demonstracije implementirane funkcionalnosti te mogućnost izvršavanja testova prihvaćenosti.

Testeri su, dakle, bili zauzeti sa drugim oblicima testiranja koji su obuhvaćali implementaciju testnih aplikacija. Programeri su, osim zahtijevanih osnovnih jediničnih testova pri implementacije HL7 podatkovnih tipova, bili obavezni proširiti početni osnovni skup testova sa cjelokupnim skupom testova, koji ne provjeravaju samo osnovnu funkcionalnost i kritične dijelove kôda, već cjelokupnu funkcionalnost.

Slijedi primjer kôda (testne klase) (dodatak A.4).

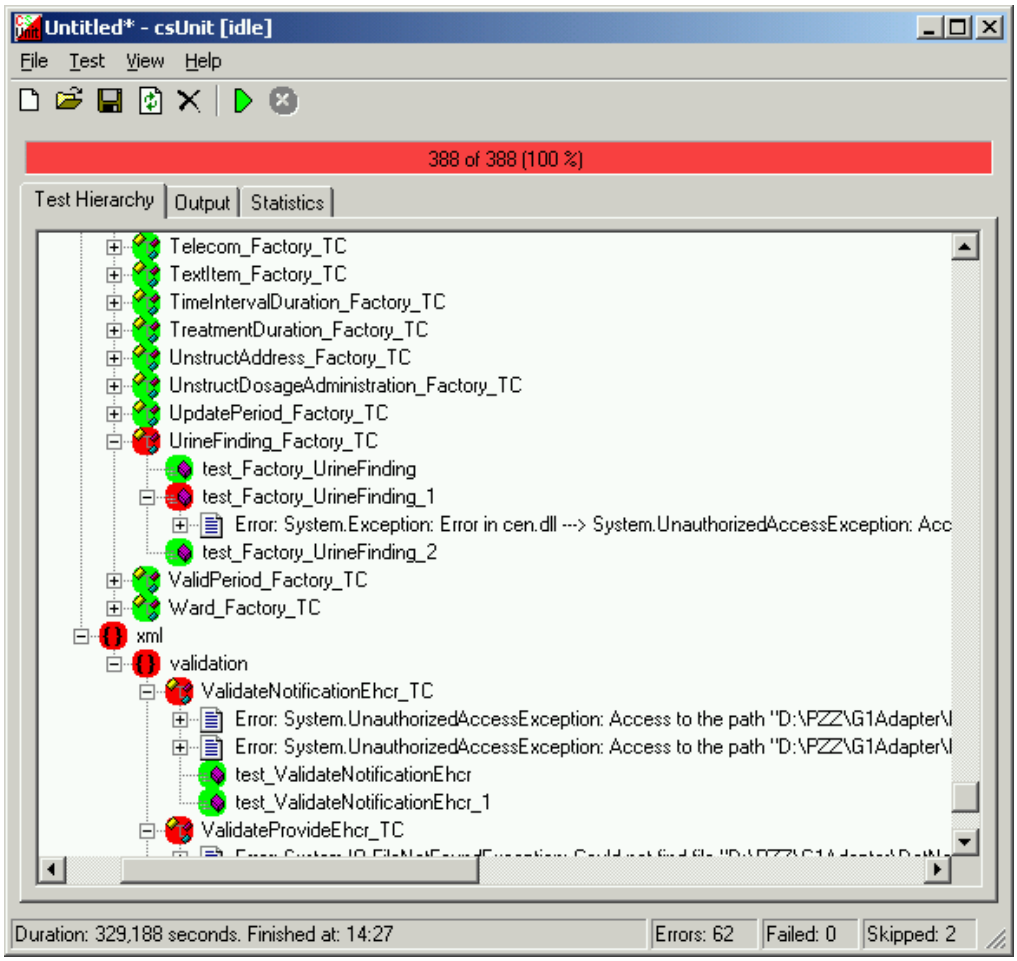
Prva važna stvar kod kreiranja testne klase je *[TextFixture]* atribut, što pokazuje slijedeći isječak kôda:

```
using System;
using csUnit;
// Don't forget to add the csUnit.dll reference to your testing project

namespace example {
    [TestFixture]
    public class FooTests() {
        public FooTests() {
            // TODO: Add constructor logic here
        }
    }
}
```

Uloga navedenog atributa je identificiranje klasa koje su testne, dakle, određivanje klasa koje sadrže testne metode.

Slika (6.9) prikazuje *csUnit* testno okružje koje je funkcijski i vizualno vrlo slično *JUnit* okružju (slika 6.8). Na vrhu se nalazi zelena ili crvena traka. Zelena traka označava da su pokrenuti testovi prošli. Crvena traka na slici označava da neki od testova (jedan



Slika 6.9: *csUnit* testno okružje

ili više) nisu prošli. Ispod toga se nalazi grafički prikaz testnih klasâ koje sadrže testne metode. Zelena boja uz testnu metodu označava da je metoda uspješno izvršena dok crvena boja označava grešku. Uz crvenu boju se nalazi i opis greške.

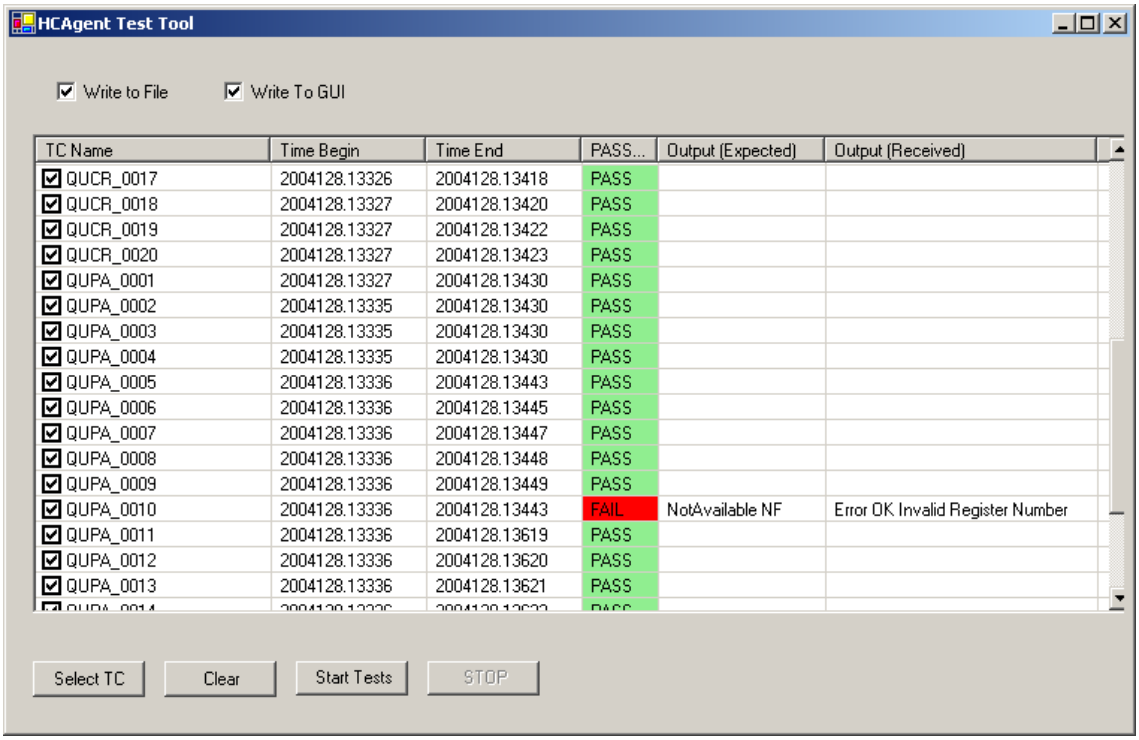
Programeri i testeri u *HC Agent* projektu su općenito bili zadovoljniji s načinom prikaza grešaka kod *csUnit* testnog okružja nego kod *JUnit*-a. Razlog je u jednostavnosti prikaza grešaka. U *csUnit*-u je odmah uočljivo koji test (testna metoda) nije prošla (crvena boja uz metodu), dok kod *JUnit*-a treba čitati tekst u prozoru s opisom greške.

6.2.3.5 Potreba za razvojem novog testnog okružja

Za vrijeme implementacije i testiranja *HC Agent* komponentnog sustava, došlo se do zaključaka da implementirane testne aplikacije nisu dovoljno efikasne za testiranje.

Potvrdilo se da su aplikacije dovoljno efikasne tek kako bi se njima ispitala osnovna funkcionalnost *HC Agent* programskog proizvoda. Aplikacije pokrivaju jedan dio funkcijskih testova sustava i ujedno služe kao alat za pokretanje testova prihvaćenosti kod kupca.

Zahtjev za testiranjem je bilo isprobati performanse centralnog sustava i komponente koji moraju zadovoljiti istovremeni rad velikog broja klijentskih aplikacija koje komuniciraju sa centralnim sustavom zdravstva. Jedna testna aplikacija *HC Agent*-a, zbog načina pojedinačne izmjene HL7/CEN XML poruka sa centralnim sustavom, predstavlja samo jednog klijenta sustava.



Slika 6.10: *HCAgentTestTool*

Rješenje koje se prirodno nametalo je bilo implementirati višestruko slanje poruka. Prva ideja je bila iskoristiti postojeći *csUnit* alat za testiranje. Zamisao je bila da svaka testna metoda bude implementirala slanje jedne poruke. Problem koji se javio je bio u implementiranom asinkronom načinu rada *HC Agent* komponente.
Komponenta asinkrono prima podatke od jednog ili više klijenata, od tih podataka generira HL7/CEN XML poruke i te poruke izmjenjuje sa centralnim sustavom zdravstva od kojeg uzima rezultate kada su oni dostupni. Kod *csUnit* alata uočena je nemogućnost asinkronog načina rada. Slijedeća poruka se može slati tek kada je obrađena prethodna poruka (poslana poruka i primljen odgovor nakon obrade poruke u centralnom sustavu). Takav način rada *csUnit* alata je sinkroni način rada i zbog takvog načina rada nije bilo moguće koristiti *csUnit* alat za testiranje.

Slijedeći problem koji se javio je bio problem regresijskog izvršavanja testova iz implementiranih klijentskih aplikacija. Zbog iznimno velikog broja testnih podataka koje je potrebno unijeti u testnu aplikaciju kako bi ona proslijedila te podatke *HC Agent* komponenti, bilo je mukotrpno i vremenski vrlo zahtjevno ponavljanje testova.

Zbog prethodnih razloga, odlučeno je implementirati specifično testno okružje za *HC Agent* komponentu i centralni sustav zdravstva (slika 6.10). Specifično testno okružje je

implementirano tako da:

1. Omogućuje asinkrono slanje podataka *HC Agent* komponenti. Komponenta asinkrono šalje poruke centralnom sustavu zdravstva i traži odgovore za poslane poruke. Dobivene odgovore komponenta prosljeđuje testnom okružju. Testno okružje vodi računa o vremenu kada je poslalo podatke komponenti, vremenu kada je dobilo odgovor od komponente, sadržaju odgovora koje je dobilo te o porukama za koje nije uspjelo dobiti odgovore.

2. Omogućuje spremanje testnih podataka u XML datotekama kako bi se riješio problem regresijskih testova.

3. Omogućuje spremanje rezultata testiranja u tekstualne datoteke kako bi postojali statistički podaci u uspješnosti testova.

Grafičko sučelje testnog okružja je dizajnirano po uzoru na *JUnit* i *csUnit* alate za testiranje. Bojama je označeno koji testovi su prošli (zelena boja) i koji testovi nisu prošli (crvena boja).

6.2.4 Male i česte isporuke

Zadaća XP razvojnog tima je da često isporučuje iterativne verzije sustava koji je u procesu razvoja naručitelju (odjeljak 5.1.3).

U postupku planiranja razvoja (odjeljak 5.1) u početnoj fazi *HC Agent* razvojnog projekta, vršila se aktivnost prikupljanja zahtjeva koji su se oblikovali u temeljne slučajeve uporabe *HC Agent* sustava a koji predstavljaju osnovu funkcionalnost koju sustav treba zadovoljavati.

Unutar radne organizacije u kojoj je razvijen *HC Agent*, forma korisničkih priča nije niti uobičajena niti dovoljna što se tiče količine informacija i formata. Umjesto korisničkih priča, koristili su se *Use Case Specification* dokumenti koji vrlo detaljno opisuju funkcionalnosti koje su uobličene u jedan slučaj uporabe. *Use Case Specification* dokumenti obično daju detaljan opis slučajeva uporabe (engl. *Brief Description*), opis primarnog scenarija (engl. *Basic Flow*) i jednog ili više sekundarnih scenarija (engl. *Alternative Flows*) koji se mogu dogoditi s dotičnim slučajem uporabe.

U postupku planiranja u *HC Agent* razvojnom projektu, vršilo se planiranje isporuke (engl. *Release Planning*, odjeljak 5.1.2) na nivou čitavog *HC Agent* projekta. Osnovne funkcionalnosti su izdvojene u planiranju i služile su kako bi se kreirao grubi plan isporuke koji je vrijedio za čitav projekt.

HC Agent tim je u dogovoru s potencijalnim kupcem odredio da je idealni vremenski interval za izradu isporuke (tj. dijela proizvoda koji se isporučuje kupcu) svaka dva tjedna. Određen je upravo taj period jer se pokazalo da je za projekt takve veličine (osam osoba) dva tjedna idealno razdoblje kako bi se isporučio upravo optimalan broj implementiranih i testiranih novih funkcionalnosti.

Određeni period isporuke od dva tjedna je u skladu s prosječnim trajanjem iteracija. Obično je svaka iteracija u dužini od jednog do tri tjedna i nije unaprijed fiksno određeno koliko će biti duga i koje funkcionalnosti će se implementirati.

U XP metodologiji razvoja softvera, određeno je da se plan isporuke koristi za kreiranje plana iteracijâ za svaku pojedinu iteraciju. Individualne iteracije su detaljno planirane prije početka svake iteracije i nikako ne u naprijed. U iterativnom razvoju, razvoj softvera se može podijeliti u raspored u oko desetak iteracija a jedna iteracija traje u prosjeku od jednog do tri tjedan.
U *HC Agent* razvojnom projektu, u pojedinu iteraciju varijabilnog trajanja (od jednog do tri tjedna) implementirao se prethodno dogovoreni skup funkcionalnosti. Varijabilno trajanje iteracija obično nije bilo u suprotnosti s konstantnim vremenom isporuke.

Prema XP-u, u iteraciju ulaze one korisničke priče za koje je naručitelj odredio da su višeg prioriteta uz poznate procjene koje donose programeri iz razvojnog tima.
U *HC Agent* razvojnom projektu, funkcionalnost koja je ulazila u pojedinu iteraciju nije bila određena prioritetima od strane kupca, već funkcionalnošću koja je prethodno implementirana na središnjem sustavu zdravstva na koji se *HC Agent* komponentni sustav spaja. Dakle, nakon implementacije određene funkcionalnosti na središnjem sustavu zdravstva, vršila se implementacija iste funkcionalnosti na *HC Agent* komponentnom sustavu. Ovisno o količini funkcionalnosti koja je trebala biti implementirana, radile su se podjele tih funkcionalnosti u iteracije. Iako XP metodologija razvoja softvera predviđa desetak iteracija prosječnog trajanja od jednog do tri tjedna, *HC Agent* razvojni projekt ih je imao dvostruko više zbog vrlo velikog skupa funkcionalnosti koja fizički nije mogla biti podijeljena u manji broj iteracija.

6.3 Rezultati primjene postupaka metode ekstremnog programiranja na projekt razvoja javne elektroničke usluge

Ovo poglavlje opisuje iskustvo uvedenih postupaka XP razvojne metodologije u opisani *HC Agent* razvojni projekt.

Zaključuju se mogućnosti primjene opisanih razvojnih postupaka XP-a od projektnog tima i od organizacije u kojoj se obavljala implementacija projekta. Ocjenjuje se utjecaj primijenjenih praksi na status i napredak projekta.

Uspoređuju se postupci XP razvojne metodologije primijenjene na opisani *HC Agent* razvojni projekt sa sličnim implementiranim postupcima drugih razvojnih projekata koji ne koriste postupke XP metodologije u organizaciji.

Zaključci su izneseni u slijedećim područjima: periodu promatranja projekta (odjeljak 6.3.1), strukturi tima (odjeljak 6.3.2), karakteristikama softvera (odjeljak 6.3.3) i postupcima XP-a (odjeljak 6.3.4).

6.3.1 Period promatranja projekta

U *HC Agent* razvojnom projektu, promatranje primjene određenih praksi XP metodologije je vršeno u periodu nešto kraćem od pola godine. Pratila se cijela faza implementacije funkcionalnosti (odjeljak 5.5.3) koja je prethodila fazi isporuke (odjeljak 5.5.4) programskog proizvoda.

Početna faza istraživanja (odjeljak 5.5.1) nije uključena u promatranje.
Prema XP-u, u fazi istraživanja se određuju korisničke priče, odnosno skup funkcionalnosti koji će se implementirati. To se vrši u dogovoru s naručiteljem.

U *HC Agent* razvojnom projektu, fazu istraživanja nije bilo moguće provesti niti analizirati zbog specifičnosti sâmog *HC Agent* programskog proizvoda. Funkcionalnost *HC Agent* programskog proizvoda prvenstveno je ovisila o skupu funkcionalnosti centralnog sustava zdravstva, a ne o dogovoru s naručiteljem *HC Agent* programskog proizvoda. Iz tog razloga nije bilo moguće promatrati ovu fazu sa stajališta XP razvojne metodologije. *HC Agent* programski proizvod je razvijan u skladu s implementacijom funkcionalnosti centralnog sustava zdravstva.

U fazi implementacije funkcionalnosti, isprobani su određeni postupci XP razvojne metodologije s naglaskom na programski proizvod.

6.3.2 Struktura tima

Razvojni tim *HC Agent* projekta je bio relativno mali. Brojio je osam članova. Taj broj je uključivao četiri programera, dva testera, tehničkog voditelja razvoja i voditelja projekta.

Osnovni razlozi malog broja ljudi koji su bili pridijeljeni projektu su bili:

1. ograničeni resursi (raspoloživi ljudi i pridijeljen budžet)
2. nepoznati rokovi isporuke *HC Agent* programskog proizvoda.

Zbog malog broja ljudi uključenih u projekt, bilo je moguće isprobati karakeristične postupke XP metodologije u *HC Agent* razvojnom projektu.

Mali broj ljudi u projektu se pokazao idealnim. Razlog tome je bilo jednostavnije nadziranje napretka projekta.

Mali broj ljudi u projektu omogućuje dovoljan (no ne predetaljan) uvid u poslove svakog pojedinca (odnosno pâra). Bilo je moguće mnogo lakše ranije identificati probleme i na vrijeme učiniti potrebne akcije da se ti problemi riješe.

Glavni nedostatak koji je primijećen u strukturi *HC Agent* tima tijekom izvođenja projekta je nepostojanje krucijalne osobe sa ulogom arhitekta sustava. Takva uloga postoji, ali na razini cjelokupnog projekta implementacije informacijskog sustava zdravstva.

U projektima koji se izvode u istoj organizaciji, obično sudjeluje veći broj osoba uključenih u projektne timove. Takvi timovi često broje i preko trideset ljudi.

Veći broj projektnih timova nerijetko uzrokuje probleme, među kojima valja izdvojiti otežan nadzor i koordinaciju projekata te otežanu identifikaciju problema.

Važno je napomenuti da se u organizaciji u kojoj je implementiran ovaj razvojni projekt mogu identificirati određene projektne uloge. Te uloge su svojstvene organizaciji (odnosno organizacijskim zahtjevima), no mogu se usporediti sa klasičnim ulogama XP metodologije razvoja softvera.

6.3.2.1 Voditelj projekta

Jedna osoba iz *HC Agent* projektnog tima je bio voditelj projekta (engl. *Project Leader*). Ova uloga nije svojstvena XP timu, već je zapravo odlika radne organizacije gdje se vrši implementacija projekta. Voditelj projekta je obavljao još i neke poslove koje su specifični ovom razvojnom projektu. Ova projektna uloga se može povezati s nekoliko različitih uloga iz XP-a i to su:

- naručitelj (odjeljak 5.6.2)
- menadžer (odjeljak 5.6.7)
- tragač (odjeljak 5.6.4)
- trener (odjeljak 5.6.5).

U opisanom *HC Agent* razvojnom projektu, zahtjevi koje je trebalo implementirati nisu bili dani u formi korisničkih priča, već u formi dokumenta softverskih zahtjeva (engl. *Software Requirements Document*). Razlog tome je što je ovaj projekt zapravo potprojekt te dio većeg projekta implementacije cjelokupnog informacijskog sustava zdravstva. Ekstremno programiranje nije niti preporučljivo u velikim projektima. Razlog je i postojanje potrebe za dobro definiranom i opsežnom dokumentacijom. Dokument softverskih zahtjeva je svakako dio takve dokumentacije. Organizacija u kojoj se izvodio projekt je zahtijevala izradu opsežne dokumentacije tako da forma kratkih korisničkih priča za izražavanje zahtjeva nije zadovoljavala niti u opsegu niti u politici organizacije.

Zbog ovih razloga, uloga naručitelja u opisanom *HC Agent* razvojnom projektu je ipak bila drugačija nego u klasičnom XP projektu gdje je naručitelj izravno komunicirao s kupcem kojega je predstavljao razvojnom timu. U *HC Agent* projektu, potencijalni kupac nije bio dovoljno vidljiv razvojnom timu i nije mogao određivati prioritete korisničkih priča. U projektu su zahtjevi bili unaprijed poznati i u pravilu se nisu mijenjali. Podložno promjenama je bilo područje implementacije funkcionalnosti, budući da su u projektu isprobane i korištene različite platforme.

Voditelj projekta je kontrolirao da li su zahtjevi, koji su bili unaprijed određeni, ispravno implementirani. To su mu prema potrebi demonstrirali testeri pomoću implementiranih aplikacija s grafičkim sučeljem kojima su izvršavali testove prihvaćenosti. Testeri su implementirali opisane aplikacije za provođenje testiranja. Također, voditelj projekta je vodio računa i o tome da li su funkcionalnosti implementirane u zadanome vremenu.

Voditelj projekta je obavljao neke zadatke koji se pripisuju XP menadžeru. On je osoba koja je bila član tima, no u timu se nije bavila tehničkim aspektima (implementacija i/ili testiranje). Predstavljao je tim prema vanjskom svijetu i prema menadžmentu organizacije. Nije imao mogućnost slaganja tima, tj. nabavljanja potrebnih resursa jer su to, prije svega, bile odluke linijskog menedžmenta i ovisile prvenstveno o raspoloživim ljudima u organizaciji.

Zbog toga, voditelj projekta je u pravilu koordinirao podjelu zadataka među članovima razvojnog tima.

U *HC Agent* projektu, voditelj projekta je obavljao dio zadataka koji se pripisuju ulozi tragača. Bavljenje procjenama i dobivanje povratnih informacija su bile osnovne zadaće voditelja projekta. Za razliku od XP-a, voditelj projekta se nije bavio rezultatima funkcijskih testova, prijavljivanjem neispravnosti i slučajevima uporabe koji su pokrivali određene neispravnosti. Taj posao su obavljali testeri u suradnji s programerima.

Voditelj projekta je vodio računa o dogovorenim i završenim zadacima u pojedinoj iteraciji te o realizaciji dogovorenih funkcionalnosti koje su ulazile u pojedinu isporuku.

Voditelj projekta je obavljao dio zadataka koji se pripisuju treneru. Osnovna razlika je u tome što se on u pravilu nije bavio programskim kôdom. Tehnička pitanja (neispravnosti) u dizajnu su bile u domeni tehničkog koordinatora (odjeljak 6.3.2.2) a ne voditelja projekta.

6.3.2.2 Tehnički koordinator

Druga osoba tima je bio tehnički koordinator razvoja (engl. *Technical Coordinator*). Ova uloga nije svojstvena XP timu, već je zapravo odlika radne organizaciji gdje se projekt realizirao. Zadaća tehničkog koordinatora u projektu je bila upravljanje i nadgledanje tehničkim razvojem projekta. Ova uloga objedinjuje nekoliko uloga iz XP-a i to su:

- trener (odjeljak 5.6.5)
- programer (odjeljak 5.6.1).

U *HC Agent* razvojnom projektu, tehnički koordinator je obavljao dio zadataka koji se pripisuju treneru. Osnovna razlika je u tome što se on bavio isključivo tehničkim pitanjima u projektu, dakle s problematikom tehničkog razvoja. Sve ostale procesne stvari je rješavao voditelj projekta.

Tehnički koordinator se bavio, uz upravljanje i nadgledanje tehničkim razvojem projekta, i programiranjem prema potrebi, rješavao je zahtjevnije probleme za koje je razvojnim tim imao premalo znanja ili je bio vremenski ograničen kako bi ih riješio u zadanom roku. Kao i kod klasičnog posla programera, tehnički koordinator je bio i autor jediničnih testova za svoj kôd kojima je pokazao da njegov kôd radi i kako radi.

Tehnički koordinator je, prema potrebi, pomagao projektnom timu u implementaciji, provođenje zahtjevnijih zahvata tehnike refaktoriranja te u inspekciji kôda.

6.3.2.3 Programer

Četvero u *HC Agent* razvojnom timu su bili programeri. Pozitivna činjenica je bila sudjelovanje šest osoba iz tima na razvoju. Zbog parnog broja ljudi, oni su se lako podijelili u tri pâra kada se u jednom trenutku željela isprobati tehnika programiranja u pâru.

U *HC Agent* razvojnom projektu koristio se tradicionalni pristup kodiranja (dizajn, kodiranje, pa testiranje), što je bila razlika u odnosu na XP pristup (testiranje, kodiranje, onda refaktoriranje).

Programer je ujedno bio i autor jediničnih testova svog kôda. Jediničnim testovima za svoj kôd pokazao je da njegov kôd radi i kako radi (primjerom pokazuje način korištenja vlastitog kôda) (odjeljak 5.4.1).

Treba naglasiti da su se, unatoč tradicionalnom pristupu kodiranja, koristili jedinični testovi, što se pokazalo kao jako korisna praksa XP razvojne metodologije. Korištenjem jediničnih testova dobila se odmah povratna informacija o funkcionalnost onoga što se testovima provjeravalo.
Prihvaćenost jediničnih testova u *HC Agent* razvojnom projektu je bila izvrsna.

Iako se na projektu koristio tradicionalni pristup kodiranja, često se iskazala potreba za provođenjem tehnike refaktoriranja (odjeljak 5.2.6). Refaktoriranje je obično provodio originalni autor kôda. U slučajevima kada je refaktoriranje bilo složeno i zahtjevno, taj posao se obično obavljao uz pomoć tehničkog koordinatora na projektu.

6.3.2.4 Tester

U *HC Agent* razvojnom projektu, od šest osoba koje su sudjelovale na razvoju, uključujući implementaciju i testiranje, dvije osobe su se bavile testiranjem.

Testeri su se bavili drugim aspektom testiranja, budući da su se programeri bavili jediničnim testiranjem. Testeri su implementirali i redovno pokretali funkcijske testove (testovi prihvaćenosti), obrađivali rezultate testiranja te objavljivali rezultate testiranja na vidljivom mjestu (zajedničkom projektnom repozitoriju).

Zbog specifične strukture *HC Agent Microsoft .NET* programske komponente, bilo je odlučeno da na projektu testeri, umjesto klasičnih testova, rade na razvoju jednostavnih testnih aplikacija s grafičkim sučeljem. Te testne aplikacije su svojim radom omogućile testiranje funkcionalnosti (primarnog scenarija i jednog ili više sekundarnih scenarija) komponenti i središnjeg sustava zdravstva na kojeg su se preko *HC Agent* komponente spajale.
Te aplikacije s grafičkim sučeljem su predstavljale klijentske aplikacije koje su koristile komponentu tako da su one zapravo predstavljale svojevrsnu kolekciju funkcijskih testova za *HC Agent* i ostale komponente.

Budući da je *HC Agent* softverska *Microsoft .NET* komponenta, bilo je odlučeno da jedna testna aplikacija koja koristi komponentu bude razvijena kao *Microsoft .NET* aplikacija (dakle, u *Microsoft .NET* razvojnom okružju). Zbog lakoće korištenja, koristio se *Microsoft Visual Basic .NET* [60] programski jezik i razvojni sustav za razvoj testne aplikacije.

Iz razloga što neki klijenti nisu koristili *Microsoft .NET* platformu za razvoj klijentskih aplikacija, *HC Agent* je mogao biti korišten s COM tehnologijama koje su omogućavale pristup *HC Agent*-u s COM *Collable Wrapper* (CCW) sučeljem. *Microsoft .NET* objekti kod COM klijenata se ne mogu istancirati izravno (COM objekt koristi CCW kao *proxy* sučelje). Zbog toga je bila razvijena i posebna testna aplikacija pomoću *Microsoft Visual Basic 6.0* razvojnog sustava i programskog jezika kako bi se provjerio rad *HC Agent*-a s

CCW sučeljem i klijentima koji koriste COM tehnologiju.

6.3.3 Karakteristika softvera

U radnoj organizaciji u kojoj se provodila realizacija *HC Agent* projekta, razlikuju se dva osnovna područja razvoja projekata:

1. Razvojni projekti koji se baziraju na razvoju složenih programskih i telekomunikacijskih sustava koji su prvenstveno namijenjeni AXE digitalnim centralama [54] (za komutaciju kanala i paketa u složenim telefonskim, mobilnim i podatkovnim mrežama), te CELLO paketskim čvorovima (ATM transportna i kontrolna platforma). Prvenstveno se koristi PLEX [61] (engl. *Programming Language for EXchanges*) konkurentni programski jezik za rad u realnom vremenu.
   Razvojni procesi za ove projekte su dobro utvrđeni i prilično učinkoviti jer su u upotrebi, uz stalna poboljšanja, nekoliko desetljeća. Najčešće, radi se o upotrebi specifičnog PROPS [55] modela vođenja i organizacije projekta upravo namijenjenog za AXE platformu.

2. Razvojni projekti koji su orijentirani razvoju novih produkata korištenjem novijih programskih jezika kao što su C, C++, Java, C# i drugi. Može se reći da je uvođenje i razvoj ovih novijih projekata tek u začecima i da su brojem puno manje zastupljeni nego prethodno navedeni razvojni projekti (prva grupa).
   Razvojni projekti pokušavaju u određenoj mjeri koristiti RUP (engl. *Rational Unified Process*, odjeljak 2.3.7) razvojni proces uz određene modifikacije kako bi bio primjenjiv određenim standardima radne organizacije u kojoj se vrši realizacija projekata.
   Razvojni projekti koji pripadaju ovoj skupini projekata su otvoreni ka novim i učinkovitim razvojnim postupcima. U novije vrijeme, kao posljedica sve veće dinamičnosti projekata i sve većeg skraćivanja vremena za isporuku programskih proizvoda, u obzir dolaze i različite agilne metode koje svojim postupcima mogu ubrzati i poboljšati razvoj.

Po ovoj opisanoj klasifikaciji, *HC Agent* (odjeljak 6.1) razvojni projekt pripada drugoj skupini projekata zbog korištenja novih razvojnih tehnologija u organizaciji i novih područja razvoja.

6.3.4 Postupci XP-a

U nastavku slijede rezultati primijenjenih postupaka XP-a razvojne metodologije na *HC Agent* razvojni projekt.

Dobiveni rezultati se uspoređuju sa postojećim implementiranim postupcima drugih razvojnih projekata, iz druge skupine projekata, koji se vrše u istoj organizaciji. Ti razvojni projekti ne koriste postupke XP razvojne metodologije.

6.3.4.1 Programiranje u pâru

U početnoj primjeni, programeri i testeri su se opirali principu programiranja u pâru, što se može protumačiti kao prirodni otpor prema nečem novom i nepoznatom.

Pokazalo se da je tehnika programiranje u pâru, kada se primjenjivala, bila općenito uspješna i korisna praksa zbog nekoliko razloga.

Uvijek su dvojica programera radila na istom dijelu kôda. Time se značajno povećala kvaliteta programskog kôda nego kada je kôd stvarao pojedinac. Pâr je ipak lakše uočavao greške koje bi pojedincu puno lakše promakle, tj. ostale neprimijećene.

Također, smanjio se broj evidentiranih neispravnosti i grešaka u implementaciji. Pokazalo se da dvojica programera koja rade u pâru za jednim računalom mogu implementirati isto toliko ili čak više funkcionalnosti kao i dvojica koji rade svaki za sebe. Dakle, produktivnost pâra je bila ista ili veća nego produktivnost svakog programera pojedinačno. Razlog tome je bio što su oboje iz pâra bili uključeni u problematiku, odlučivanje, jedinično testiranje, kodiranje i refaktoriranje.

Unatoč ovim dobrim karakteristikama, pokazalo se da programiranje u pâru ima određena negativna obilježja koja su se odražavala na *HC Agent* razvojni projekt.

Duže korištenje ove prakse se pokazalo lošije jer, unatoč većoj produktivnosti na početku korištenja (početnih nekoliko tjedana), programeri su pokazali veću premorenost, odnosno zamor, nego kada su samostalno obavljali radne zadatke.
Ovo je bio subjektivan stav četvero od ukupno šestero programera i testera u razvojnom timu.

Stoga se, unatoč početnim izvrsnim rezultatima bolje radne učinkovitosti, u *HC Agent* razvojnom projektu produktivnost kod nekih parova nije značajno promijenila (povećala) u usporedbi sa samostalnim obavljanjem zadataka, kada se promatrao projekt od početka.

Primijećeno je da kod jednog parâ zajednički rad nije donosio nikakve promjene koje su se odnosile na radni učinak. Smatra se da je tome bio razlog teška prilagodljivost određenih pojedinaca zajedničkom radu te naginjanje samostalnom radu.

Programiranje u pâru je zahtijevalo puno veći angažman i bolju koncentraciju pojedinca nego kada je pojedinac sâm obavljao zadatke.

Budući da je u implementaciji i testiranju sudjelovalo ukupno šest osoba a *HC Agent* razvojni projekt je bio podijeljen u podsustave, svaki pojedinac u razvojnom timu je bio zadužen za određeni podsustav ili više njih. Programiranje u pâru nije bilo pogodno kao stalna praksa za projekt ovako malih ljudskih resursa i podjele projekta na podsustave.
Prvenstveno je tome bio razlog nemogućnost sudjelovanja većeg broja ljudi. Podjela projekta na podsustave nije svojstvena razvojnoj metodologiji ekstremnog programiranja.

Zbog tih specifičnih razloga, programiranje u pâru se koristilo u *HC Agent* projektu redovito u slučajevima kada je trebalo:

- zajednički revidirati kôd
- naći i ispraviti neke teže uočljive neispravnosti
- ubrzati implementaciju i/ili jedinično testiranje

- drastično promijeniti kôd kako bi bio pogodan za implementaciju novih funkcionalnosti
- činiti veće zahvate refaktoriranja koji su bili dosta česti.

Uvidjelo se da je ova praksa neophodna kada se druga osoba iz tima željela ili trebala upoznati sa podsustavom za koji je bio zadužen drugi pojedinac u razvojnom timu, a vremenski okviri nisu dopuštali samostalno proučavanje materije.

Također, ovaj pristup se pokazao učinkovitim u slučaju kada se potpuno nova osoba pridruživala timu kako bi se upoznala s razvojnim projektom ili njegovim određenim podsustavom.

Ovdje je bilo poželjno da ta nova osoba ima približno isti nivo kompetencija kao i druga osoba s kojom čini pâr, kako bi se izbjeglo da jedna osoba radi dok druga pasivno promatra.

U sličnim projektima u organizaciji, princip programiranja u pâru se nije koristio u većoj mjeri. Prema potrebama se koristio zajednički rad kada je to bilo neophodno zbog složenosti radnih zadataka, no ta praksa je bila vrlo rijetka.

Jako dinamičan razvojni projekt kakav je *HC Agent*, s vrlo malim brojem ljudi u timu koji su dio velike radne organizacije, bio je pogodan za povremenu primjenu postupka programiranja u pâru te je pokazao izvrsne rezultate povremenom primjenom ove tehnike.

6.3.4.2 Tehnika refaktoriranja

Iako je refaktoriranje jedna od dvanaest temeljnih postupaka XP metodologije razvoja softvera, *HC Agent* razvojni tim je u određenoj mjeri koristio ovu tehniku i prije samog upoznavanja s XP-om i ovim njegovim postupkom.

Refaktoriranje se provodilo ne samo prije dodavanja neke velike nove funkcionalnosti, već kontinuirano i to:

- Kada se dodavala nova funkcionalnost.
  Često je kôd napisao netko drugi i bilo je potrebno načiniti promjenu funkcionalnosti ili/i dodavati nove funkcionalnosti. Refaktoriranje je znatno pomoglo da se takav kôd bolje razumije.
- Kada se trebala ispraviti evidentirana neispravnost.
  U ispravljanju grešaka (evidentiranih neispravnosti) u kôdu, često je bilo potrebno izvršiti refaktoriranje kako bi se kôd učinio razumljivijim te olakšao proces pronalaženja i ispravljanja neispravnosti, naročito ako je autor kôda bio netko drugi.
- Kada se činila inspekcija (engl. *review*) kôda.
  Neke organizacije, a tako i ova u kojoj se vrši realizacija *HC Agent* projekta, imaju praksu provođenja inspekcije kôda. Inspekcijom kôda se omogućuje širenje znanja unutar razvojnog tima, naročito prijenos znanja sa iskusnijih na manje iskusne članove tima.
  Iako je ova praksa idejno dobra, zbog malog broja članova tima (ograničenih resursa), neodređenim ali prilično kratkim vremenskim rokom isporuke te striktne

podijele posla prema podsustavima, praksa nije u *HC Agent* razvojnom timu bila prisutna u dovoljnoj mjeri. Uočeno je da bi bilo korisno koristiti inspekciju kôda u većoj mjeri, naročito da iskusniji članovi tima pregledavaju kôd manje iskusnih članova tima i predlažu korisna poboljšanja, čime bi se značajno podigla razina znanja u timu.

Programeri su ovu tehniku prihvatili kao svakodnevnu i nužnu jer su uvidjeli važnost veće jednostavnosti i razumljivosti kôda na *HC Agent* projektu.

Pokušaj znatnijeg korištenja tehnike refaktoriranja u projekt je donijelo mnoga poboljšanja u programski kôd. Refaktoriranje je pomoglo da:

- kôd bude tehnički napredniji i bolji, bez da se promijenila funkcionalnost
- budu uočene neispravnosti koje jedinični testovi nisu uspjeli detektirati.

Prilikom provođenja refaktoriranja, prethodno implementirane jedinične testove nije bilo poželjno mijenjati. Bilo je moguće uočiti da li su promjene na programskom kôdu prilikom izvođenja refaktoriranja promijenile implementiranu funkcionalnost tako što su se izvršavali testovi nakon izvršenog refaktoriranja i pratili rezultati testova.

Uklanjanje redundantnog kôda i uklanjanje nekorištene funkcionalnosti je bila praksa koja se koristila i prije pokušaja da se implementira postupak XP refaktoriranja u trenutni razvojni projekt.

Česta praksa u uobičajenim projektima u organizaciji je bila da se refaktoriranje koristilo sporadično i samo onda kada je bila potreba za dodavanjem nove funkcionalnosti koja drastično mijenja postojeći kôd ili kada je kôd trebalo pripremiti da bude sposoban da može primiti novu funkcionalnost.
Tada se obično primjenjivala tehnika refaktoriranja, iako programeri zapravo i nisu znali da se to što rade upravo tako zove.

6.3.4.3 Jedinično testiranje

U *HC Agent* razvojnom projektu, jedna od prvih praksi XP metodologije razvoja softvera koja je uvedena u projekt je bila upravo jedinično testiranje.

Kreiranje jediničnih testova je pomoglo programerima u *HC Agent* razvojnom timu razmotriti što zaista treba biti implementirano, bez implementacije onoga što nije nužno ili što nije bilo određeno kao izričiti zahtjev. Jedinični testovi su pratili zahtjeve sustava.

Korištenjem jediničnih testova u projektu se dobila odmah povratna informacija za vrijeme rada, tj. implementacije.

Korištenjem jediničnih testova u *HC Agent* razvojnom projektu točno se znao trenutak kada je implementacija funkcionalnosti završila time što je pokretanje testova bilo uspješno, tj. jedinični testovi su prošli.

Općenito, jedinično testiranje u *HC Agent* razvojnom projektu je postupak XP razvojne metodologije koji se najviše koristio od svih provedenih postupaka. To je bio postupak koji je najbolje prihvaćen u projektnom timu. Ova praksa je jednostavno

zaživjela i programeri u timu su je od prvog dana prihvatili kao nešto *prirodno* i *neophodno* za rad.

Također, programeri u timu su uvidjeli i dokazali da jediničnim testiranjem provjeravaju vlastiti kôd, ubrzavaju implementaciju, uklanjaju greške i evidentirane neispravnosti te dokazuju da implementirani kôd radi. Razvojni tîm je već nakon kraćeg korištenja ove prakse uvidio da mu se povećao nivo sigurnosti u vlastita programerska znanja ali i sigurnost da vlastiti kôd dobro radi.

Korištenjem jediničnog testiranja na *HC Agent* razvojnom projektu višestruko je smanjio broj neispravnosti u usporedbi sa prijašnjim projektima na kojima se nije provodilo jedinično testiranje. Prema iskustvima sa prethodnih projekata, faza testiranja se u *HC Agent* razvojnom projektu značajno skratila.

Otprilike, pola uloženog vremena potrebnog za implementaciju funkcionalnosti je otpalo na implementaciju jediničnih testova. No, prema mišljenju osoba iz projektnog tima, taj trud se višestruko isplatio.

Stav dvije osobe iz razvojnog tima je da u budućnosti više ne žele kodirati bez korištenja jediničnih testova.

Razvojnom timu u *HC Agent* projektu je ostavljeno na volju da li će ili neće koristiti razvoj diktiran testiranjem. Glavna ideja je zapravo bila kontinuirano koristiti jedinično testiranje.

Kasnije je utvrđeno da razvoj diktiran testiranjem nije zaživio u projektu u dovoljnoj mjeri niti se kontinuirano koristio, iako je ova ideja zapravo jedna od temeljnih praksi XP razvojne metodologije.

Budući da je jedinično testiranje prema XP-u primarna uloga programera, testeri su bili rasterećeni i mogli su se baviti drugim aspektima testiranja. U *HC Agent* razvojnom projektu su samo u početnom dijelu procesa pri implementaciji komponenti podatkovnih tipova (HL7 *Data Types*) testeri bavili proširivanjem početnog skupa jediničnih testova. Kasnije, jedinično testiranje je bilo u potpunoj domeni programera. Testeri su, kao što je ranije rečeno, vršili implementaciju testova prihvaćenosti u obliku posebnih aplikacija s grafičkim sučeljem.

U sličnim projektima koji su u implementaciji u organizaciji u kojima nije korišteno jedinično testiranje, često nije bilo dovoljno jasno kada je programer završio, tj. implementirao svu zahtijevanu funkcionalnost.

U dosadašnjim projektima razvoja softvera u sklopu organizacije, ova praksa korištenja jediničnog testiranja se nije dovoljno koristila jer u najvećem broju slučajeva projekti nisu bili prikladni za to. Programski jezici koji su se koristili jednostavno nisu omogućavali korištenje jediničnog testiranje. U većini slučajeva, radilo se o upotrebi programskog jezika PLEX, specijalnom konkurentnom jeziku za rad u stvarnom vremenu namijenjenom isključivo za razvoj telefonskih sustava još od 1970. godine kada je razvijen. PLEX ne podržava korištenje jediničnog testiranja. U vrijeme nastanka PLEX-a, praksa jediničnog testiranja je bila nepoznanica.

Jedinično testiranje danas podržava općenitije programske jezike te jezike novije generacije. S novijim projektima koji koriste novije tehnologije u organiziciji, odlučeno

je probati ovaj postupak kao i ostale primjenjive postupke XP razvojne metodologije.

U projektima koji nisu koristili jedinično testiranje, rad testera u završnim fazama projekta je bio jako intenzivan i naporan. Obično je traženje neispravnosti bio dugotrajan i mukotrpan proces. Primijećeno je da se dosta vremena trošilo na zajednički rad testera i programera kako bi se ispravile evidentirane neispravnosti.

6.3.4.4 Male i česte isporuke

Razvojni tim *HC Agent* projekta je zaključio da je prilikom planiranja isporuke bilo važno definirati malene jedinice funkcionalnosti koje su bile pogodne za isporuku i koje su mogle biti isporučene u okružje potencijalnih kupaca, tj. razvijatelja klijentskih aplikacija zdravstvene skrbi, u ranim fazama razvoja projekta.

HC Agent tim je uočio da je bilo iznimno važno pravovremeno dobiti vrijednu povratnu informaciju od potencijalnih korisnika (kupaca komponente). Takve informacije su bile važne za pravovremeni i jači utjecaj na danji razvoj sustava. Čest je bio slučaj da je korisnik uočio određene nedorađenosti i propuste koje je ukazao *HC Agent* timu a tim ih je predvidio u razvoju i testiranju.

Pokazalo se da je određeni period isporuke svaka dva tjedna pružao dovoljno vremena za odgovor na određene primjedbe naručitelja i za popravak evidentiranih neispravnosti. Dakle, taj period se pokazao dovoljan kako bi se implementirala i testirala nova funkcionalnost i, za vrijeme kada je tim vršio implementaciju nove funkcionalnosti i popravke za slijedeću isporuku, kupac mogao validirati dobivenu funkcionalnost.

Projektno vodstvo i razvojni tim *HC Agent*-a su pozitivno prihvatili ovaj postupak XP-a. Glavni razlog tome je bio psihološke naravi jer se odmah mogla vidjeti reakcija kupca na isporučeni programski proizvod.

U organizacijskim projektima koji nisu koristili XP metodologiju, u pravilu se nije koristila praksa redovitih isporuka. Razlog tome je bio postojanje velikog broja razvojnih projekata koji su isporučivali programske proizvode za interne potrebe organizacije.

Neki projekti u organizaciji su bili distribuirani. U takvim projektima se povremeno koristila praksa izdavanja malih isporuka. Zbog sve većeg skraćivanja vremena raspoloživog za razvoj, za novije AXE razvojne projektne u zadnje vrijeme se prakticira isporuka određenih podsustava svakog tjedna.

6.4 Osvrt na budući rad iz područja efikasne implementacije XP razvojne metodologije

Područja koja nije dotakao ovaj rad odnose se na opisane postupke XP razvojne metodologije i druge agilne postupke koji nisu analizirani niti dovoljno praćeni na opisanom projektu razvoja javne elektroničke usluge.

Tu valja posebno izdvojiti spomenute postupke kontinuirane integraciji kôda (odjeljak 5.3.5), korištenje standarda (tj. dogovora) pisanja programskog kôda (odjeljak 5.3.2) i

drugih navedenih postupaka XP razvojne metodologije čije korištenje nije analizirano ovim radom (poglavlje 5).

Zbog standarda organizacije u kojoj se vršila implementacija opisanog *HC Agent* razvojnog projekta da se koristi opsežna ali nužna dokumentacija za pripremu, razvoj i isporuku programskog proizvoda, budući rad na području primjene XP razvojne metodologije trebao bi obuhvatiti analizu korištenja potrebne projektne dokumentacije na razvojnim projektima sa postojećim organizacijskim standardima.

Osobitu pažnju treba posveti činjenici da je XP metodologija za razvoj softvera, a ne metodologija za razvoj dokumentacije.

Slijedeća činjenica je postojanje distribuiranih projekata čiji se dijelovi obavljaju u različitim organizacijama u različitim zemljama. Takvi razvojni projekti su znatno zastupljeni u organizaciji u kojoj se vršila implementacija *HC Agent* razvojnog projekta.

Budući rad trebao bi istražiti mogućnost primjene određenih postupaka XP razvojne metodologije na takve distribuirane razvojne projekte.

U navedenoj organizaciji većinom prevladavaju projekti koji koriste PROPS opći model vođenja i organizacije projekta.

Budući rad trebao bi istražiti mogućnosti primjene nekih postupaka agilnih metoda za povećanje učinkovitosti navedenog procesa.

Kao proširenje područja XP razvojne metodologije, agilne metode uključuju tehniku korištenja projektnih retrospektiva (engl. *Project Retrospectives*).

Budući rad na tom području treba istražiti i analizirati korištenje navedene tehnike projektnih retrospektiva kao mogućnost rješenja problematike ponavljanja istih ili sličnih grešaka i propusta u projektnim timovima.

Poglavlje 7

Zaključak

Napredak tehnike u današnjem nemirnom informacijskom dobu neminovno vodi do složenih programskih sustava. U postupcima razvoja softvera, neprestano se postavlja temeljno pitanje da li će softver biti isporučen na vrijeme, sa što manje otkrivenih pogrešaka i nedostataka te sa što zadovoljnijim naručiteljima i kupcima.

Povijesni razvoj metodologija razvoja softvera uvijek je težio za novijim i efikasnijim metodologijama koje mogu odgovoriti na sveprisutne probleme u razvojnim postupcima.

Sposobnost da se skrati razvojni ciklus proizvodnje softvera i mogućnost što bržeg odgovora na promjene sve je više imperativ u današnjoj poslovnoj zajednici.

Kao rezultat tih nastojanja, javljaju se agilne metode razvoja softvera koje su manje opsežne i u svojim temeljnim stavovima prihvaćaju činjenicu da je softver teško kontrolirati. To su lakše, manje procesno-orijentirane metodologije razvoja i usredotočuju se na proizvod, a ne na sâm proces.
Fokusiraju se na male jedinice posla i time smanjuju rizike. Glavni aspekt agilnih metoda je jednostavnost i brzina. Principi i vrijednosti agilnog pokreta su: individualnost i interakcije iznad procesâ i alatâ, softver koji radi iznad opsežne dokumentacije, suradnja s kupcem iznad ugovornih obaveza, brzi odgovor na promjene iznad slijeđenja plana.

Ekstremno programiranje, XP, je pristup, odnosno metoda razvoja softvera koja je jedan od predstavnika agilnih metoda. Nastala je kao potreba da se posao razvoja završi na vrijeme, s postupcima koji su smatrani dovoljno djelotvornima da to omoguće.

Prakse XP nisu same po sebi nove. XP ih sakuplja kako bi zajedno funkcionirale i činile novu metodologiju razvoja softvera. Termin ’ekstremno’ odnosi se na te zajedničke prakse za koje se dokazalo da su dobro primjenjive i koje su dovedene do svojih krajnjih granica.

Ideja magistarskog rada je bila prikazati i istražiti osnovne postupke XP razvojne metodologije koje se mogu primijeniti na projekt razvoja javne elektroničke usluge.

Magistarski rad je imao za cilj pokazati glavne postupke, odnosno temeljne karakteristike XP metodologije razvoja softvera. Detaljno su opisani osnovni postupci XP razvojne metodologije, s osobitim naglaskom na programski proizvod.

Istražene su mogućnosti primjene određenih postupaka XP-a u realizaciji razvojnog projekta javne elektroničke usluge. Projektni tîm koji je sudjelovao u implementaciji se sastojao od četiri programera, dva testera, jednog tehničkog koordinatora razvojnih aktivnosti i od voditelja projekta. Okolnosti u kojima se provodila implementacija programskog proizvoda su bile: mali projektni tim od svega osam osoba, kratko vrijeme predviđeno za implementaciju, testiranje i isporuku programskog proizvoda te rad sa novim razvojnim tehnologijama i programskim pomagalima za razvoj i testiranje.

Posebna je pažnja posvećena postupcima koji su našli dobru primjenu i prihvaćenost u razvojnom timu: programiranje u pâru, tehnika refaktoriranja programskog kôda, jedinično testiranje, korištenje malih i čestih isporuka. Analizirane su prihvaćenosti tih postupaka od projektnog tima koji je sudjelovao u razvoju programskog proizvoda (testeri, programeri i menadžeri),

Programiranje u pâru je općenito uspješan, iako ne i najbolje prihvaćen postupak XP razvojne metodologije. Razlog tome je bila teža prilagodljivost nekih pojedinaca zajedničkom radu. Duže korištenje prakse je uzrokovalo veći zamor i smanjenje radnog učinka u razvojnom timu. Primjenom postupka se smanjio broj evidentiranih neispravnosti u implementaciji te ubrzala implementacija. Postupak se redovito primjenjivao u slučajevima kada je bilo potrebno: zajednički revidirati kôd, naći i ispraviti neke teže uočljive neispravnosti, ubrzati implementaciju i/ili jedinično testiranje, drastično promijeniti kôd kako bi bio pogodan za implementaciju novih funkcionalnosti, činiti veće zahvate refaktoriranja koji su bili dosta česti.

Tehnika refaktoriranja programskog kôda se provodila kada se dodavala nova funkcionalnost, kada se činila inspekcija kôda i kada se trebala ispraviti evidentirana neispravnost. Programeri su ovu tehniku prihvatili kao svakodnevnu i nužnu jer su uvidjeli važnost veće jednostavnosti i razumljivosti kôda.

Jedinično testiranje je postupak XP razvojne metodologije koji je bio najbolje prihvaćen u razvojnom timu. Korištenjem jediničnog testiranja programeri su jasnije uočili ono što treba biti implementirano, dobili su odmah povratnu informaciju za vrijeme implementacije o radu vlastitog programskog kôda, ubrzala se implementacija, višestruko se umanjio broj neispravnosti u kasnijem provođenju testiranja. Razvojni tîm je već nakon kraćeg korištenja ove prakse uvidio da mu se povećao nivo sigurnosti u vlastita programerska znanja ali i sigurnost da vlastiti kôd dobro radi. Jedinični testovi su poslužili kao određena dokumentacija i način ispravnog korištenja programskog kôda. Ova praksa je jednostavno zaživjela i programeri u timu su je od prvog dana prihvatili kao nešto prirodno i neophodno za rad. Stav nekih pojedinaca u timu je bio da više ne žele kodirati bez korištenja jediničnih testova.

Korištenje prakse malih i čestih isporuka projektni tim je pravovremeno dobivao povratne informacije od kupca, tj. naručitelja programskog proizvoda. Prema reakciji korisnika, tim je znao da li je na dobrom putu u implementaciji, što još treba implementirati i promijeniti.

Istražene su mogućnosti uvođenja postupaka XP-a u fazi implementacije funkcionalnosti razvojnog projekta javne elektroničke usluge. Faza implementacije u kojoj su provedeni i analizirani postupci XP razvojne metodologije je bila u trajanju oko pola godine. Početna faza istraživanja i prikupljanja zahtjeva nije uključena u promatranje zbog

specifičnosti projekta javne elektroničke usluge.

Opisana su postojeća programska pomagala za testiranje: *csUnit* i *JUnit* testna okružja te *Rational TestManager* [58]. Posebna pažnja je posvećena integraciji programskih pomagala za testiranje, *JUnit* testnog okružja i *Rational TestManager*-a. Opisani su glavni nedostaci postojećih programskih pomagala za testiranje. Zbog nemogućnosti asinkronog načina rada postojećih alata za jedinično testiranje i zahtjeva za asinkronim načinom rada programskog proizvoda, nastala je potreba za razvojem novog testnog okružja.

Dodatak A

Izvorni kôd

Ovo poglavlje sadrži dodatke magistarskom radu:

A.1 BN.java

```
package com.ericsson.hl7.types.basic;
import java.util.List;
import java.util.Collections;
import java.util.Arrays;

public final class BN extends BooleanAbstract implements Comparable {
    public final static BN TRUE = new BN(true);
    public final static BN FALSE = new BN(false);

    // collection that contains all typesafe enums (necessary for searching)
    private static final BN[] PRIVATE_VALUES = {TRUE, FALSE};
    public static final List VALUES = Collections.unmodifiableList(
            Arrays.asList(PRIVATE_VALUES));

    private BN(boolean b) {
        super(b);
    }

    public String getLongName() {
        return "BooleanNonNull";
    }

    public BN isNull() {
        return BN.FALSE;
    }

    public static BN valueOf(boolean b) {
        return (b ? BN.TRUE : BN.FALSE);
    }

    /**
     * Method that uses ordinality to compare two BNs
     * @param o - BN that is compared to <this>
     * @return  0 - BNs are identical
     *         >0 - <this> is larger then o
```

```
         *         <0 - o is larger then <this>
         */
        public int compareTo(Object o) {
            return ordinal - ((BN) o).ordinal;
40      }

        /**
         * Method that is used to find whether an object is of type BN or not
         * @param o object that is examined
         * @return true  - object is one of typesafe enum classes
         *         false - object is not of type BN and is not one of typesafe
         *                 enum classes
         */
        public boolean contains(Object o) {
50          if (!(o instanceof BN))
                return false;
            return VALUES.contains(o);
        }

        /**
         * Conjunction (AND) is associative and commutative, with true as a
         * neutral element. False AND any Boolean value is false.
         * @param x left hand operand
         * @return BN
60       */
        public BN and(BN x) {
            return BN.valueOf(value() && x.value());
        }

        /**
         * Negation of a Boolean turns true into false and false into
         * true.
         * @return BN
         */
70      public BN not() {
            return BN.valueOf(!value());
        }

        /**
         * The disjunction x OR y is false if and only if x is false and y is
         * false.
         * @param x BN
         * @return BN
         */
80      public BN or(BN x) {
            return BN.valueOf(value() || x.value());
        }

        /**
         * The exclusive-OR constrains OR such that the two operands may not both
         * be true.
         * @param x BN
         * @return BN
         */
90      public BN xor(BN x) {
            return (this.or(x)).and((this.and(x)).not());
```

```
        }

        /**
         * A rule of the form IF condition THEN conclusion. Logically the
         * implication is defined as the disjunction of the negated condition and
         * the conclusion, meaning that when the condition is true the conclusion
         * must be true to make the overall statement true. The logical
         * implication is important to make invariant statements.
100      *
         * invariant(BN condition, conclusion) {
         *   condition.implies(conclusion).equal(condition.not.or(conclusion));
         * };
         *
         * The implication is not reversible and does not specify what is true
         * when the condition is false (ex falso quodlibet lat. from false follows
         * anything).
         *
         * A B A implies B
110      * T T     T
         * T F     F
         * F T     T
         * F F     T
         *
         * @param x BN
         * @return BN
         */
        public BN implies(BN x) {
            BN negation = this.not();
120         return negation.or(x);
        }
    }
```

A.2 TC_BN.java

```
 1  package com.ericsson.hl7.types.basic;
    import junit.framework.*;
    import com.ericsson.hl7.types.basic.*;

    public class TC_BN extends TestCase {
        private String className="TC_BN";
        private BN bn1, bn2, bn3;

        /**
10       * Description: Test Method for BN Object Creation.
         */
        public void testCreateObject() {
            try {
                bn1 = null;
                Assert.assertNull (className, bn1);
            }
            catch (Exception e) {
                fail (className);
            }
20
```

```
            try {
                bn1 = null;
                bn1 = BN.TRUE;
                Assert.assertNotNull (className, bn1);
            }
            catch (Exception e) {
                fail (className);
            }

            try {
                bn1 = null;
                bn1 = BN.FALSE;
                Assert.assertNotNull (className, bn1);
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn2 = null;
                bn2 = BN.valueOf(true);
                Assert.assertNotNull (className, bn2);
            }
            catch (Exception e) {
                fail(className);
            }
        }

        /**
         * Description: Test Method for testing 'public BL and(BL x)' Method.
         */
        public void testAND() {
            try {
                bn1 = BN.TRUE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.and(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.TRUE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.and(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.FALSE, bn1.and(bn2));
            }
            catch (Exception e) {
```

```
                fail(className);
            }

 80         try {
                bn1 = BN.FALSE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.and(bn2));
            }
            catch (Exception e) {
                fail(className);
            }
        }

 90     /**
         * Description: Test Method for testing 'public BL not()' Method.
         */
        public void testNOT() {
            try {
                bn1 = BN.TRUE;
                Assert.assertEquals (className, BN.FALSE, bn1.not());
            }
            catch (Exception e) {
                fail(className);
100         }

            try {
                bn1 = BN.FALSE;
                Assert.assertEquals (className, BN.TRUE, bn1.not());
            }
            catch (Exception e) {
                fail (className);
            }
        }
110
        /**
         * Description: Test Method for testing 'public BL or(BL x)' Method.
         */
        public void testOR() {
            try {
                bn1 = BN.TRUE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.or(bn2));
            }
120         catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.TRUE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.TRUE, bn1.or(bn2));
            }
            catch (Exception e) {
130             fail(className);
            }
```

```
            try {
                bn1 = BN.FALSE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.or(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.or(bn2));
            }
            catch (Exception e) {
                fail(className);
            }
        }

        /**
         * Description: Test Method for testing 'public BL xor(BL x)' Method.
         */
        public void testXOR() {
            try {
                bn1 = BN.TRUE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.FALSE, bn1.xor(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.TRUE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.TRUE, bn1.xor(bn2));
            }
            catch (Exception e) {
                fail (className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.xor(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.xor(bn2));
            }
            catch (Exception e) {
```

```
                fail (className);
190         }
        }

        /**
         * Description: Test Method for testing 'public BL implies(BL x)' Method.
         */
        public void testIMPLIES() {
            try {
                bn1 = BN.TRUE;
                bn2 = BN.TRUE;
200             Assert.assertEquals (className, BN.TRUE, bn1.implies(bn2));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.TRUE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.implies(bn2));
210         }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.implies(bn2));
            }
220         catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                bn2 = BN.FALSE;
                Assert.assertEquals (className, BN.TRUE, bn1.implies(bn2));
            }
            catch (Exception e) {
230             fail(className);
            }
        }

        /**
         * Description: Test Method
         */
        public void testDATA_TYPE() {
            TYPE type1;
            CS name1;
240
            // Short Name
            try {
                bn1 = BN.FALSE;
                type1 = bn1.dataType();
```

```
            name1 = type1.shortName();
            Assert.assertEquals(className, "BN", name1.value());
        }
        catch (Exception e) {
            fail (className);
        }

        // Long Name
        try {
            bn1 = BN.FALSE;
            type1 = bn1.dataType();
            name1 = type1.longName();
            Assert.assertEquals(className, "BooleanNonNull", name1.value());
        }
        catch (Exception e) {
            fail (className);
        }

        // Implies
        try {
            BIN bin = new BIN("");
            BN bn = BN.FALSE;
            BL bl = BL.FALSE;
            CD cd = new CD(new ST(""), new UID(""));
            CR cr = new CR(cd, new CV(new ST(""), new UID(""), new ST(""),
                               new ST(""), new ST(""), new ED("")));
            CS cs = new CS("");
            CV cv = new CV(NullFlavor.NASK);
            ED ed = new ED("");
            GTS gts = new GTS();
            II ii = new II(new UID(""));
            INT int1 = new INT(0);
            LIST_BN lbn = new LIST_BN();
            LIST_CR lcr = new LIST_CR(new CS("NA"));
            OID oid = new OID ("1.2.3.4");
            PQ pq = new PQ ();
            QTY qty = new QTY ();
            REAL real = new REAL();
            ST st = new ST("Test only");
            TEL tel = new TEL();
            TS ts = new TS();
            UID uid = new UID ("");
            URL url = new URL ();

            TYPE t_bin = bin.dataType();
            TYPE t_bn = bn.dataType();
            TYPE t_bl = bl.dataType();
            TYPE t_cd = cd.dataType();
            TYPE t_cr = cr.dataType();
            TYPE t_cs = cs.dataType();
            TYPE t_cv = cv.dataType();
            TYPE t_ed = ed.dataType();
            TYPE t_gts = gts.dataType();
            TYPE t_ii = ii.dataType();
            TYPE t_int = int1.dataType();
            TYPE t_lbn = lbn.dataType();
```

```
            TYPE t_lcr = lcr.dataType();
            TYPE t_oid = oid.dataType();
            TYPE t_pq = pq.dataType();
            TYPE t_qty = qty.dataType();
            TYPE t_real = real.dataType();
            TYPE t_st = st.dataType();
            TYPE t_tel = tel.dataType();
            TYPE t_ts = ts.dataType();
            TYPE t_uid = uid.dataType();
310         TYPE t_url = url.dataType();

            Assert.assertEquals(className, false,
                (t_bn.implies(t_bin)).value());
            Assert.assertEquals(className, true,
                (t_bn.implies(t_bn)).value());
            Assert.assertEquals(className, true,
                (t_bn.implies(t_bl)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_cd)).value());
320         Assert.assertEquals(className, false,
                (t_bn.implies(t_cr)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_cs)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_cv)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_ed)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_gts)).value());
330         Assert.assertEquals(className, false,
                (t_bn.implies(t_ii)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_int)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_lbn)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_lcr)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_oid)).value());
340         Assert.assertEquals(className, false,
                (t_bn.implies(t_pq)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_qty)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_real)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_st)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_tel)).value());
350         Assert.assertEquals(className, false,
                (t_bn.implies(t_ts)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_uid)).value());
            Assert.assertEquals(className, false,
                (t_bn.implies(t_url)).value());
        }
```

```
            catch (Exception e) {
                fail(className);
            }
360     }

        /**
         * Description: Test Method for testing 'public BN nonNull()' Method.
         */
        public void testNON_NULL() {
            try {
                bn1 = null;
                bn1 = BN.TRUE;
                Assert.assertEquals (className, BN.TRUE, bn1.nonNull());
370             assertEquals(className, BL.TRUE,
                            bn1.isNull().equal(bn1.nonNull().not()));
            }
            catch (Exception e) {
                fail (className);
            }

            try {
                bn1 = null;
                bn1 = BN.FALSE;
380             Assert.assertEquals (className, BN.TRUE, bn1.nonNull());
                assertEquals(className, BL.TRUE,
                            bn1.isNull().equal(bn1.nonNull().not()));
            }
            catch (Exception e) {
                fail (className);
            }
        }

        /**
390      * Description: Test Method
         */
        public void testNULL_FLAVOR() {
            try {
                bn1 = BN.TRUE;
                assertEquals(className, null, bn1.nullFlavor());
            }
            catch (Exception e) {
                fail (className);
            }
400
            try {
                bn1 = BN.FALSE;
                assertEquals(className, null, bn1.nullFlavor());
            }
            catch (Exception e) {
                fail(className);
            }
        }

410     /**
         * Description: Test Method for testing 'public BN isNull()' Method.
         */
```

```
        public void testIS_NULL() {
            try {
                bn1 = BN.TRUE;
                Assert.assertEquals (className, BN.FALSE, bn1.isNull());
            }
            catch (Exception e) {
                fail (className);
            }

            try {
                bn1 = BN.FALSE;
                Assert.assertEquals (className, BN.FALSE, bn1.isNull());
            }
            catch (Exception e) {
                fail(className);
            }
        }

        /**
         * Description: Test Method
         */
        public void testNOT_APPLICABLE() {
            try {
                bn1 = BN.TRUE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch(Exception e) {
                fail(className);
            }
        }

        /**
         * Description: Test Method
         */
        public void testUNKNOWN() {
            try {
                bn1 = BN.TRUE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch(Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.FALSE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch (Exception e) {
```

```
                fail(className);
            }
        }

        /**
         * Description: Test Method
         */
        public void testOTHER() {
            try {
                bn1 = BN.TRUE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch (Exception e) {
                fail (className);
            }

            try {
                bn1 = BN.FALSE;
                assertEquals(className, BL.FALSE, bn1.notApplicable());
            }
            catch (Exception e) {
                fail(className);
            }
        }

        /**
         * Description: Test Method for testing 'public BL equal(ANY x)' Method.
         */
        public void testEQUAL() {
            try {
                bn1 = BN.TRUE;
                bn2 = BN.TRUE;
                bn3 = BN.TRUE;

                Assert.assertEquals (className, BL.TRUE, bn1.equal(bn2));

                assertEquals(className, BN.TRUE, bn1.nonNull());
                assertEquals(className, BN.TRUE, bn2.nonNull());
                assertEquals(className, BN.TRUE, bn3.nonNull());

                assertEquals(className, BL.TRUE, bn1.equal(bn1));
                assertEquals(className, BL.TRUE,
                      bn1.equal(bn2).equal(bn2.equal(bn1)));
                assertEquals(className, BL.TRUE,
                      bn1.equal(bn2).and(bn2.equal(bn3)).implies(bn1.equal(bn3)));
                assertEquals(className, BL.TRUE,
                      bn1.equal(bn2).implies(bn1.dataType().longName().equal(
                      bn2.dataType().longName())));
            }
            catch (Exception e) {
                fail(className);
            }

            try {
                bn1 = BN.TRUE;
                bn2 = BN.FALSE;
```

```
            Assert.assertEquals (className, BL.FALSE, bn1.equal(bn2));
        }
        catch (Exception e) {
            fail(className);
        }

        try {
            bn1 = BN.FALSE;
            bn2 = BN.TRUE;
            Assert.assertEquals (className, BL.FALSE, bn1.equal(bn2));
        }
        catch(Exception e) {
            fail(className);
        }

        try {
            bn1 = BN.FALSE;
            bn2 = BN.FALSE;
            bn3 = BN.FALSE;
            Assert.assertEquals (className, BL.TRUE, bn1.equal(bn2));

            assertEquals(className, BN.TRUE, bn1.nonNull());
            assertEquals(className, BN.TRUE, bn2.nonNull());
            assertEquals(className, BN.TRUE, bn3.nonNull());

            assertEquals(className, BL.TRUE, bn1.equal(bn1));
            assertEquals(className, BL.TRUE,
                  bn1.equal(bn2).equal(bn2.equal(bn1)));
            assertEquals(className, BL.TRUE,
                  bn1.equal(bn2).and(bn2.equal(bn3)).implies(bn1.equal(bn3)));
            assertEquals(className, BL.TRUE,
                  bn1.equal(bn2).implies(bn1.dataType().longName().equal(
                  bn2.dataType().longName())));
        }
        catch (Exception e) {
            fail(className);
        }
    }

    /**
     * Description: Suite Method that uses reflection to dynamically create a
     * test suite containing all the testXXX() methods.
     * @return TestSuite
     */
    public static Test suite() {
         return new TestSuite(TC_BN.class);
    }

    /**
     * Description: Main Method for running the test with the textual test
     * runner.
     * @param args String[]
     */
    public static void main (String[] args) {
         junit.textui.TestRunner.run(suite());
    }
```

```
}
```

A.3 TM_BN.java

```
package test.com.ericsson.hl7.types.basic;
import com.rational.test.tss.TestScript;
import com.rational.test.tss.TSSLog;
import com.ericsson.hl7.types.basic.TC_BN;
import junit.framework.*;
import java.util.Enumeration;

public class TM_BN extends TestScript {
    public void testMain (String[] args) {
        try {
            tms.startTestServices();

            TestSuite suite = new TestSuite(TC_BN.class);

            // Runs the tests and collects their result in a '(RestResult)
            // result'
            TestResult result = new TestResult();
            suite.run(result);

            TSSLog log = new TSSLog();

            // returns the tests as an enumeration
            Enumeration tests = suite.tests();

            // Returns an Enumeration for the failures
            Enumeration test_failures = result.failures();

            // successfull
            if (result.wasSuccessful()) {
                while (tests.hasMoreElements()) {
                    TestCase testCase = (TestCase) tests.nextElement();
                    log.testCaseResult(testCase.getName(),
                        log.TSS_LOG_RESULT_PASS);
                }
            }

            // unsuccessfull
            else {
                while (test_failures.hasMoreElements()) {
                    TestFailure failure = (TestFailure)
                                          test_failures.nextElement();
                    TestCase failedTestCase = (TestCase)
                                              failure.failedTest();
                    log.testCaseResult(failedTestCase.getName(),
                        log.TSS_LOG_RESULT_FAIL,
                        failure.exceptionMessage(),
                        null);
                }
            }
        }
```

```
            catch (Exception e) {
                System.out.println ("Exception: " + e.toString()
                                                  + " in TMWrapper!");
                e.printStackTrace();
            }

            finally {
                tms.endTestServices();
            }
        }
    }
```

A.4 Telecom_Factory_TC.cs

```
using provide = com.ericsson.healthcare.cen.xml.ehcr;
using apiClass = com.ericsson.healthcare.cen;
using factoriesClass = com.ericsson.healthcare.cen.factories;
using csUnit;

namespace com.ericsson.healthcare.cen.factories {
   /// <summary>
   /// Summary description for Telecom_Factory_TC.
   /// </summary>
   [TestFixture]
   public class Telecom_Factory_TC {
      apiClass.Telecom apiTelecom = null;
      apiClass.Telecom apiTelecom_1 = null;
      provide.Telecom cenTelecom = null;

      [SetUp]
      public void setUp() {
         apiTelecom = new apiClass.Telecom();
         apiTelecom_1 = new cen.Telecom();
         cenTelecom = new provide.Telecom();
      }

      [TearDown]
      public void tearDown() {
         apiTelecom = null;
         apiTelecom_1 = null;
         cenTelecom = null;
      }

      [Test]
      public void test_Factory_Telecom() {
         fillTelecom(apiTelecom);

         cenTelecom =
            factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
         apiTelecom_1 =
            factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

         Assert.True(apiTelecom == apiTelecom_1);
```

```
        }

        [Test]
        public void test_Factory_Telecom_Former() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Former;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
        }

        [Test]
        public void test_Factory_Telecom_Home() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Home;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
        }

        [Test]
        public void test_Factory_Telecom_Other() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Other;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
        }

        [Test]
        public void test_Factory_Telecom_Temporary() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Temporary;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
```

```
        }

        [Test]
        public void test_Factory_Telecom_Unspecified() {
100        fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Unspecified;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
110        }

        [Test]
        public void test_Factory_Telecom_Vacation() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Vacation;
           apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
120        apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
        }

        [Test]
        public void test_Factory_Telecom_Work() {
           fillTelecom(apiTelecom);
           apiTelecom.AddrType = apiClass.AddrType_Type.Work;
130        apiTelecom.AddrTypeSpecified = true;

           cenTelecom =
              factoriesClass.Telecom_Factory.ConvertTelecom(apiTelecom);
           apiTelecom_1 =
              factoriesClass.Telecom_Factory.ConvertTelecom(cenTelecom);

           Assert.True(apiTelecom == apiTelecom_1);
        }

140     internal static void fillTelecom(apiClass.Telecom apiTelecom_) {
           if (null != apiTelecom_) {
              factoriesClass.RandomString rand =
                 new factoriesClass.RandomString();

              apiClass.StructTelecomNum structTelecomNum =
                 new apiClass.StructTelecomNum();
              factoriesClass.StructTelecomNum_Factory_TC.fillStructTelecomNum(
                 structTelecomNum);

150           apiClass.ValidPeriod validPeriod = new apiClass.ValidPeriod();
              factoriesClass.ValidPeriod_Factory_TC.fillValidPeriod(validPeriod);
```

```
                apiTelecom_.StructTelecomNum = structTelecomNum;
                apiTelecom_.TelecomType = apiClass.TelecomTelecomType.Email;
                apiTelecom_.UnstructTelecomNum = rand.GetString();
                apiTelecom_.ValidPeriod = validPeriod;
            }
        }
    }
}
```

Literatura

[1] Gary Chin: *Agile Project Management: How to Succeed in the Face of Changing Project Requirements*,
Amacom, ISBN:0814471765, 2004.

[2] Philip G. Armour: *The Laws of Software Process: A New Model for the Production and Management of Software*,
Auerbach Publications, ISBN:0849314895, 2004.

[3] Pete McBreen: *Questioning Extreme Programming*,
Addison Wesley Pub Co, ISBN:0201844575, srpanj 2002.

[4] Alstair Cockburn: *Agile Software Development*,
Addison Wesley Professional, 2001.

[5] Jason Chavert: *Project Management Methodologies: Selecting, Implementing, and Supporting Methodologies and Processes for Projects*,
John Wiley & Sons, ISBN:0471221783, 2003.

[6] Martin Fowler: *The New Methodology*
(`http://www.martinfowler.com/articles/newMethodology.html`)

[7] Dean Leffingwell, Don Widrig: *Managing Software Requirements*,
Addison Wesley, ISBN:0201615932, 1999.

[8] Extreme Programming: A Gentle Introduction
(`http://www.extremeprogramming.org`)

[9] Top-down and bottom-up design (*Wikipedia*)
(`http://en.wikipedia.org/wiki/Top-Down_Model`)

[10] Top-down and bottom-up design (*Wikipedia*)
(`http://en.wikipedia.org/wiki/Bottom_Up`)

[11] Winston W. Royce: *Managing Development of Large Scale Software: Concepts and Techniques Proceeding*,
Wescon, 1970.

[12] Joseph M. Firestone, Ph.D: *Knowledge Management Process Methodology: An Overview*,

Volume One, No. Two, January 15, 2001 (c) Knowledge Management Consortium International, Inc.

[13] Barry Boehm: *A Spiral Model of Software Development and Enhancement*
ACM SIGSOFT Software Engineering Notes, kolovoz 1986,
IEEE Computer, vol.21, #5, svibanj 1988., str. 61-72.

[14] Christian Cappelan, Justin Chu, Katur Patel, Zinger Yang: *Spiral Software Development*,
Computer Science 180 - Software Engineering, Fall 2004, Professor Judith Stafford, Tufts University, Medford, Massachusetts

[15] L.B.S. Racconn: *The Chaos Model and the Chaos Life Cycle*,
ACM Software Engineering Notes, ACM Press, Volume 20, Number 1, str. 55-66, siječanj 1995.

[16] Center for Technology of Government: *A Survay of System Development Process Models*,
University of Albany/SUNY, 1998.

[17] Craig Larman: *Agile and Iterative Development: A Manager's Guide*,
Addison Wesley, ISBN:0131111558, 2003.

[18] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen & Juhani Warsta: *Agile software development methods*
(`http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf`)

[19] Gary Pollice, Liz Augustine, Chris Lowe, Jas Madhur: *Software Development for Small Teams: A RUP-Centric Approach*,
Addison-Wesley Professional, ISBN:0321199502, 2004.

[20] Manifesto for Agile Software Development, 22. ožujka 2002.
(`http://www.agilemanifesto.org`)

[21] Robert K. Wysocki, Rudd McGary: *Effective Project Management: Traditional, Adaptive, Extreme, Third Edition*,
Wiley Publishing, Inc., srpanj 2003.

[22] Miller, G. G.: *The Characteristics of Agile Software Processes. The 39th International Conference of Object-Oriented Languages and Systems (TOOLS 39)*,
Santa Barbara, CA, 2001.

[23] Steve McConnel: *Rapid Development*,
Microsoft Press, A Division of Microsoft Corporation, Redmond, 1996.

[24] Michele Marchesi, Giancarlo Succi, Don Wells, Laurie Williams: *Extreme Programming Perspectives*,
Addison-Wesley Pub Co, ISBN:0201770059, 1. izdanje, kolovoz 2002.

[25] Kent Beck: *Extreme Programming Explained: Embrance Change*,
Addison Wesley Professional, ISBN:0201616416, listopad 1999.

[26] William C. Wake: *Extreme Programming Explored*,
Addison Wesley Professional, srpanj 2001.

[27] Amr Elssamadisy: *XP On A Large Project - A Developer's View*
ThoughtWorks, 2004.

[28] Lisa Crispin, Tip House: *Testing Extreme Programming*,
Addison-Wesley Pub Co, ISBN:0321113551, 1. izadanje, listopad 2002.

[29] Kent Beck, Martin Fowler: *Planning Extreme Programming*,
Addison Wesley Professional, ISBN:0201710919, 1. izdanje, listopad 2000.

[30] Ron Jeffries, Ann Anderson, Chet Hendrickson: *Extreme Programming Installed*,
Addison Wesley Professional, ISBN:0201708426, listopad 2000.

[31] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts: *Refactoring: Improving the Design of Existing Code*,
Addison Wesley Professional, ISBN:0201485672, lipanj 1999.

[32] William C. Wake: *Refactoring Workbook*,
Addison-Wesley Pub Co, ISBN:0321109295, 1. izdanje, kolovoz 2003.

[33] Mary Poppendieck, Tom Poppendieck: *Lean Software Development: An Agile Toolkit*,
Addison Wesley, ISBN:0321150783, svibanj 2003.

[34] Kent Beck: *Test Driven Development: By Example*,
Addison Wesley Professional, ISBN:0321146530, 1. izdanje, studeni 2002.

[35] JUNIT, TESTING RESOURCES FOR EXTREME PROGRAMMING
(`http://www.junit.org`)

[36] Java 2 Platform, Standard Edition (J2SE)
(`http://java.sun.com/j2se`)

[37] Java 2 Platform, Enterprise Edition (J2EE)
(`http://java.sun.com/j2ee`)

[38] csUnit, Testing Resources for Extreme Programming
(`http://www.csunit.org`)

[39] Microsoft .NET
(`http://www.microsoft.com/net`)

[40] Laurie Williams, Robert R. Kessler, Ward Cunningham, Ron Jeffries: *Strengthening the Case for Pair Programming*,
July/August 2000, IEEE Software, pages 19-25

[41] Laurie Williams, Robert Kessler: *Pair Programming Illuminated*,
Addison Wesley, ISBN:0201745763, lipanj 2002.

[42] Microsoft Visual SourceSafe
(http://msdn.microsoft.com/ssafe)

[43] Rational ClearCase
(http://www-306.ibm.com/software/awdtools/clearcase)

[44] Concurrent Versions System
(https://www.cvshome.org)

[45] Andrew Hunt, David Thomas: *Pragmatic Unit Testing*,
The Pragmatic Programmers, rujan 2003.

[46] Health Care Agent - HC Agent
(http://www.ericsson.com/hr/produkti/e-zdravstvo/hc_agent.shtml)

[47] HL7 version 3, ballot 6
(http://www.hl7.org/v3ballot6/html/foundationdocuments/welcome/index.html)

[48] CEN/TC 251 - European Standardization of Health Informatics
(http://www.centc251.org)

[49] XML, Extensible Markup Language
(http://www.w3c.org/xml/)

[50] HL7 - Health Level 7
(http://www.hl7.org)

[51] Web Services Activity
(http://www.w3.org/2002/ws/)

[52] Microsoft Visual J# .NET Developer Center
(Bringing the Java language to the .NET Framework)
(http://msdn.microsoft.com/vjsharp)

[53] Microsoft Visual C# .NET Developer Center
(An innovative language and tool for building .NET connected solutions)
(http://msdn.microsoft.com/vcsharp)

[54] AXE
(http://www.ericsson.com/technology/tech_articles/AXE.shtml)

[55] PROPS Online - Introduction to PROPS
(http://www.semcon.se/spm/eng/model/props_intro_en/intro.props.asp)

[56] COM: Component Object Model Technologies
(http://www.microsoft.com/com)

[57] IntelliJ IDEA, multi-platform Java IDE
(http://www.jetbrains.com)

[58] IBM Rational Software
(http://www.rational.com)

[59] Miljenko Kalenik, Krešimir Maržić, Karlo Šmid: *Testing Methodology for Primary Health Care Enterprise System*,
Mipro konferencija, Opatija, R. Hrvatska, svibanj 2004.

[60] Microsoft Visual Basic Developer Center
(The most productive Tool for building .NET-connected applications)
(http://msdn.microsoft.com/vbasic)

[61] A formal Semantics for PLEX
(http://www.mrtc.mdh.se/publications/0736.pdf)

Indeks

Acceptance Test, 25, 30, 33, 38, 42, 44–46, 48, 51
Adaptive Software Development, 15, 18
Agile Methods, 6, 9, 11, 13, 14, 18–20
agilne metode, 6, 9, 11, 13, 14, 18–20
ASD, 15, 18

Bottom Up model, 6, 9
brzina projekta, 31

Chaos model, 9
Coach, 50, 52
Consultant, 50, 53
Crystal, 15, 18
Customer, 11, 13, 16, 17, 20–22, 25, 26, 28–30, 37, 39, 41, 42, 44–47, 50–52, 54, 81

Death Phase, 46, 49
DSDM, 18
Dynamic Systems Development Method, 18

ekstremno programiranje, 6, 11, 15, 18, 19, 22, 24, 25, 29, 32, 33, 36, 37, 43, 44, 55
Exploration Phase, 46, 47, 80
Extreme Programming, 6, 11, 15, 18, 19, 22, 24, 25, 29, 32, 33, 36, 37, 43, 44, 55

faza isporuke, 46, 48, 79
faza istraživanja, 46, 47, 80
faza od iteracija do isporuke, 46, 48, 79
faza održavanja, 46, 48, 49
faza planiranja, 46, 48
faza završetka, 46, 49
FDD, 15, 18
Feature Driven Development, 15, 18

HC Agent, 56, 59
heavyweight methodologies, 5
  thick methods, 5

iteracije, 26–30, 32–34, 38, 46–49
Iterations, 26–30, 32–34, 38, 46–49
Iterations to Release Phase, 46, 48, 79
Iterative and Incremental development, 11, 14
iterativni i inkrementalni razvoj, 11, 14

jedinično testiranje, 8, 16, 21, 22, 26, 31, 36, 39–42, 44, 45, 69, 87

korisničke priče, 25, 26, 28–30, 35, 38, 44, 46–49, 51

lightweight methodologies, 5, 6, 14, 19
  thin methods, 5

Maintenance Phase, 46, 48, 49
Manager, 50, 53, 81
model kaosa, 9
model prototipa, 11

naručitelj, 11, 13, 16, 17, 20–22, 25, 26, 28–30, 37, 39, 41, 42, 44–47, 50–52, 54, 81

Open Source Software Development, 18
OSS, 18

Pair Programming, 20, 21, 31–33, 35, 40, 42, 51, 54, 59, 60, 85
Planning Phase, 46, 48
proces razvoja softvera, 4, 11, 14
Productionizing Phase, 46, 48, 79
programiranje u paru, 20, 21, 31–33, 35, 40, 42, 51, 54, 59, 60, 85
Programmer, 50–52, 82
programski zahtjevi, 11, 19, 20, 22, 25, 27, 39, 42, 59
programsko inženjerstvo, 5

Project Velocity, 31
Prototyping model, 11

Rational Unified Process, 13, 18, 84
Refactoring, 31, 35, 36, 42, 45, 51, 52, 54, 60, 86
refaktoriranje, 31, 35, 36, 42, 45, 51, 52, 54, 60, 86
RUP, 13, 18, 84

Scrum, 18
software development process, 4, 11, 14
software engineering, 5
software requirements, 11, 19, 20, 22, 25, 27, 39, 42, 59
Spiral model, 9
spiralni model, 9

Tester, 50, 52, 83
Top Down model, 6, 9
Tracker, 50, 82

UML, 13
Unified Modelling Language, 13
Unit testiranje, 8, 16, 21, 22, 26, 31, 36, 37, 39–42, 44, 45, 69, 87
User Stories, 25, 26, 28–30, 35, 38, 44, 46–49, 51

vodopadni model, 7, 9, 14

Waterfall model, 7, 9, 14

Sažetak

Prilagodba metode ekstremnog programiranja za projekt razvoja javne elektroničke usluge

Agilne metodologije razvoja programskog proizvoda nastale su kao odgovor tradicionalnim metodama razvoja, primjenjujući specifične razvojne postupke koji se fokusiraju na brzi razvoj programskog proizvoda, uz zadržanu kvalitetu. Praksa pokazuje da tradicionalne i agilne metode ne moraju nužno biti isključive, te da postoji veliko i neistraženo područje gdje se ta dva pristupa mogu međusobno kombinirati i nadopunjavati.

Ovaj rad provodi analizu razvojne metodologije ekstremnog programiranja. Istražuje mogućnosti primjene razvojnih postupaka definiranih unutar te metode za primjer specifične projektne okoline i s naglaskom na programski proizvod. Posebna pažnja je posvećena testiranju, mogućnostima poboljšanja programskog kôda i integraciji programskih pomagala. Istraživanje je provedeno na razvojnom programskom projektu javne elektroničke usluge.

Ključne riječi

Agilne metode, XP (ekstremno programiranje), programiranje u pâru, jedinično testiranje, refaktoriranje, male i česte isporuke

Summary

Tailoring the method of Extreme Programming for public e-service development project

Agile methods of software development have developed as response to the more traditional development methods, applying specific development procedures which focus on rapid software development while containing the quality. Praxis shows that traditional and agile methods need not be exclusive and that there is a vast and unexplored area where those two approaches can be combined.

In this work was conducted the analysis of Extreme Programming development method. The possibility of application of development techniques defined whitin the method as an example of specific project encircling with an emphasis to the software are explored. Special attention was given to the testing, possibility of enhancements of program code and the integration of program tools. The research was undertaken on the example of public e-service development project.

Keywords

Agile methods, XP (Extreme Programming), Pair programming, Unit testing, Refactoring, Small and frequent releases

Životopis

Krešimir Maržić je rođen 07. srpnja 1977. godine u Zagrebu. Osnovno obrazovanje je započeo 1984. godine i završio u osnovnoj školi *Malešnica* u Zagrebu 1992. godine. Nakon položene mature u prirodoslovno-matematičkoj *Gimnaziji Lucijana Vranjanina* u Zagrebu, 1996. godine upisuje studij *elektrotehnike* na *Fakultetu elektrotehnike i računarstva* (FER) *Sveučilišta u Zagrebu.*
Diplomirao je iz područja *elektrotehnike* na smjeru *Telekomunikacije i informatika* 11. srpnja 2001. godine sa temom *Funkcijsko testiranje komponenata programske opreme* pod stručnim vodstvom prof.dr.sc. Branka Mikca.

Nakon završetka studija zapošljava se 2001. godine u *Institutu za telekomunikacije* (*Research & Development Center*) u kompaniji *Ericsson Nikola Tesla d.d.* u odjelu za istraživanje i razvoj novih Internet proizvoda i usluga na *HT Telex over IP* razvojnom projektu.
Trenutno radi u istoj kompaniji u *Centru za komunikacijska rješenja, usluge, logistiku i e-sustave* (*Operations*) na razvoju sustava primarne zdravstvene zaštite.

Autor je više znanstvenih radova objavljenih na međunarodnim konferencijama. Za rad *Testing Methodology for Primary Halth Care Enterprise System* dobio je priznanje za izuzetno istaknuti rad na savjetovanjima MIPRO 2004.

Poslijediplomski znanstveni studij iz *znanstvenog polja Elektrotehnike, smjer Telekomunikacije i informatika* je upisao u travnju 2003. godine.